Notes on the Plan 9™ 3rd edition Kernel Source

Francisco J Ballesteros

nemo@lsub.org

May 8, 2007

Contents

To my wife.

Trademarks

Plan 9 is a trademark of Lucent Technologies Inc.

License

To my wife.

Preface

The very first time I understood how an operating system works was while reading the source code of Minix. Years after, I had the pleasure of reading John Lions "Commentary on UNIX" along with the source code of UNIX v6.

Although time has passed, I still feel that the best way to learn how an operating system works is by reading its code. However, contemporary UNIX (read Linux, Solaris, etc.) source code is a mess: hard to follow, complex, full of special cases, plenty of compiler tricks and plenty of bugs. When Plan 9 source code was released to the public on its 3rd edition, I knew it was just the material I needed for my Operating Systems Design course. This commentary is an attempt to provide a guide to the source code of Plan 9 3rd edition.

The concepts included are those covered on the "Operating Systems Design" course of 4th year at Universidad Rey Juan Carlos de Madrid [2].

Any reader, specially when following the course, is encouraged to read the source along with the commentary as well as to modify and enhance the system.

Chapter 1

Introduction

An OS does mainly two things: it multiplexes the hardware and provides abstractions built on it. Plan 9 does it for a network of machines. The nice thing of Plan 9 is that it is centered around a single abstraction: the file. Almost everything in the system is presented as files. Therefore, most of the complexity lies on the "multiplexes the hardware" part, and not on the "provides abstractions" part. By not optimizing the system where it is not necessary, even the "multiplexes the hardware" part is kept simple (You should compare the source code with that of Linux if you don't believe this).

Before proceeding with the source code, I give you a piece of advice regarding how to read this document, which shouldn't be read as a regular book.

1.1 How to read this document

This commentary is that, a commentary to the Plan 9 kernel source. I have used the source for the June release of the third edition. It should be read like any commentary of a program, by keeping both the commentary and the source side by side. In fact, you should try to read the system code without reading the commentary. If while you are reading the code and the commentary, you feel curiosity about what else is done at a particular file, you should go read it all: remember that nobody can teach you what you don't want to learn.

The final goal is to understand the system, how it is built, what services it provides and how are them provided. As with any program, it is better to focus on the tasks the system has to carry out. Most of the commentary will be centered on them. Before understanding the code of the system, it is wise to take a view to the system as a user. You are strongly advised to read the manual pages relevant for each chapter, as well as to use the system either at the Plan 9 laboratory, at home, or at both. Ask for help if feel you can't install Plan 9 at home. Once you know the set of services provided, you also know that has to be implemented, and you will understand the code better.

While you read the commentary, you will see that I refer to the authors of the source code as "the author". Each time I mention the author, I am referring to the author(s) of the particular piece of code discussed. Plan 9 is the joint result of many people. The main authors of the code are Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. As far as I know, Rob Pike and Dave Presotto were the system architects; and Ken Thompson was the architect for the file server.

Also, whenever I say "he" or "his", you should understand that I am saying "he or she" and "his or her". I do not like typing so much and for me it is hard to find impersonal sentences where "he" and "his" can be avoided. So excuse me, no offense intended.

1.1.1 Coming up next

In the next section, I give you some pointers you should follow. They are mostly research papers about Plan 9. You should read them (well, at least you are expected to read the first one) to learn more about the system before looking at its internals. It is good to learn to follow the documentation pointers "on demand", as you feel you need to know more about a particular topic to understand the code.

What remains of this chapter is whatever I think is the bare minimum to understand the source code. Next section gives a quick introduction to reading code written in C, the language used for the Plan 9 kernel. You can skip this whole section but take a loot at it when you find something that is not "ANSI C" in the code. The following one is a quick introduction to PC hardware facilities.

Remaining chapters describe different topics of the system and can be read randomly, although it would be good to read chapter 2, about system source code organization, and chapter 3, on system startup, before proceeding with the following ones. Besides describing how the system boots, chapter 3 describes several important concepts to understand the design of Plan 9.

To save trees, the source code is not printed on paper. All chapters refer to code using pointers like `/dir/file.c:30,35`. They focus on a given line (or lines). These pointers can be used as "addresses" on the Plan 9 editors you will be using during the course. It is very convenient to print this commentary, open the `acme` editor[1] full-screen, and then follow the commentary giving the pointers to acme as they appear. It is even better to use a text version of the manuscript and open it on acme. Then you can jump to the source by clicking button 3 on the pointer. What? you don't know how to use acme? Don't worry, forget this and the next couple of paragraphs and reread them when you get started with acme. To get started you can read the paper on acme from volume 2 of the Plan 9 manual [7].

If you open the text version on acme, I suggest you execute these commands by using button 2 on them:

```
Local bind -a .  /sys/src/9/port
Local bind -a . /sys/src/9/pc
```

If you used button 2 to execute them, your namespace in acme will have been arranged so that the files in this directory appear to be also in the directories with the Plan 9 source code. This way, by using button 3 on file pointers acme will jump to the given location in a different window. So, now that your namespace is ready,

[1] As you will see, acme is much more than a editor; it is a full environment to do your daily work on Plan 9.

close this file, go to /sys/src/9/pc and open this file there. This document will be jumping to code in other directories (e.g. port); in that case, I suggest you simply edit the tag of the Acme window for this file to update its directory (e.g. so that the tag is /sys/src/9/pc/9.txt while reading files in the pc directory, and it is /sys/src/9/port/9.txt while reading files in the port directory). Do not Put this file.

More on acme advice, to get line numbers on a file, select it all by typing `:,` and using button 3; then type `| awk '{printf("%-5d\t%s\n",NR,$0)}'` (or `|cat -n` if you are on UNIX), select it and use button 2. Don't Put the file. To locate identifiers through the source you can create a script to grep the parameter in `*.[ch]`. For your convenience, a copy of the kernel source with line numbers is installed both at the Linux laboratory and at the Plan 9 laboratory.

Whenever we refer to a file, a relative path has as the working directory the directory where we are looking source files on Plan 9. Absolute path names start always at the Plan 9 root. If you are browsing on Linux, and Plan 9 is installed at /plan9, that means /plan9/absolute path name instead. Remember that if you use Linux you still have wily, an acme look-alike. You have also Inferno, where you have acme (See the web page for the "Advanced Operating Systems" course [1]).

I suggest you install Plan 9 on your PC and then use it to read the source code as I said before. By using the system you will "feel" how it works better, and you will use something that is neither UNIX nor Windows. There are excellent pieces of advice regarding how to install Plan 9 in volume 2 of the manual [10].

If you feel emotionally attached to Linux, you can at least install wily, an acme-look-alike for UNIX; but you will be missing something great.

When discussing a particular data structure or function, it is good to see where is it used through the system. To find that, you can use the grep program. By using it within acme, you can simply click with your (three button) mouse jump through the occurrences found by grep -n.

When a particular section of a classical OS textbook would further discuss a topic being addressed, a pointer of the form *[n]/section* will appear, where the "[n]" part is a reference to the bibliography. You do not need to follow this kind pointer immediately, although that might help you if you feel lost.

Several times I will be discussing code implementing a system call or used by a popular command. References such like man(1) instruct you to read the manual page on "man" on section "1" of the manual as a definitive reference on the program or system call discussed. You should at least browse the manual pages as they are cited; and you can skip parts that you don't understand there.

One of the abilities you are expected to learn is that of browsing through a reasonably sized piece of code or documentation. While doing that, remember that it is important to ignore at first things we don't understand and try focus on what you can understand. Of course, unless you know the not-understood part is not relevant for you, you should try to understand that part too, and ask for help if you can't.

1.2 Other documentation

The third edition of Plan 9 comes with a two volume programmer's manual [9, 10]. The first volume, "the manual", is the set of manual pages for the system. Manual pages are packaged into sections. There are several sections, including a section on commands and another on system calls and library functions. Manual pages are similar to that of the "man" command on UNIX, although the set of sections vary.

The second Plan 9 programmer's manual volume, "the documents" is a set of papers relevant for Plan 9. They discuss one aspect or another of the system. I expect you to read at the very least several ones, and I highly recommend you read all of them. You will find that papers on volume two are not like typical research papers these days, on the contrary, they are simple, show a new idea or a new way of doing something, and can be understood by themselves; moreover, they are implemented. Reading Plan 9 papers is a fine way of get a kind introduction to the system.

1.2.1 Manual pages

The manual [9] is divided in sections. When you refer to a manual page like `man(1)`, you are referring to the manual page for "man" on section 1 of the manual. Manual pages can be found at several places:

- Using the `man` command on Plan 9, like in `man 1 man`.
- Writing the name of the page (e.g. "man(1)") on the acme editor, and clicking on it with mouse button-2.
- running nroff on Linux for the manual page found at `/sys/man/manX/xxxx`. For example, if your Plan 9 tree is at `/plan9`, you can:

  ```
  nroff -man  /plan9/sys/man/1/man
  ```

  On the Linux laboratory, you have also the `9man` command that refers to the manual of Plan 9 installed on the Linux file system; and ignores Linux manual pages.
- Using your favorite web browser and looking at *http://plan9.bell-labs.com/sys/man*

If, as I recommended, you are using acme to read the source, method 2 is the most convenient one.

Now go, read `intro(1)`, and drink some coffee. Give yourself enough time to assimilate what you read there.

Done? Ok, if you are really done, you should now know that

- Section 1 of the manual is for general user commands. You type them on a shell, or click on their names with button 2 in acme.
- Section 2 is for library functions and system calls. This is the programatic interface to the system. You are studying how the system calls described here are implemented.
- Section 3 shows kernel devices, which supply "kernel files" you need to access to use the system. These files show up typically under `/dev`. You will be interested mostly on manual pages for devices we discuss.

- Section 4 has manual pages on file systems that you can mount. They are supplied usually by user programs that implement some service. For instance, access to FAT file systems is provided through a program that services a FAT file system—using the FAT partition as the storage medium.
- Section 5 shows how you talk to files on Plan 9. Plan 9 is a distributed system that permits remote access to files. This section shows the 9P protocol used for that purpose. It is at the very heart of the system. During the chapter on file systems, you should be reading this section.
- Section 6 discusses several file formats. For example, the format of manual pages is shown at `man(6)`.
- Section 7 addresses databases and programs that access them.
- Section 8 is about system administration. Commands needed to install and maintain the system are found here. Some of them will appear while reading the code, and you should read their manual pages.

Too many things to read? I recommend you read manual pages on demand, as they are mentioned on the commentary, or as you use the tools described on them. The very first time you use a new Plan 9 program or tool, it is good to take a look to its manual page. In that way, as you use the system, you will be learning what it has to offer.

1.2.2 Papers

Documents from the manual [10] can be found at several places too. You can use `page` on Plan 9 (or `gv` on Linux) with postscript files in the Plan 9 directory `/sys/doc`. You can also use your favorite web browser and look at *http://plan9.bell-labs.com/sys/doc*. These are the papers:

Plan 9 From Bell Labs is an introductory paper. It gives you an overview of the system. Reading it you will find that Plan 9 is not UNIX and also that networks are central to the design of Plan 9.

You are expected to read this one soon.

The Use of Name Spaces in Plan 9 gives you more insight into one key feature of Plan 9: every process has its own name space. You can think that every process has a "UNIX mount table" for itself; although that is not the whole truth.

You are expected to read this one.

Getting Started with Plan 9 is an introductory document with information you need to know to start running Plan 9.

The Organization of Networks in Plan 9 shows how networking works on Plan 9. The section on Streams is no longer relevant (Streams are gone on 3rd edition), although it is worth reading it because the spirit remains the same.

How to Use the Plan 9 C Compiler will be helpful for you to do your assignments. Once you know how to use C, this paper tells you how to do it on Plan 9. More on this on the next section.

Maintaining Files on Plan 9 with Mk describes a tool similar to `make`. It is used to build programs (and documents) on Plan 9. This paper will be also of help for doing your assignments; as you are expected to use both C and mk. More on this on the next section.

The conventions for using mk in Plan 9 is also good to read. It shows how mk is used to build the system. This paper can save you some time.

Acid: A Debugger Built From A Language is an introduction to the debugger. You will find that it is not similar to the kind of debuggers you have been using, and it is highly instructive to debug using Acid.

Acid Manual is the reference manual for the debugger.

Rc The Plan 9 Shell shows you the shell you will be using. If you have used a UNIX shell that is probably all you need. You can learn more of `rc` as you use it. Remember that `rc` is installed also on Linux.

The Text Editor sam describes the editor used on Plan 9. It is a fine editor although you can go with acme instead. Indeed, I heavily suggest you start by using Acme. Of course, it is healthy to try sam too. The `sam` editor is installed on Linux too.

Acme: A User Interface for Programmers describes the Acme editor. Well, as the title says, it is more like an environment. You have a clone for Linux called `wily`. I used Emacs for years until I found acme, and the same may happen to you. You should read this document and play with acme or wily, to navigate the source code. By the way, it is named "acme" because it does everything.

Installing the Plan 9 Distribution is something to print and keep side by side with the keyboard if you intend to run Plan 9. The title says it all.

Lexical File Names in Plan 9 or Getting Dot-Dot Right describes how file name resolution works despite the existence of the `bind(2)` system call. Read this before you read the chapter on Plan 9 files.

There are several other papers, good to read too, that I have omitted here for the sake of brevity.

I recommend you fork now another process in your brain and read all of them in background. Whenever I feel its better for you to read first any of them, I will let you know.

1.3 Introduction to Plan 9

This section intentionally left blank[2]

[2] Read "Plan 9 From Bell Labs" [11] instead. The paper is so clear that nothing else has to be said.

1.4 Source code

Plan 9 is written in assembly (only a few parts) and C. You *must* read C code to understand how the system works. Moreover, you are expected to write your own C code to modify the kernel in your assignments.

This section will introduce you to C to let you read it. Nevertheless, you should read the following two books if you have not done so:

The C programming Language, 2nd ed. is a good, kind, introduction to the language [5]. It is easy to read and it pays to do so. The compiler used on the book is the UNIX C compiler. You can find how to use the Plan 9 C compiler on the paper "How to use the Plan 9 C compiler" [8]—these guys use descriptive titles, don't do them?

The practice of programming is a "must read" [4]. It will teach you those things you should have been taught during the programming courses.

Now that you have pointers, I will first comment a bit of Plan 9 C, then a bit of how to use `mk` to avoid the need to manually call the C compiler.

1.4.1 Notes on C

I include this section on C mostly to document a few differences with respect to ANSI C, and for the sake of completeness. But I *really* recommend you to read *The C programming Language* [5] if you don't know C yet.

Where is the kernel C code?

The system source code is structured as a set of directories contained on `/sys/src/9`. Although there are valuable include files at `/386/include` (and similar directories for other architectures) and `/sys/include`. You will be using mostly these directories:

/sys/src/9/pc contains machine dependent code for PC computers. This code assumes that you are running on a PC.

/sys/src/9/port contains portable code. This code is shared among different architectures.

/sys/src/9/boot contains code used to bring up the system.

Other directories under `/sys/src/9` contain source for other architectures, but for the `ip` directory—which contains a TCP/IP protocol stack. For Plan 9 file systems, the kernel source code is found at `/sys/src/fs` instead. Subdirectories of `fs/` follow the same conventions that subdirectories of `9/`. I do not comment the file system kernel, it is a specialized kernel (borrowing a lot from the generic kernel) designed to serve files fast.

C and its preprocessor

If you take a look to any of those directories, you will find files named “xxx.c”, “xxx.h”, and “sss.s”. Files terminated on “.s” are assembly language files. They contain low-level glue code and are used where either C is not low-level enough to let the programmer do the job, or where it is more natural to use assembler than it is to use C. Files named “xxx.c” and “xxx.h” are the subject of this section: they contain C source code.

The C language has a compiler proper, and a preprocessor. Files are first fed to the preprocessor, which does some work, and the result is finally sent to the compiler. The compiler generates assembly code that will be translated to binary and linked into an executable file. On Plan 9, the compiler is usually in charge of preprocessing the source too, so a single program is run on the source; nevertheless, you better think that source is first fed into the preprocessor and the result goes automatically to the compiler proper.

C source files can be thought as “implementation modules” or package bodies. H source files can be thought as “definition modules” or package specs. When someone writes a C module to be used on a program, the module has a header file (a “.h” file) with declarations needed to interface to the module and a C file (a “.C” file) with the implementation.

Consider these three files:

```
/* this is main.c */
#include "msg.h"
main()
{
   Msg m;
   set(&m,"Hi world!");

   print(get(m));
}
```

```
/* this is msg.h */
typedef char *Msg;
void set(Msg *m,char*s);
char *get(Msg m);
```

```
/* this is msg.c */
#include "msg.h"

void set(Msg*m,char*s){
  *m=strdup(s);
}
char*get(Msg m){
   return m;
}
```

The point to get here, is that `msg.h` has the interface to `msg.c`. It contains a type definition for `Msg` and the header of a couple of functions. The main program (always called `main` in C) can include these definitions and then use them. Files “main.c” and “msg.c” can be compiled separately into object files and then linked together.

When compiling `main.c`, the preprocessor will notice the `#include` directive and replace it by the set of lines found in the named file (`msg.h` in the example). It is textual substitution. The preprocessor knows nothing about C. The resulting (preprocessed) file would be sent to the C compiler proper. In the example, some includes must be missing since there is no prototype for the `print` function (and the same happens with `strdup`).

Another useful preprocessor directive is `#define`, which lets you define symbols. Note that again, this is textual substitution—the preprocessor knows nothing about C.

```
#define SPANISH 0
#define ENGLISH 1
extern int lang;
```

```
void hi() {
     if (lang == SPANISH)
          print("hola");
     else print("hello");
}
```

After the `#define` lines, the preprocessor will replace any “SPANISH” text with “0” and “ENGLISH” with “1”. The compiler will see none of these symbols.

Functions

C has no procedures: every subroutine is a function. The result of a function may be ignored though. Look at this function:

```
int add(int c, int l)
{
    return c+l;
}
```

It receives two integer parameters named “c” and “l”—parameters are always passed by value on C. It returns an integer value (the first “int” before the function name). The “return” statement builds the return value for the function and transfers control back to the caller routine.

A function can return “void” (which means “nothing”) is provided to let a function return nothing. There is another use of `void`, a pointer to `void` is actually a pointer to anything.

When a parameter passed to a function is not used, you can declare it without a name, as in

```
int add3(int c, int)
{
    return c+3;
}
```

This is not allowed by ANSI C. In this example, of course, it is silly to declare a parameter and not to use it. However, when a function should present a generic interface, and a concrete implementation of the function does not need a particular parameter, it is wise to leave the interface untouched and not to use the parameter.

For instance, imagine that to open a file you should use a function with this prototype:

```
int open(char *name, int just_for_read);
```

Now imagine a particular file on a CDROM is opened. In the implementation, it is not necessary to specify the open mode because it has to be “read-only”. Now look at this function:

```
int open_cdrom_file(char *name, int)
{ ...
}
```

It implements the above interface, and has an unused parameter.

To use a function it suffices to know its header. We can know it because we `#include`d a file where the header is kept, or because we are calling the function after its implementation.

Of course, functions can be (mutually) recursive.

Data types

There are several primitive data types: `char` for characters, `int` for integers, `long` for long integers, `long long` for longer integers, `double` for real numbers. These are signed, and you have types defined with a leading "u" for the unsigned versions (e.g.: `ulong`, which is actually `unsigned long`).

Arithmetic operators are as usual, with the addition of `++` and `--` which increment and decrement the operand. They may be used either prefix or postfix. When prefix, the argument is incremented (or decremented) prior to its use in the expression; when postfix, the argument is modified after used for the expression. For instance, `i++=j++` means:

```
i=j;
i++;
j++;
```

whereas `++i=++j` means

```
i++;
j++;
i=j;
```

The modulus operator is "`%`". Assignment is done with `=`. Assignment is sometimes "folded" with another operator. For instance, `i%=3` means `i=i%3`, `x|=0x4` means `x=x|4`, which does a bitwise OR with for. '&' is the bitwise AND operator. The operator "`~x`" negates each bit in `x`, so `x=~x` inverts every bit in `x`.

Booleans and conditions

There is no boolean in C, any non-zero integer value (or convertible to integer) is understood as "TRUE". Zero, means false.

Relational operators are `==`, `!=`, `<=`, `>=`, `<`, `>`, where `!=` means "not equal". You can use `!` to negate a boolean expression. More complex boolean expressions can be built using `&&` (and), `||` (or) and `!` (not). Once the compiler knows that a boolean expression will be true (or false) it will not evaluate the rest of the expression. Some would say that C has shortcut evaluation of boolean expressions. For those of you how know Ada, in C you are always using "and then" and "or then". For instance: on `1 || f()`, function `f` would never be called. The same would happen to `0 && f()`. This is very useful because you can check at a pointer is not nil and dereference it within the same condition.

As Plan 9 is meant to run anywhere in the world, it has to cope with every language. A Rune data type is defined (it is actually an `unsigned short`) to support strings of "characters" in any language. `Rune` is used to represent a character or a symbol, hence the name (some languages use symbols for words, or lexems). Remember that `char` has only 256 values which are not enough to accommodate symbols on all languages. The character encoding system is called Unicode, encoded using UTF-8. UTF-8 is compatible with the first 128 ASCII characters, but beware that it will use several bytes when needed. And beware too that it is not compatible with ISO.8859.1 that you use on Linux.

Given primitive types, you can build more complex types as follows.

pointers A pointer to a given type is declared using `*`; e.g. `char *p` is a pointer to character. You refer to the pointed-to value by using also `*`, like in `*p`—which is the character pointed by `p`.

The operator `&` gives the address of a variable; hence, given `int i`, the declaration `int *p=&i` would declare a pointer `p` and initialize it to point to `i`. A function name can be used as a pointer to the function, like in

```
int (*f)(int a, int b) = add3; /* the previous add3 function */
...
g=(*f)(1,2);
```

arrays An array in C is simply a pointer with some storage associated, do not forget this. For instance, `char s[3]` declares an array named `s` of three `char` slots. The slots are `s[0]`, `s[1]`, and `s[2]`. Array indexes go from zero to the array length minus one.

Arrays can be initialized as in

```
int arry[3] = { 10, 20, 30 };

int tokens[256] = {
    ['$']    DOLLAR,
    ['/']    SLASH
};
```

The last example is an array of 256 integers. We plan to index it using a character (which is a small integer in C). And we only initialize slots corresponding to characters `'$'` and `'/'`. This initialization style is an addition to ANSI C in Plan 9, and it is very useful: instead of using conditionals to check for dollars and slashes and generate a number, we can spend a few extra bytes and allocate one array holding an entry for every character; for those of interest, we place there the values desired; for others, we don't care.

Because arrays are actually pointers with some storage, C has pointer arithmetic. Assume this

```
int a[3];
int *p=a;
```

Here, `p` points to `a[0]`. Well, `p+1` is a pointer pointing to `a[1]`, `p-1` would point to the integer right before `a[0]`.

Also, if `p` points to `a[1]` and `q` points to `a[0]`, then `p-q` would be `1`: the number of elements between the two pointers.

As you can imagine, these to expressions are equivalent: `a[i]` and `*(a+i)`.

Beware, `p=a` will copy pointers, not array contents. Use `memmove` to do that.

structures An struct is the equivalent of a record in Pascal. It is declared by giving a set of field declarations. E.g.:

```
typedef struct Point Point;
struct Point {
    int x;
    int y;
};

Point p = (Point){3,2};
Point q = (Point){ .x 3, .y 2 };
```

declares a new struct tag, `Point`; declares a point `p` of type `Point`; initializes it with a copy of the Point {`3,2`}. "Literals" of structures are called "structure displays" in Plan 9's C. They are an extension to ANSI C.

In the example above, `struct Point` {`...`} declares the structure with an structure tag `Point`, so that you can say `struct Point p`. But it is customary to give a synonymous for the new type "`struct Point`" by using `typedef`. `Typedef` defines a new name (the `Point` on the right in the example) for an existing type (`struct Point` in the example).

Once `p` has been declared, `p.x` and `p.y` are names for the members (i.e. fields) of the `p` structure.

Structures can be nested like in:

```
struct Line {
    Point origin;
    Point end;
};
```

And we would say `l.origin.x`, provided that `l` is a `Line`.

If a struct, member of another struct, has no name its members are "promoted" to the outer struct. That is to save some typing. For example:

```
typedef struct Circle Circle;
struct Circle {
    Point;        /* has no name! */
    int radius;
};
Circle c;
```

And we could say things like `c.x`, `c.y`, and `c.radius`. Both `x` and `y` come from `Point`! You can even say `c.Point`, although it would be tasteless on this case. Member promotion and unnamed fields are an extension to ANSI C.

When you have a pointer to an structure, you can refer to members of the structure in several ways:

```
/* The way that you should know by now: */
Point *p;
(*p).x =3;

/* A more convenient way */
p->x = 3;
```

Everybody uses the `->` form, and nobody uses the `(*x).y` form.

Unions A union is a struct where *only one* of its fields will be used at a time. It is used to build variant records, although it is a bit more flexible. For instance:

```
typedef struct Vehicle Vehicle;
struct Vehicle {
#define CAR 0
#define SHIP 1
    int kind;
    union {
        int number_of_wheels;
        char can_go_underwater;
    };
};
```

Note how we used an anonymous union (one with no name) within `Vehicle`. We can use either one field or another of the union: they will be sharing storage. We cannot use both at the same time.

More examples:

```
/* array of 4 points; to use like in: arry[3].x, arry[2].y, etc. */
Point arry[4];
/* A polygon that contains the number of points, and an array with them. */
struct Polygon {
    int npoints;
    Point points[MAXPOINTS];
};
```

Control structures

Control structures are very easy to learn. All of them use sentences as their building blocks. Sentences are always terminated by semicolons. Several sentences may be grouped to form a block using { and }. At the beginning of a block, you may declare some variables.

while To repeat while the condition holds:

```
while(*p){
    *q++ = *p++;
}
```

A variant tests the condition at the end and iterates at least once:

```
do {
    x[i]++;
    x[j]--;
} while(i != j);
```

for is a generic iteration tool.

```
for (i=0; i<n; i++){
        x[i]=i;
}
```

means

```
i=0;
while(i<n) {
    x[i]=i;
    i++;
}
```

You specify the initialization, the continuation condition, and the re-initialization to prepare for the next iteration.

conditionals Well, you know

```
if (condition) {
        then-part;
}
if (condition) {
        then-part;
} else {
        else-part;
}
switch (variable) {
case A_VALUE:
    do this;
    break;
case OTHER_VALUE:
    do that;
    break;
default:
    do the default;
}
```

On the `switch`, you need to place `break`s at the end of every branch to avoid "falling through" the next branch.

Gotos are easy, you define a label and then can branch there. They are used inside a function. The label here is `error`.

```
    while(getinput()){
        processit();
        if (haserror())
                goto error;
    }
    return result();
error:
    abortprocessing();
```

Storage classes

When the compiler gets a new symbol, either a function or a global variable it attaches an "storage class" to it. The storage class determines among other things whether the symbol should go in the symbol table for the object file or not. In other words, you can export a symbol to the rest of the program, or keep the symbol private to a given file by using a particular storage class.

Symbols declared `extern` (they will be by default) are exported to other object (read "source") files. If they are not initialized, they are initialized to all-zeroes (by the loader). For instance:

```
/* this is file a.c */       /* this is b.h */       /* this is b.c */
#include "b.h"
                             extern int a;           int a=3;
int f()
{
    return a;
}
```

The function `f` will return `3`, unless `a` be modified.

On the other hand, if a symbol is declared `static`, its scope will go from the place of the declaration to the end of the file. Globals and functions to be used just within a source file, are declared `static`.

How to compile?

Ok, but how do you compile source code on Plan 9? The compiler is actually a compiler suite. When you compile you use one of the compilers, according to your target architecture (usually, `$objtype`). The compilers making the suite are named `Xc`, where the `X` identifies the architecture. For instance, you will be using `8c`, which is the compiler for `386` machines. But you could use `kc` (for the sparc) instead.

The assembler and the linker follow the same convention: `8a` is the 386 assembler and `8l` the linker.

To compile `foo.c`, you simply call the compiler `8c foo.c`.

As you will see in the next section, `mk` is used to automatically compile the source for your `$objtype`, or for the whole set of supported platforms.

By following simple naming conventions (“8c”, “8a”, “8l”) the compiler enhances portability, rather than enlarging the differences between different machines. Conventions are important in that they can simplify your life and allow the automation of tasks. I hope you will appreciate it on Plan 9.

1.4.2 mk

The program `mk` is a successor of `make`. If you don’t know make, don’t worry. `mk` simply instructs the machine to build certain “products” by means of source files.

`mk` uses a file named `mkfile` to learn what product(s) should be built, and how it should be done. For each directory with sources to build a product there use to be a `mkfile`. If you want to build the product for that directory, you only need to call `mk` there.

This is an (edited) excerpt from
/sys/src/9/pc/mkfile, the `mkfile` used to build the kernel for the pc:

```
CONF=pc              #defines the variable CONF to have the value "pc"
objtype=386          #building on intel
</$objtype/mkfile    #use the variable just defined to include the mkfile
                     #that contains machine dependent definitions
                     #(e.g. which one is the C compiler for this platform)

DEVS=`{rc ../port/mkdevlist $CONF}
                     #defines the variable DEVS with the string printed
                     #by the shell command within brackets. The command
                     #uses the variable CONF defined above.
                     #The string will be the list of object files for
                     #drivers on this platform
OBJ=$DEVS
        # several lines deleted here...
        # defines the variable OBJ with the list of object files,
        # which includes the list of object files for drivers.

#see below what this means....
plan9pc:     $OBJ
        $CC $CFLAGS $CONF.c
        $LD -o $target -l $OBJ $CONF.$O

#other rules follow....
```

By including `/$objtype/mkfile`, `mk` defines the actual compiler, assembler, and linker for the current architecture. You achieve portability not by using a single compiler that works for everyone; you achieve portability by following the same rules everywhere, and by keeping a set of simple compilers for all supported platforms. You port Plan 9 to a new architecture by adding a new compiler, assembler, and linker;

not by modifying the existing ones[3].

The last three lines of the excerpt are a *rule*. Rules tell mk how to build things using other things. In this case, the target of the rule (the product) is plan9pc. To build this, mk will need those files listed in $OBJ. Should those files be missing, mk will look in the mkfile how to build them. Once the dependencies are satisfied (i.e. the files in $OBJ do exist, and are up-to-date with respect to their sources), mk will use the two last lines to build the product. The rule says that we need $OBJ, and indeed the commands used within the rule do use $OBJ. The rule also uses variables to specify which C compiler ($CC) should be used, and the same for the linker.

The variable $target is defined by mk to be the current target for the rule being processed.

If you go to the directory /sys/src/9/pc and call mk, it will see the first rule in the mkfile. As its target is plan9pc, it will try to build it by recursively obtaining targets following the rules.

You will not need to write mkfiles until you start with your lab assignment; therefore, what I said about mk is enough for you to proceed. Nevertheless, I recommend you put the paper on mk early in your list of "to-read-things" so you know it well before you need it.

1.5 PC hardware facilities

In this section, I will briefly describe the hardware facilities found at modern Intel based PCs. The aim is to let you know enough of what the hardware provides so you could understand the software. For a complete description, I suggest you refer to the Intel manuals [3]. What I am describing here applies for processors from the i386 up to the most recent (32 bit) one. If you already know how the Intel 386 works, you will notice that I am oversimplifying many things: forgive me, but I am most interested in the the software.

When it is reset, the processor operates into what Intel calls "Real-address mode". On this mode, the Intel is emulating an old 8086, a 16bit machine. In real mode, the only virtual memory facility is segmentation: no paging. One of the first things all modern OSes do on Intels, is to quickly set the processor into what intel calls "protected mode", which is the native mode of the processor. In protected mode, the machine is using 32bit words, as it should. Most of the processor structures a describe below are used while in protected mode.

1.5.1 Registers

The processor has eight 32 bits "general purpose" registers called eax, ebx, ecx, edx, esi, edi, ebp and esp. They are not really "general purpose" as there are instructions that operate on some of them. For example, esp is the stack pointer register, and ebp is used as a frame pointer register (to point to the activation frame for functions in the stack). eax is generally used as the "accumulator" register. esi and edi registers are used by string processing instructions, which repeat a given operation on a series

[3] Of course, if you plan to do so, you should reuse the existing source. The machine dependent part of the compiler is the only thing that needs to be done

of bytes in memory; they are used as indexes into memory (source index/destination index). Because the processor also has real mode, there are register names `ax`, `bx`, `cx`, and `dx` which refer to the 16 bit version of the (actually 32 bits) registers. Bytes within this 16 bit registers, can be addressed by using names such like `ah` and `al`.

Two other registers are `eip` (the program counter) and `eflags` (status flags). `eflags` is used both to keep arithmetic and logic flags (e.g. to control conditional tests) and also to keep status bits for the processor (e.g. interrupt-enable). I will mention some flags later.

There are also six segment registers: `cs` is the code segment register, `ds` is the data segment register, `ss` is the stack segment register. The other three ones (`es`, `fs`, and `gs`) are not so tied to instruction execution as the first three ones. Instruction fetching is done from the segment described by `cs`, operands and data is fetch from the segment described from the `ds` register (unless specified otherwise), and stack instructions work using the `ss` register.

Other registers are the `tr` (task register), used to point to a TSS (task state segment); `cr2`, used to place there the faulting address on page faults; `cr3`, used to point to the page table; `gdt` (global descriptor table) used to point to a table with segment descriptors; `idt` (interrupt descriptor table) used to point to a table with interrupt descriptors; and other ones that I omit.

1.5.2 Instructions and addressing modes

The paper on volume 2 of the manual, "A manual for the Plan 9 assembler" [6], can be of help here. I suggest you read that paper when you have problems following assembly code. Nevertheless, to help you a bit, I reproduce here the information you are likely to need just to understand the code.

The assembler uses 32bit registers, named `ax`, `bx`, `cx`, `dx`, `sp`, `bp`, `si`, and `di`. Note the convention! which is different from the one I used while describing the Intel register set[4].

Several registers are invented by the assembler. An important one is `fp`, which appears as the "frame pointer" register, and points to local storage area to keep procedure parameters. For example, `0(fp)` is the first parameter (int), `1(fp)` the next, and so on. If you don't understand what "`0(fp)`" means, read it like `fp[0]` by now.

The set of instructions is the set found at the intel manual (see [3]), with `b`, `w`, or `l` appended for operations using bytes (8bits), words (16bits), and longs (32bits). You can use names like `ah`, `bh`, etc. to access the high part of the 16bit version of `ax`. Assignment order is left to right. For example:

movl ax, bx Move ax into bx.

movb ax, bx Move low 8bits from ax into low 8bits at bx.

movb ah, bx Move high 8bits in the low 16bits of ax to bx.

Although Intel forgot to implement instructions for several combinations of "movs", the Plan 9 compiler suite emulates such instructions, and you can forget about that.

[4] I used the typical convention found in Linux and popular assemblers for the PC.

Some instructions are "invented" by the assembler. For example, `text` is used to define a procedure. Its parameters are the procedure name and the number of cells to be used for local variables in the procedure stack frame. Parameters are passed in the stack, and the result value from the function is passed in `ax`.

There are several addressing modes used in the Intel.

ax The register `ax`.

$b Immediate value `b`.

(ax) The cell whose address is in `ax`.

10(ax) The cell whose address is found by adding 10 to contents of `ax`.

(ax)(bx*4) A cell in a table starting at the address in `bx`, with 4 cells per entry, using the contents of `ax` as an index.

2(ax)(bx*4) Works in the same way, but adding 2 as an offset.

Although Intel allows you to specify which segment to use for a given address, the Plan 9 assembler uses always `cs` for the code segment, `ds` for the data segment, and `ss` for the stack segment—usually, `ss` is the same of `ds`. Things are simple!

Beware that names for conditional branch operations follow the names of the 68020, and not the Intel ones. But this should be clear while you read the code.

1.5.3 Memory

An Intel address is specified as a 32 bit address (4G address space) on a given segment. To specify a segment, you must tell the instruction which one to use (`cs`, `ds`, etc.). You can tell the instruction either by defaulting to `cs`, `ds` or `ss`, or by specifying the segment register to use.

To load a segment register to specify a particular segment, you use a load instruction that is given a "segment selector" as its operand. The selector is an offset into the GDT (global descriptor table). Each entry in the GDT specifies a segment base address, limit, and protection. These descriptors are loaded (by using that offset) into segment registers, which are used later by the instructions. The whole picture is shown at figure 1.1.

In Plan 9, there are segment descriptors in the GDT for kernel and user text (code) and data. The stack segment is usually set like the data segment (same protections).

The processor runs at a given privilege level, from 0 to 3, as specified by the privilege level of the running text segment. As segment descriptors include a privilege level, they can be used to prevent a non-privileged segment (e.g. ring 3) from using code/data placed at a privileged segment (e.g. ring 0). Plan 9 keeps the kernel within protection ring 0, and user code and data within protection ring 3.

Apart from protection, segments are not used. This means that their base address is usually set to zero, and their limit set to the maximum. As an address is resolved by adding the segment base to the 32 bit offset, it turns out that the address is actually the 32 bit offset.

Intel refers to addresses resulting from the segmentation unit (i.e. after the segment base has been added) as "linear addresses". Plan 9, as almost every one else, uses linear addresses as its virtual addresses, by using base zero for segment.

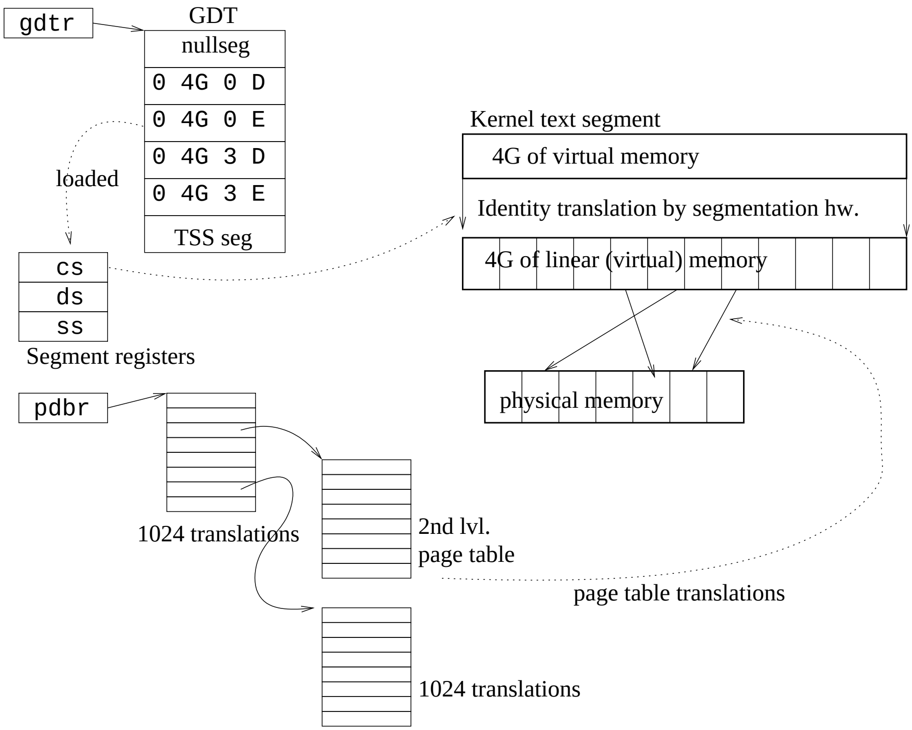


Figure 1.1: Virtual memory in the Intel processor. GDT entries contain base, limit, ring, and kind for all segments. GDT segment descriptors can be loaded into segment registers. The processor applies segmentation using segment registers and then applies paging.

A linear address is later fed into the paging unit, which uses a two-level page table pointed to by `cr3`. Pages (therefore page frames too) are 4K bytes long. The first-level page table (PD, or page directory) keeps 1024 entries, mapping 4Mbytes each. The Intel can use a PD entry to map a whole 4Mbyte "super-page" to a 4M "super-page-frame". The second level page table has 1024 entries too, mapping 4Kbytes each.

1.5.4 Interrupts and exceptions

Both interrupts (e.g. clock) and exceptions (e.g. page faults) are handled by the hardware with the help of the IDT (interrupt descriptor table). The IDT contains "interrupt descriptors". When a trap or interrupt happens, the hardware uses the exception number to index into the IDT and see where is the handler for the event. Each descriptor contains a privilege level too, which determines which protection rings may use them by instructions like `int`, which causes an interrupt event. A trap or interrupt may cause a privilege level change in the processor, if the handler's ring is more privileged that the caller's ring. For instance, to implement the system call mechanism, Plan 9 sets the `SYSCALL` entry with privilege level 3, so that the user ring can use `int` instructions to cause a trap. After the `int`, the system is running in ring 0, where the kernel executes. See figure 1.2 to get a glance of the structures involved.

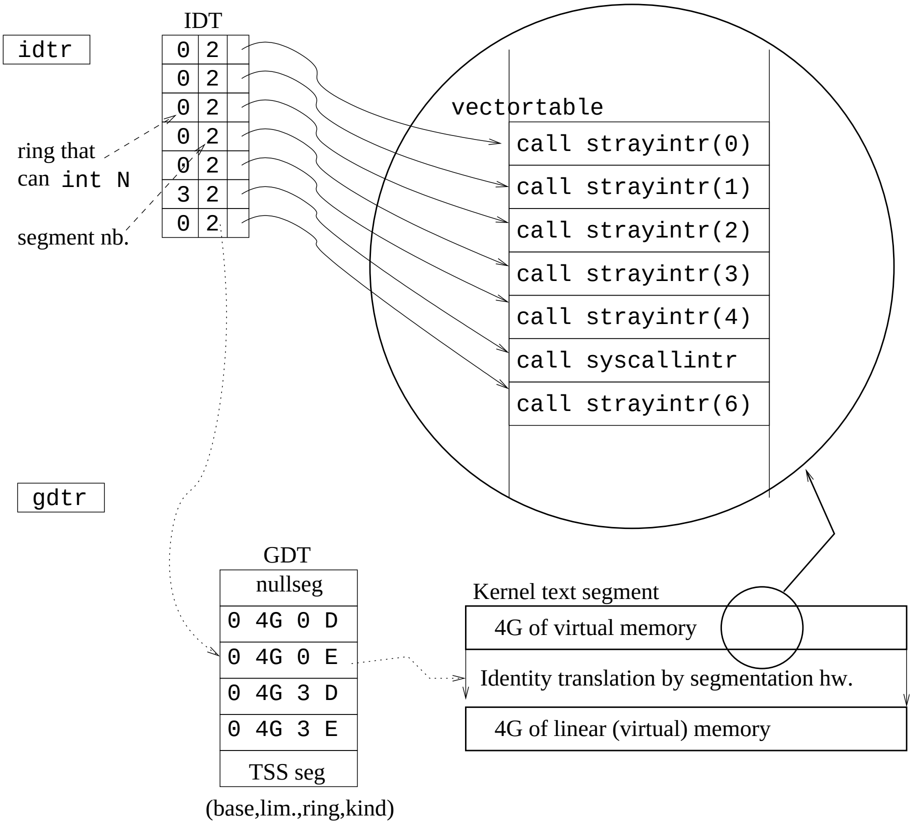


Figure 1.2: The `idtr` register points to an IDT table, used to vector interrupts. Each interrupt handler runs in a segment as described by a field in the IDT entry, which selects a descriptor from the GDT table. The IDT table shown here has been simplified (less entries than the real one).

There is one weird data structure, the TSS (task state segment), used by Intel to describe tasks. It specifies the processor state for a task and can be used to do context switching. The processor uses it to see what stack should be used for handling exceptions at the various protection rings. Plan 9 uses it to specify which kernel stack to use in case an event occurs while running at ring 3. While in ring 0, the processor uses the current stack (`ss` and `esp`) to save the processor state upon exceptions.

Chapter 2

System source

I will give you a quick tour through the source code first, and then, in the next chapter we will start to see the set of data structures used through the system as we learn how it boots. Therefore, in this chapter we are going to study the overall distribution of the source code of Plan 9.

2.1 Quick tour to the source

The source code of the kernel for the PC can be found in the `/sys/src/9/pc` directory (machine dependent part) and also in the `/sys/src/9/port` directory (portable part). The code for the kernel loader on PCs is at `/sys/src/boot/pc`.

2.1.1 Interesting include files

Almost every source file includes `u.h`, found at `/386/include`. It contains definitions for common data types and symbols for the 386 (e.g. `uint`, `ulong`, etc. It includes also several macros to handle variable argument lists on function calls (e.g. like that of `print`). It is defined here below `/386` because it assumes a particular stack layout which is only guaranteed to work for the Plan 9 compiler on the 386.

Another interesting file here is `ureg.h`. It contains the definition of the 386 register set.

The directory `sys/include` contains include files for users the system. The most interesting one is `libc.h`. This file contains definitions for the set of available system calls and utility functions of the C library. There are others, but the most important are placed here. The set of function prototypes starting at `_exits` is the set of system calls. You can look at section 2 of the manual to see what they do.

2.1.2 Interesting source files

The machine dependent part of the system is the one that “boots” it and uses services provided by the portable part. The same holds for the compilation. The machine dependent part contains the `mkfile` necessary for compiling the kernel. As compilation proceeds, it uses both headers and C files from the portable part to build a kernel image. The best way to see this is to compile a kernel:

Compiling a kernel

Go to `/sys/src/9/pc`, and type `mk CONF=9pcdisk`. You will see how a new kernel is built. The `CONF=9pcdisk` sets a variable for `mk`. It tells `mk` that `9pcdisk` is the configuration file to be used for the kernel.

The configuration file contains a description of the devices that should be linked into the system. An `rc` script uses this file to generate source code that initializes and starts these drivers. The `mkfile` includes also a list of relevant object files to link into the system.

If you take a look to the `mkfile` you will see how it includes the portable code `mk` file, `../port/portmkfile`. The `portmkfile` found there assumes it will be used from a machine dependent directory. In that way, different kernels can be built that pull up code from the `../port` directory. Again, the machine generates (first two rules of `../port/portmkfile`) the list of files to be built into the kernel. This includes also the source of tcp/ip, found in `../ip`. Because all compilation process from the machine dependent directory, both `port` and `ip` have a leading `../` when used to locate files.

The rules using the variable `CONF` in `portmkfile` generate source code from the configuration file using `mkdevc` and compile it.

> **Lesson:** whenever the machine can do something for you, like generating the source code that you should write by hand, let the machine do the job.

You can see how there are a bunch of `rc` scripts named `mk...`, that generate source code which can be generated mechanically. Let the machine do the job!

The machine dependent directory

Go now to `pc`. files named `dev...` contain device code. The code for the device named `ether` in the configuration file goes into `devether.c`. Naming conventions are important in Plan 9, they allow the automation of tasks, like generating the list of file names where drivers configured are to be found. Other files like `cga`, `dma`, etc. contain code to handle machine dependent facilities like the video card.

Two important files are `dat.h` and `fns.h`. To avoid the problem of circular include files, data structure and function definitions for the machine dependent part are placed here. The common and crucial stuff is found here. There are other definitions, more self-contained, that have their own include files. For example, memory management definitions are found at `mem.h`.

An important source file is `l.s`, that contains the low-level glue code. This code contains the entry points into the system. As examples, `start` is where the system starts executing, `inb` and `outb` do byte IO on IO ports, and `strayintr` is where the kernel starts executing after the hardware gets an interrupt/exception. System calls are also "exceptions" as far as the hardware is concerned.

Part of the low level glue is in `plan9l.s`, although it could go perfectly on `l.s`. `plan9l.s` contains the couple of routines that call to user code and return from a system call.

Another important source file is `main.c`. Once the assembly code has things set up enough, it calls `main` to start the system. Most of system initialization goes here. We will be looking through `l.s` and `main.c` in the next chapter.

Regarding memory handling, `memory.c` is in charge of allocating memory within the kernel for different purposes, and `mmu.c` is in charge of handling paging (virtual memory) facilities.

The file `trap.c` contains C code to handle traps. That code is called from `l.s` once the hardware jumps into `l.s` code to handle traps. Although `trap` could call `syscall` to handle system calls, `plan9l.s` calls `syscall` directly. It dispatches to the appropriate system call.

Most other files can be ignored by now.

The portable directory

Most system services are found here. "Abstract" devices that do not operate on real hardware, processes, virtual memory, and files along with several other things.

Several files contain portable utilities: `alloc.c` and `xalloc.c` contains portable memory allocation routines, `alarm.c` alarms, `cache.c` is in charge of caching, `taslock.c` and `qlock.c` implements locks, `qio` implement queues for block IO, etc.

There are also "portable" devices like pipes, the console, etc. defined in files named `dev...`. We will detail some of them, but you can ignore them by now. As it happens with machine dependent device files, an important part of the code provides hierarchies of files to export the devices to the user. For example, `devproc.c` implements files seen on `/proc`, which represent system processes.

Communication channels are implemented in `chan.c`. Channels are central in Plan 9. They represent an IO endpoint to a file. In other words, channels are "files begin used". Remember that everything is a file, therefore, this abstraction is really important.

Files `portdat.h` and `portfns.c` contain portable common data structures and functions. They are the portable counterpart of `pc/dat.h` and `pc/fns.h`. This is the place where to start searching for the definition of data structures used within the kernel. For example, `Chan`, is defined here. Another interesting file defining common functions is `lib.h` which defines routines from the C library used within the kernel. The kernel uses this instead of the real include file for the C library I mentioned above. It is common that kernels use part of the library used for user code as a convenience.

Processes are implemented at `proc.c`.

Files `fault.c`, `page.c`, `segment.c` and `swap.c` have to do with virtual memory handling. Their names give an idea of what they have to do with it.

Network interfaces are handled at `netif.c`. Note that tcp/ip code is not here, but at `/sys/src/9/ip` instead.

Finally, files named `sys...` contain the implementation of famous system calls built on services provided by the rest of the code.

2.2 System structures

As you proceed through next chapters, remember, the code mostly follows from the data structures. Therefore, try to imagine what the system will do with the information kept within the structures. Ask yourself what is each field for. Try to `grep` the source for declarations of the structures and see how they are used.

> **Lesson:** When you implement anything, plan for your algorithms but pay special attention to your data structures. If can do anything with a data structure, don't do it with code.

What? You didn't read `intro(1)`? You didn't read *Plan 9 From Bell Labs*? Go, read them, and don't continue before you do so.

Chapter 3

Starting up

Plan 9 starts with the bare hardware, and it must provide a bunch of services. If you did read your assignments, you should know which ones.

During this chapter, you will be reading these files:

- Files at `/sys/src/boot/pc`:

  l.s Low-level assembly routines and entry points.

  load.c main procedure for `9load`.

  dat.h data structures.

  devfloppy.c floppy device driver.

  dosboot.c Code for using FAT formatted floppies.

  conf.c configuration (`plan9.ini`).

  console.c Console I/O for the PC.

  boot.c Kernel loading and boot.

- Files at `/sys/src/9/pc`:

  l.s Low-level routines, including entry points (the main program, and interrupt handlers).

  main.c PC system initialization.

  dat.h Machine dependent data structures.

  devarch.c `arch` device driver, which has also routines to start some hardware services.

  memory.c Physical memory allocation.

  trap.c Trap and interrupt handling procedures.

  plan9l.s handler for system call trap and code to jump to user code.

  mmu.c Memory management unit code.

  i8259.c Programmable Interrupt Controller code.

  io.h I/O data structures.

pcdisk.c (generated) initialization for configured devices.

- Files at `/sys/src/9/port`:

  xalloc.c memory allocator for long lived allocated structures.

  alloc.c dynamic (kernel) memory allocation.

  qio.c Queues for I/O.

  portdat.h Portable data structures.

  proc.c Processes.

  pgrp.c Process groups and file descriptor groups.

  sysfile.c System calls for files.

  devroot.c root device.

  page.c Page allocator.

- Files at `/sys/src/9/libc`:

  pool.c memory pools (to support malloc style memory allocation).

...and several other files used as examples.

3.1 Introduction

In Plan 9, there are different kernels for terminals, CPU servers, and file servers. The CPU server is very much like the terminal, however, the file server is optimized to serve files, and not to run user programs. Indeed, the few programs you need to run at the file server are executed from the file server console with a lot of help from the kernel.

I will be considering only terminals in what follows.

The boot process starts when you press the power button on your PC. The BIOS (a program in ROM) is instructed to search for several devices to boot from. Usually, it will search for a floppy disk unit, a cdrom unit, and a hard disk. Once the boot device is located, the BIOS loads a block from the device. For hard disks, it loads the Master Boot Record (MBR), for floppies, it loads the boot sector (PBS).

Once either the MBR or the PBS get loaded, the BIOS jumps to its starting address. The BIOS is done. Both MBR and PBS contain a tiny program that proceeds the loading process. The MBR scans the partition table for active partitions and loads the PBS sector of the active partition. Thus, all in all, we end up with the PBS loaded in memory. Plan 9 supplies its own PBS program. It will load the program `9load` which will continue the job. To keep the PBS program small, `9load` is stored contiguously on disk; i.e. to load it, the PBS only needs to load a bunch of contiguous disk blocks. Keep in mind that PBS needs to fit in a sector. That is why `9load` is a different program, to be bigger than a sector. The source for `mbr`, `pbs`, and `9load` is in `sys/src/boot/pc`. If you want to know what programs are generated on that directory, look at the `mkfile` and see what are the targets.

Why not load the kernel instead of loading `9load`? Because the kernel may come from the network, and need a program that knows how to do that. Such a program would not fit in a boot sector. Another reason is that perhaps the code in `9load` has been compiled into a DOS COM file, which can be at most 64K. That file must obey the limits of DOS (if Plan 9 is being started from DOS), but it may load a real kernel bigger than 64K.

It is `9load` that loads the Plan 9 kernel and jumps to it. But `9load` does a bunch of useful things apart of loading the kernel. For instance, it parses the MBR and partition table to locate a configuration file (`plan9.ini`) and maybe a kernel to boot. That information is read in-memory and kept there, together with the information about existing partitions that `9load` got.

`9load` parses a file `plan9.ini` on a specified FAT partition. That file makes a provision to boot different kernels with different options. Again, simplicity demands, `9load` only knows how to read FAT partitions. Therefore, `plan9.ini` must be kept on a FAT partition. The standard Plan 9 disk partitions include a small `9fat` partition, guess why?

I now proceed with the source code. Relevant manual pages are `booting(8)` (bootstrapping procedures), `9load(8)` (PC bootstrap program), and `plan9.ini(8)`.

3.2 Running the loader

Where does `9load` start? Look at the `mkfile`, the first object file linked is `l.8` (for intel). The PBS simply jumps to the first instruction so let's look at `l.s`.

3.2.1 Preparing for loading

- `/sys/src/boot/pc/l.s:/origin`
  This is the entry point, running on real mode (emulating an old 8086, yes, ask Intel).

- `l.s:80,81`
  the data segment is set to be the same that the text segment. The text segment is ok because we are executing right now, but we know nothing about our data segment, yet.

- `l.s:83,113`
  set the video mode, say hi. What does it say when it says hi? Look at line `101` and lines `:755,764`. The author uses the bios to write on the screen, hence `int 0x10`. That's a procedure call into the bios code.

- `l.s:121,179`
  skip this. We are not using `b.com`.

- `l.s:181`
  Now we go into protected mode, loading a GDT and selectors for code and data segments.

- `l.s:240,251`
  Plan 9 executables assume the BSS segment is cleared. The BSS is where uninitialized global variables go. They are usually initialized to 0 by the program loader.

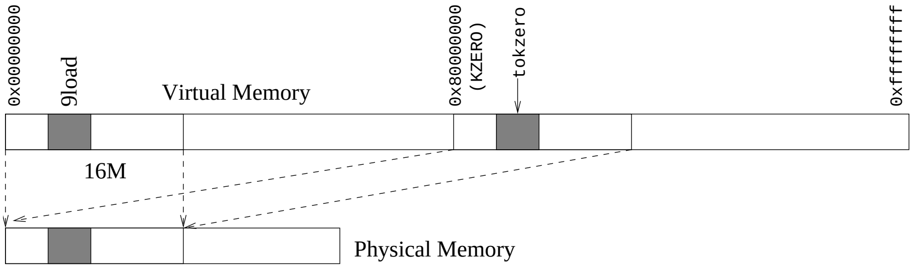


Figure 3.1: Virtual memory layout while booting.

- `l.s:257,285`
  Identity mapping for the first 16M of memory, and also a mapping at address `KZERO` (kernel address 0), which is 2G—the last 2G are conventionally used to keep the kernel code, and `9load` follows that convention. The page table is at `tpt`, which is at `0x6000`.

- `l.s:289,292`
  Now we have paging. This looks more like a reasonable machine with virtual memory, and not like an old 8086. C code can do its job now. Virtual memory looks now like shown in figure 3.1.

- `l.s:298`
  Jump to the address of routine `tokzero` starting at line `:301`. The absolute jump leaves us with the EIP pointing to the address of `tokzero` in the virtual memory mapping at `KZERO`. Note that until now, jumps were relative. This is the first absolute jump and `9load` was linked to execute at address 0x80010000. Therefore, although `9load` was executing at addresses below `KZERO` (using physical addresses—and the identity map at 0 after enabling paging), it is now executing at its proper location.

- `l.s:311`
  Just call `main` and C code will do everything else. If it ever returns, we loop forever. By looping, the author hopes to be able to read at least any interesting message on the console. Don't loop and you will get a reboot and miss any message with clues about the boot failure.

`main()` *C entry point for the loader.*

- `load.c:/^main`
  This is the entry point of `9load`, well, the C entry point—there was some assembly before. As you see, lines `:251,255` are initializing trap handling, clock, etc. More on that when we see the real kernel.

- `load.c:261`
  The `for` loop at this line is iterating through an array of known device types. The array `types` at `/sys/src/boot/pc/load.c:10` defines a set of devices where `9load` knows how to boot from. How do we know what is the type

of `types`? Hmmm, there is `/sys/src/boot/pc/dat.h` file, and there is where the author likes to put data structure declarations. At line `:143` there is a declaration for `Type` (the initial capital tells us that it is a type or a constant name). Pay attention to the set of pointers to functions, depending on the value of `type` they will point to a function or another. This is how the most useful feature of object-orientation (polymorphism) is brought to C. And it was brought to C back in the days of UNIX 6th edition!

- `load.c:264`
  Back to the loop, we call `probe` for every type (not for ether). The flags mean "we want a `plan9.ini`" and "any device will be good for us".

  That routine returns a `Medium` pointer with information about the probed medium, including the `plan9.ini` file. When the returned medium has a `plan9.ini` file (known by the `Fini` flag) we got it. Then line `:266` calls `dotini` to read and parse the just loaded `.ini` file.

`main`

`probe()` *Probe devices for media, seeking for `plan9.ini`.*

- `load.c:178,241`
  Probe iterates again through the array of known boot device types. Why? The author wants to call probe with things like `Tany`, to specify "them all". Therefore, we iterate in line `:188`. If line `:192` is reached, we are interested on this device type.

- `load.c:192`
  As initially the flag is not 0, we enter here now. This `if` is used to retrieve information found on previous runs about media for this device type. No such information by now, so the loop is doing nothing now.

- `load.c:199`
  Interesting stuff happens. The flag does not have the `Fprobe` bit set: this means the device was not probed and must be probed now. `init` must be called for the device type; remember that `init` points to a function or another depending on the `types` array entry—looks like polymorphism. If you look to the array declaration (`:10,31`), you notice a `floppyinit` as the value for floppies.

`floppyinit()` *Initialize the floppy devices.*

- `devfloppy.c:133`
  has the entry for `floppyinit`. I won't tell you how to probe for a floppy. But you can feel how I/O proceeds (e.g. `:163` is stopping the drive motor).

- `load.c:201`
  The `mask` is set with a set of bits given by `init`. The bits are used in the loop at line `:204` to scan for different media on this device type. For instance, you may have different floppies installed.

- `load.c:205,207`
  When a given media is processed, its bit in the mask is reset—on following

calls to `probe`, that media is not scanned. The idea is to link the information wanted into `Media` structures hanging from the `types` array entry. Once the information has been obtained, it is not interesting obtaining it again; hence the bit mask.

- `load.c:215`
  The call to `initdev` simple builds a string with the name for the device representing this media, e.g. "`fd0`".

- `load.c:217,233`
  Remember that the `types` array had `Fini` set as a flag? Line `:213` set the media flag to the value of the device flag. That means that the device media may contain a `plan9.ini`. now going to check if that is the case. By now, clear `Fini`.

- `load.c:219,220`
  Try to locate a partition with name `dos` or `9fat`. The routine doing the work is `getdospart`, which again points to one function or another depending on the device type. For floppies, it is `floppygetdospart` at `devfloppy.c:330`. In the case of a floppy, locating a partition is easy: if it is named `dos`, it is there. No matter the device, the partition with `plan9.ini` must be either `dos` or `9fat`—to keep `9load` simple.

  `floppygetdospart()` *Prepare to use a dos partition.*

- `devfloppy.c:330`
  It gets a filled-up `Dos` structure, which allows us to read/seek a FAT partition. How? well, `getdospart` fills a `Dos` structure with pointers to functions that know how to read and write it. For instance, `floppygetdospart` is placing pointers to `floppyread` and `floppyseek`. `Probe` is learning to load a `plan9.ini` (and a kernel) from a device step by step.

  Besides, `floppygetdospart` calls `dosinit`, which reads the first block of the floppy and initializes structures like the one for the root directory in the partition.

- `load.c:223,231`
  Ask the `dosstat` routine to locate a file named either `plan9/plan9.ini` or `plan9.ini`. Both names are kept in the `inis` array, to search for a different name we would only add it there and recompile. The array is terminated with a null entry. That is a usual convention to let routines using an array where does the array end. It works for strings, and it works for other arrays as well.

  `dosstat()` *Check (stat) a file is in the dos partition.*

- `dosboot.c:446`
  contains `dosstat`, it walks the path given and locates the file, starting from the root directory. Where is the root directory? `floppygetdospart` also called `dosinit`, which used `seek` and `read` routines to read bits from the drive and initialize the `root` member of the `Dos` structure.

- `load.c:223,231`
  When `dosstat` locates the `plan9.ini` file, its name is recorded in the `Medium` structure and a flag set to note the existence of a `plan9.ini` on this media.

`main`

- `load.c:264,266`
  Once probe returns back to main, the user is told the device, partition and `.ini` file used; then it calls `dotini` that reads the `.ini` using the functions selected by the filled up `Dos` structure. `Dotini` also parses the `.ini` file.

`main`
`dotini()` *Read the* `.ini` *file.*

- `conf.c:88 :93 :105,112`
  `dotini` calls to `dosstat` and `dosread` to read the `.ini`, and `memmoves` copy it to address `BOOTARGS` (an stat is a good way to check that the file is indeed there). The dance around `id` on lines `:105,112` is to ensure that the just copied file image starts with the line `ZORT 0`, forget it. Later, the kernel will find its `plan9.ini` at address `BOOTARGS`. It would be better to do the parsing just once here, then copy the cooked arguments for the kernel to address `BOOTARGS` and avoid the need to re-parse the whole thing again within the kernel, but the code is nice anyway, isn't it?

  Well, if you are curious and didn't forget the `ZORT 0` thing, look at `/sys/src/9/pc/main.c:64,65` and `:146,147`, not a big deal.

`main`

- `load.c:271`
  Got the configuration parameter, so the console can be initialized. `consinit`, at `console.c:16`.

`consinit()` *Initialize the console.*

- `console.c:16,38`
  Initializes an input queue `:21` and the keyboard. If the console is not `cga`, it must be a serial line. In that case it initializes an output queue and sets up the serial line. To setup a serial line is a matter of calling `uart` routines with the configured baud rate. Note how it can call `getconf` through `consinit` to get configuration parameters. Since now on, the user will see `9load` messages at the configured console.

- `load.c:278`
  As the comment says, it is doing some work for the upcoming kernel. Noticed the `Tany` flag?

- `load.c:283`
  Ask for the configured bootfile, which is simply a path as said in `plan9.ini`.

- `load.c:285,332`
  The next lines are more complex than they could because they attempt to simplify things for the user. They permit `bootfile=local!9pcdisk.gz` and related syntax. What we are interested in, is that these lines set `flag` to filter the set of known boot devices and obtain both the `file` name for the kernel, and the media (`mp`) where it resides. Then they call `boot(mp,file)` to do the job.

`main`
`boot()` *Load and boot the kernel.*

- `load.c:140`
  `boot` records at address `BOOTLINE` the full path for the kernel loaded (e.g. `fd0!9pcdisk.gz`) and calls the media dependent `boot` routine to do the real job. Remember that there was a pointer to a boot routine for each device type?

3.2.2 Loading the kernel

`boot()` *Load and boot the kernel.*

- `load.c:148`
  Why does `boot` set state to `INITKERNEL`?

- `load.c:150,156`
  Notice the `static` qualifier in `didaddconf`. The first time `boot` is called, it initializes media configuration entries. They are added to remaining entries found in `plan9.ini` for the kernel to use. For floppies, there are no extra parameters. Therefore, the test at line `:153` fails and we skip this.

`floppyboot()` *Boot procedure for floppy devices.*

- `devfloppy.c:204`
  `floppyboot` calls `dosboot` after checking that the file name looks fine. Noticed that kernel code is always paranoid? Why does the author check that when `load.c` got a fine name? Months later, the author could edit `load.c` and make a mistake, if `devfloppy.c` assumes that names are ok, it should check for that, and it is doing so.

  Well, I admit, previous checks for the file name could go away, `floppygetdospart` at `devfloppy.c:330` is checking for that itself. Nevertheless, this code exposes what is known as “defensive programming”. Checking for errors that cannot happen, and failing gracefully when they do happen. On the other hand, `9load` runs just once, before the system boots. Nobody cares if it is comparing against `dos` a couple of extra times. That is why nobody cared to optimize this code.

`floppyboot()` *Boot procedure for dos partitions.*

- `dosboot.c:488`
  stats the file (i.e., ensures that it is on disk and gets its length along with other attributes. The call on line `:494` is doing that.

- `dosboot.c:507,510`
  8K at a time, the call to `dosreadseg` calls `dosread` to read contents of the file until `bootpass` says it has read enough. The buffer just read is given to `bootpass`. Once done with reading, a new call to `bootpass` with a null buffer tries to boot the kernel. The calls to `bootpass`, which receive a non-null buffer and then gets called with null, suggest that `bootpass` is first recording some state (i.e. the kernel image) and will use it later on its final call. The state is being recored in the first parameter.

`boot`

    ...

        `bootpass()` *Load portions of the kernel and boot it.*

- `boot.c:28,166`
  Two things to notice in `bootpass`. First, the switch on `Boot.state` at line `:43` that may be `INITKERNEL`, `READEXEC`, . . . is simply implementing a finite state automata that builds a kernel image on memory. You start in state `INITKERNEL` and make state transitions as you read more kernel bytes. The second thing to notice is a common error handling trick. When error recovery at several points in a function requires mostly the same code, a label is defined past the return point of the function (line `:109`) and a `goto` to the label is used to signal the error. Done with care, this can lead to very clear code. Putting the error handling code within another procedure (to avoid the goto) would require many arguments, and it would be very dependent on the function calling it. The way to read the code is very similar to the way you read exception handling code in other languages.

  I admit that `Endofinput` is not an error, but the rationale is the same. In this case, the `goto` clearly splits the function into the part that processes the kernel read, and the part that tries to boot it. If you replace the goto with a big `then` body including the `while` below, you would need to indent more, and the `if` then-arm would get so big that the code would be less clear.

- `boot.c:28`
  The main data structure used here is `b` (Why not `bp`?). It is of type`Boot`, defined in `dat.h`, and keeps the state for the automata, the header of the executable and several pointers. The pointers are used to aid in the copy from the buffers passed to their final memory location.

- `boot.c:42,49`
  For instance, to *prepare* to get the `Exec` file header, `bp` is set pointing to the start of the target memory location, `wp` there too, `ep` pointing to the end of the target memory, and keep on calling `addbytes`. `Addbytes` advances `wp` as it gets more calls until it reaches `ep`. If the given buffer has not enough bytes `addbytes` returns non-zero, which makes `bootpass` to return `MORE`. The caller reads another chunk of 8K, calls `bootpass`, which calls `addbytes` again. The next call will continue the copy where it was left at.

  The `Exec` file header is a table of data found at the beginning of executable Plan 9 files. The structure defines a magic number, sizes of text, data, and bss segments, the size of the symbol table, the entry point and the size of a couple other tables. Everything you need to understand the bytes following the file header! Where can you find the definition? No clue? Try in `dat.h`.

As the header fits within the first 8K block, our next state is `READEXEC`.

- `boot.c:51,52`
  Get a pointer to the exec header just read and check that it has a magic number `I_MAGIC` on it. Magic numbers let you know that you got what you expected. When you compile a Plan 9 file, the exec header will contain an `I_MAGIC` so you can later check that it is indeed a binary.

- `boot.c:53,57`
  If it is a binary, our next state is `READTEXT`, to read the text segment. Before reading it, set up pointers (`bp`,`wp`,`ep`) to fill up the memory going from the entry point to the end of the text segment. In effect, the kernel entry point is its first instruction. (To know where the kernel starts executing in the source code, you only need to check its mkfile to locate the first piece of code linked into the kernel image). The call to `GLLONG` builds an address (long) from the array of 4 bytes `entry`. Forget about `PADDR` by now.

- `boot.c:62,71`
  if the check succeeds, the kernel has been compressed with `gzip`. In that case, allocate a 1M buffer, readjust our pointer to `addbytes` to it and transite to `READGZIP` state. The compressed kernel will be read and decompressed.

- `boot.c:73,75`
  The kernel format is unknown: jump to `FAILED` state. Although line `:75` suffices, it is safer to set the state to a value that will cause a panic (line `:103`) if used. More defensive programming here.

  > **Lesson:** Prepare your code for things that cannot happen. They will happen and you will save a lot of (debugging) time.

- `boot.c:78,84`
  Assume the kernel was not compressed, if the call to `addbytes` returned 0, it was all copied, otherwise return `MORE` in line `:106` and keep on adding bytes. If it all was copied, we have the kernel text segment in place, loaded at `entry` (as said in `Boot.exec`. Therefore, next state is `READDATA`.

- `boot.c:80`
  Now, it rounds the end of the text segment to the next page boundary. When the kernel (later) sets up paging, the text segment could get different page protections that the data segment. The author ensures here that a kernel page is either text or data.

- `boot.c:81,84`
  Once more, pointers to the memory being filled up are adjusted. They now tell `addbytes` to place bytes being red into the data segment for the kernel. That segment starts at the first page following the text segment. The code uses lengths found in the exec header to know how much to put in every segment.

- `boot.c:87,92`
  Got it all. The kernel needs a base stack segment (bss) too, but it does not need to be loaded from the kernel image. The BSS contains uninitialized data that should be set to all zeroes while loading. No more stuff to read in. The next state is `TRYBOOT` and it has `ENOUGH`. The caller calls back again with a null buffer. The state transition is more of defensive programming, if the callers keeps on supplying a buffer, it returns `ENOUGH` as many times as needed, until a call without buffer instructs us to boot the kernel.

- `boot.c:37,38`
  No buffer, end of input reached.

- `boot.c:112,118`
  This can happen if the kernel file on disk is truncated. Notify that and fail.

- `boot.c:121`
  Read the address pointed to by `entry` in the exec header. That is the entry point of the kernel.

- `boot.c:123`
  `warp9` will try to boot using that entry point. If it returns, something has failed. If it succeeds, the current program is both done and gone.

```
boot
   ...
      warp9()
```
Jump to the loaded kernel.

- `load.c:484,490`
  Forgetting about ethernet, `Warp9` calls `consdrain` to flush I/O on the system console, and jumps to the entry point. Well, actually it calls the entry point and if it ever returns, `bootpass` will fail and cause a panic. I/O on the console must be flushed because it could be a serial line, and characters could be sitting on the output queue. The upcoming kernel knows nothing about this early system console, and it must be terminated now. From now on, the kernel is on its own; with some help info at addresses `BOOTARGS` and `BOOTLINE`.

3.3 Booting the kernel

The relevant manual page here is `boot(8)`. What you should learn here is what structures there are, and how are basic services started.

We have the kernel loaded at... where? The `/sys/src/9/pc/mkfile` links it with the entry point `0x80100020` (see the rule for `$p$CONF`). That was the entry point found by `9load` in the kernel's `Exec` header.

We also have protected mode, paging enabled, with identity mapping for the first 16M (so that we can use a kernel virtual address safely to refer to a physical address below 16M; very convenient). It starts executing the very first instruction of the kernel (text segment). The system memory looks like that shown in figure 3.2, butnote that the kernel must still initialize some structures depicted in the figure.

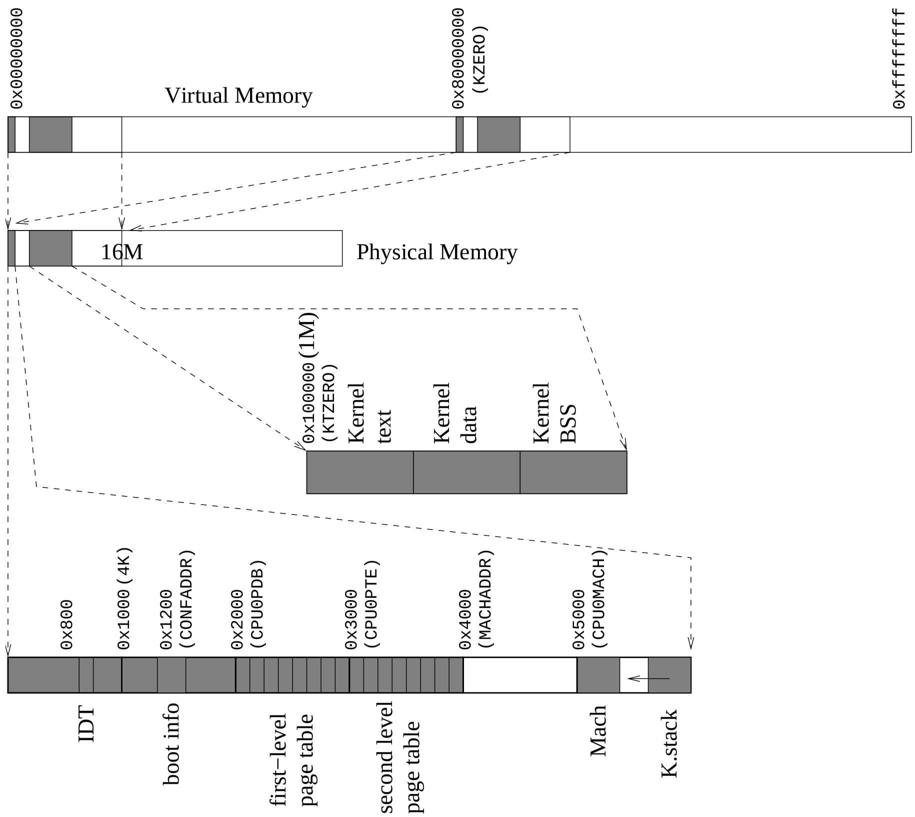


Figure 3.2: Virtual memory layout. After booting, the map of physical memory at zero will go. The first two Gbytes are used for user virtual memory.

All the work done by the `9load` program was just to get the kernel loaded on its proper place. The kernel is much bigger than `9load`, and also more complex. For example, although `9load` understood just `dos` and `9fat` partitions, the kernel knows how to handle many other devices.

What is our current program counter? Again, by looking at the `mkfile`, you see that the first file linked in the kernel is `l.s`. The authors follow the rule that the same thing is named the same way, everywhere. That helps to follow the code. A file `l.s` was the first file in `9load` too, data structures where in a file named `dat.h` too, etc.

> **Lesson:** do the same thing, the same way, everywhere. That will help when you get back to your program months after; and it will help others.

- `/sys/src/9/pc/l.s:30,33`
  This is the entry point. Clear interrupts, not yet prepared to handle them. Jump to an absolute address once more. The kernel is repeating part of the job of `9load`. Place for a future cleanup.

- `l.s:42,97`
  Again repeating the job that `9load` did, to ensure paging is enabled with a reasonable initial page table. Looks like `9load` was not the first program to load the kernel, and the kernel itself was initializing bits that `9load` handles now. The kernel only checks that it has the identity map for 4M, although `9load` did map 16M. If the current page table is at `CPU0PDB`, the kernel assumes to have a valid mapping done and goes to line `:113`. Otherwise a map for 4M is done.

  If I am not mistaken, `9load` set our page table at `0x6000`, which is not `CPU0PDB`. So, the kernel forgets about the 16M mapping and maps 4M (both at 0, and at `KZERO`). The page table is now at `CPU0PDB`, and the `Mach` structure at `CPU0MACH` (describing the boot processor) knows where the page table is. `CPU0MACH` is mapped also at `MACHADDR`. Although there is a `Mach` per processor, each one can see its `Mach` at `MACHADDR`. See figure 3.3. The reason is that the structure is used a lot; but keeping it at a fixed (virtual) address, some time can be saved. Yet another reason is that a kernel stack for use on each processor is kept in the page of its `Mach` structure. The `cr3` register in the processor (the page table) is set to `CPU0PDB` too. Now you are out of the temporary mapping done by `9load`.

- `l.s:107,111`
  The map of `KZERO` at 0, (identity mapping) is removed. The author does not want address 0 to be valid to catch null pointer dereferencing. The `or` at line `:109` is obtaining a kernel virtual address from the physical address of the `pdb`.

- `l.s:113,121`
  clearing the BSS (`9load` did that before!).

- `l.s:123`
  The machine (processor) information structure at address `MACHADDR`, set the stack there, after the `Mach` structure.

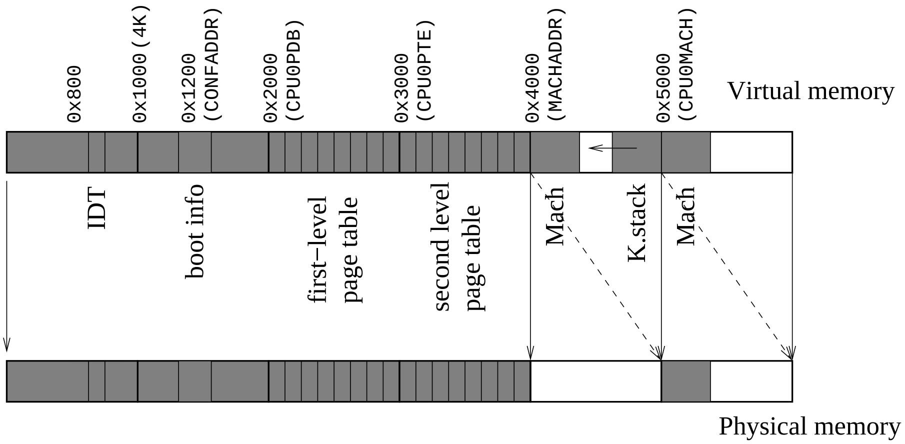


Figure 3.3: Mapping of the `Mach` structure.

- `l.s:124`
  `m` is a global pointer to the machine structure, initialize it to point to `MACHADDR`, which has a map of `CPU0MACH`.

- `l.s:125`
  it is running at processor number 0 (boot processor).

- `l.s:127`
  The machine structure resides at `MACHADDR` (`0x4000`; i.e. `0x80004000` kernel virtual). Now the stack is set to its current value (`MACHADDR`) plus the page size (`MACHSIZE`). That means that the page where the machine structure resides (low addresses) is also used for the kernel start (high addresses) from now on. The `-4` makes the stack pointer point to the last word of the page, and not to the first word of the following page.

- `l.s:133,137`
  Once we are executing in the kernel stack for this processor, `popfl` can be used to load 0 from the stack into the flags—i.e. clear them—and the author can call `main` to run the kernel C entry point. Note that if `main` ever returns, the loop at `:143,147` would make the processor halt forever. Interrupts are enabled to let the kernel attend interrupts but it would halt again later.

  Other parts of this file (l.s) have routines that are better written in assembler either because they glue the kernel with hardware facilities or because of performance.

`main()` *C entry point for the kernel. Initializes it and creates a first process.*

- `main.c:111`
  The start of it all. From now on, we will see how important kernel structures are initialized before the first process is brought to life. Most of them are defined in `dat.h` and `../port/portdat.h`.

- `main.c:113`
  Turn off the floppy motor? Yes. `9load` used timers to turn off the floppy motor after a period of idle time. Where is `9load` now?. Safety first.

- `main.c:137`
  Look at this line and skip the previous ones by now. Looks like it is redoing the job of `9load`.

In the following subsections I show how different parts of the kernel are initialized during boot. All of them are simply describing what happens in `main.c:130,165`. As we see the initialization of system services, you will learn a bit of how are them designed and what are the data structures involved. If at some point you feel like you miss where you are, go back to main and follow the call graph down to where you are. Remember that everything else shown in this chapter describes how the `...init` routines are called from main and what do they do. In the commentary, I try to preserve the order of execution as much as possible. I suggest you try to read structure definitions as they appear in the code; for all structures found, you should try to guess what is each field for.

3.4 Processors and system configuration

`main`

- `main.c:130,133`
  Plan 9 runs on multiprocessors (MP). Each processor executes both user and kernel code. When a processor is servicing a system call, it needs to know what is the current process running there. Instead of using a global variable, an array of structures (one per processor) is needed. By indexing on the array with the processor number, the kernel code executing in a particular processor can get the information about the process running on it. In Plan 9, the per-processor structure is `Mach` (see `dat.h:144,195`). As you can see, it contains a `Proc*` among other things. You also know that it also contains a pointer to a page table and that a kernel stack for use on that processor is found after `Mach`.

- `main.c:130`
  Starting to initialize the global `conf` declared below in `main.c`. It is a `Conf` structure, defined in `dat.h:67,84`, that includes the overall system information including the number of processors, maximum number of processes, installed memory, etc. By now, it only knows of the boot processor (other ones are still inactive). `nmach` must be 1, then. Later, `main.c:140`, `confinit` is called to initialize remaining fields of `conf`. We'll get back to it later.

- `main.c:131,133`
  `machp` is an array with pointers to `Mach` structures for each processor. `MACHP` is a macro that gets a reference to an array entry. It is probably defined as a macro to provide other means to reach the machine struct for a given processor. Besides, it is so simple and used so frequently that it is not probably worth to pay a procedure call just to use it. Although `m` points to the `Mach` for this

processor, `MACHP` has to be used to reach the `Mach` for other processor than the current one.

The routine sets the pointer to the `mach` structure for processor 0 (see above in `l.s`) and a pointer to the page table being used by this processor. The page table was initialized before by `l.s`. Each processor uses its own page tables, which is reasonable because each one can run a different process on a different address space. Everything else in `mach` is set to zero in `machinit`.

- `main.c:134`
  Ignore `ioinit`, it initializes IO port allocation. We'll get back to it later.

- `main.c:135,136`
  Record that cpu number 0 (bit 0) is active. It is not doing a shutdown. The `active` structure defines the status of known processors. The kernel can look at it to see whether a processor is panicing, or halting, or running or not.

`cpuidentify()` *Identifies the processor model*

- `main.c:139`
  `cpuidentify` does two things:

- `devarch.c:473,494`
  It identifies the processor model (recording that in the `mach` structure)

- `devarch.c:495`
  and starts the timer. From now on, the intel 8253 prepared to generate clock ticks (more on this later, but note that interrupts are still disabled). The timer is very important for system operation because it provides the heartbeat needed to preempt processes among other things.

- `main.c:140`
  `confinit` has the important task of initializing the idea the kernel has of available memory.

`main`
  `confinit()` *Initialize* `conf`*.*

- `main.c:369,376`
  `confinit` looks at parameters `maxmem` and `kernelpercent` from `plan9.ini`. If not found there, they are set to a null value and will be adjusted later. `maxmem` tells the kernel what is the size of installed physical memory. The kernel could guess that value as you will see, but on certain cases, the CMOS does not hold a valid value and it is necessary to force the real value using the configuration parameter.

  `kernelpercent` is the percentage of memory to be used by the kernel. Remaining memory is used for user processes. The appropriate size depends on how is the kernel to be used. More on this later.

- `main.c:378`
  The call to `meminit` initializes the physical memory allocators in `memory.c`. The

parameter is a hint about existing physical memory, but `meminit` will guess by itself if told nothing. After `meminit`, we know the number of pages actually installed. Besides, `meminit` leaves in `conf` the address and size for the first RAM memory block found (`base0` and `npage0`) and also for the largest RAM block found (`base1` and `npage1`). We'll get back to `meminit` later.

- `main.c:380,382`
  Now the total number of pages is known, and it limits the number of processes to 100 plus 5 more per MByte installed. It is not reasonable to be prepared to handle more processes than can be afforded with existing resources.

- `main.c:383,386`
  For machines not used as terminals, but as CPU servers, the number of allowed processes is increased. And in any case, this number is kept below a reasonable limit. These numbers are a guess from the author about what are reasonable values for terminals and cpu servers. Probably, they have been adjusted over time as the author gained experience with running terminals and CPU servers. Not an exact science.

- `main.c:387`
  At most 200 different images (for files) in memory handled at a time. That means that on the limit, assuming an image per file, only if there were groups of 10 processes per program running would we be able to reach the 2000 processes limit.

- `main.c:388,389`
  Establish swap limits as functions of the number of processes allowed. I will discuss swap in a following chapter.

- `main.c:391,408`
  For kernels running as CPU servers, give an enough percentage of available memory to the user—i.e. not for kernel— by adjusting the user percentage `userpcnt` and limiting the portion for the kernel. The number of images is increased an order of magnitude if enough kernel memory is available, to avoid constraining the maximum number of different processes. What is enough kernel memory? The biggest thing the kernel keeps is a big array of `Page` structs, one entry per page frame installed. They estimate that, besides the array, 16MB plus whatever is needed to keep kernel stacks for processes is a reasonable value.

- `main.c:410,424`
  For terminals, increase the user percentage of memory up to a 40% or a 60% depending on the amount of memory installed. For small machines it is given just a 40%.

- `main.c:422,423`
  If a terminal very low on memory, tell the allocator for images to take 4M as soon as possible.

- `main.c:426`
  Half of kernel pages to be used by the allocator for use at interrupt time—more on this later.

- `main.c:433,448`
  The field `maxsize` of `mainmem` is updated to reflect the actual available memory for the kernel. That is the number of bytes in kernel pages minus the size of arrays for `Page`, `Proc`, and `Image` structures.

3.5 I/O ports

We forgot `ioinit` before. Let's get back to it.

- `main.c:134`
  Starting to do resource management now.

`main`
  `Ioinit()` *Initializes I/O port allocation.*

- `devarch.c:49,61`
  `Ioinit` initializes a data structure called `iomap` that simply records what I/O ports are in use. If a driver wants to do I/O on a port, it should (note, not "must") allocate a it on the `iomap`. If allocation fails it means that somebody else is using the port. Any line in the kernel can do I/O at will to any port, but it is the responsibility of the kernel to note what ports are in use, what ports are not, and which drivers are using which ports.

  Both `ioinit` and `IOMap` are defined in `devarch.c`. `iomap` holds an array of maps that represent I/O port ranges (from `start` to `end`). Because I/O ports are a sparse resource (many different port numbers, and only a few used), it is better to record which ports are in use rather than using a bitmap to keep the allocation status of every I/O port. The data structure used by Plan 9 fits well with the resource usage because drivers tend to allocate a small set of contiguous ports. The `IOMap` defines precisely that.

3.5.1 Port allocation

Let's stay in `devarch.c` for a while to see how port allocation works.

- `devarch.c:54,57`
  Maps are linked together on a free list. It is not clear for me why are maps linked on a free list instead of scanning just the 32 entries for a free map—ports are not allocated so frequently.

`ioalloc()` *Allocates I/O ports.*

- `devarch.c:70,123`
  `ioalloc` allocates ports using the `iomap` initialized before. Line `:75` is very important. Not now, but later, multiple processors could be running `ioalloc` at the same time. `lock` ensures mutual exclusion on the `iomap`. The parameter to `lock` is the address of `iomap`. Any value would work, but every lock requires an unique value. By using the address of the structure to be locked, the author has a fine way to give an unique value. Only routines locking that structure would give that value. Following calls to `unlock` release the lock. Locks are discussed together with processes in the next chapter.

- `devarch.c:76,93`
  Negative port values can be given to request any port within a particular range (0x400–0x1000). Looks like some device is interested in any port between such range, and the routine is reused to provide that service as well.

- `devarch.c:93,104`
  This is where allocation happens when the caller specified a port. The loop starts at the first map and iterates through the next pointer. You see how used port ranges (`maps`) are linked together—unused ones are linked on the free list. Perhaps it would be more simple (and more inefficient) to iterate through the entire array of port maps. Why does the author use a pointer to the next pointer to follow the list?

- `devarch.c:97,98`
  If the end of the range is before the port address, it must be further on the list. The list is sorted.

- `devarch.c:99,100`
  If the starting address of the range lies after the allocated range there is no I/O map for that range and we reach the break for the loop. Otherwise allocation will fail—i.e. it is already allocated.

- `devarch.c:105,111`
  The first map of the free list is extracted for this new allocation.

- `devarch.c:112,116`
  The range of ports is noted, and `tag` is set. Good for debugging and to know who is holding which ports. The author ensures that the string is null terminated.

- `devarch.c:117`
  That is why the author used pointers to pointers to maps to iterate through the list instead of pointers to maps. He wants to insert the node in the allocated port list (sorted). The code already knows where to insert it because the loop was broken at line `:100`. By knowing what pointer must be adjusted to point to the newly inserted node, a new loop to find the insertion point, which would be a waste, can be avoided.

- `devarch.c:119`
  Everything is a file, and the file representing allocated ports has been updated. That is why it increments the file version number. More on that later.

- `devarch.c:121,122`
  Allocation succeeds, the map can be unlocked so other processors can gain the lock; return the starting address of the allocated region.

Port deallocation is easy too.

`iofree()` *Deallocates I/O ports.*

- `devarch.c:126,144`
  `iofree` releases a port. After gaining the lock to avoid races, the allocated map

list (`iomap.m`) is searched. If the starting address is the port being deallocated, the node is removed from the list. Again, the author is playing the pointer-to-pointer-to-node trick. If the starting address is bigger than the port, the port is not allocated and nothing has to be done: it is ok to `iofree` a free port. To deallocate a range it is only necessary to know the starting port number—compare to `malloc` and `free`.

3.5.2 Back to I/O initialization

- `main.c:138`
  `screeninit` simply prepares the console to receive characters.

  `screeninit()` *Initialize the screen.*

- `cga.c:100,106`
  Initialize the global `cgapos` with the actual cursor position read from `0x0e-0x0f`. The line `:104` is because the CGA memory holds pairs "character:attribute" for every text character on screen.

- `main.c:139`
  `cpuidentify` (which you saw before) starts the timer leading to...

```
main
  cpuidentify...
     i8253init()
```
Initialize the timer chip.

- `i8253.c:30,38`
  `i8253init` uses `ioalloc` to request the port `T0cntr`. This port is registered as allocated, and nobody else will (should!) use it. At this point, the kernel is starting to use resource allocation. I will ignore the rest of `i8253init`. But note that the clock will be sending `HZ` ticks (interrupts) per second.

3.6 Memory allocation

Before looking at `meminit` let's take a look at kernel (physical) memory allocation.

- `memory.c:31,34`
  The kernel has to keep track of which parts of memory are there, which ones are allocated, and which ones are free. The `Map` specifies a memory range starting at the physical address `addr`.

- `memory.c:36,42`
  An `RMap` (RAM map?) holds a list of maps and is the real allocation data structure used through `memory.c`. Every `Rmap` is given a name, for debugging, and to know what kind of memory it is managing. Pay attention to the `Lock`. It is used to achieve mutual exclusion between different processes doing (de)allocations. Routines `mapalloc` and `mapfree` operate on `Rmaps`, so they are used to allocate and deallocate any kind of memory.

- `memory.c:44,71`
  On the PC, there are several kinds of memory:

  – The memory up to 1M can support I/O and DMA. It is called conventional memory (from 0K to 640K) and upper memory (from 640K to 1M). For upper memory there are two maps, `rmapumb` and `rmapumbrw` because some upper memory blocks (UMB) may hold device memory (read only) and some others may be used like regular ram (RW). Some conventional memory will be placed under `mapram`.

    RO memory is placed under the allocator too. It is not for r/w, but drivers must allocate the portion they want to read to refrain other drivers from operating on devices using it—and also to know that the memory is really there!. That is the way multiple drivers whose devices get mapped on the same slot can avoid interfering with each other.

  – Memory from 1M up to 16M can do just DMA.

  – From 16M on, neither I/O nor DMA is supported.

  Different `RMaps` are used for each region. The names tell you which kind of memory you are referring to. The `rmapram` is used for memory that can be read and written, and is not used by device I/O; i.e. it is regular RAM. For memory below 16M that can be used for devices, `rmapumb` and `rmapumbrw` are used. The first one is used for ROMs showing up as UMBs; and the second for device memory that can also be written.

  Another interesting thing is the couple of `upa` allocators. That is memory that does not exist. The kernel has the addresses, but there is no memory there. The memory will appear when a particular device supplies it. More on this later.

`mapalloc ()` *Allocate memory from a `RMap`.*

- `memory.c:150,205`
  `mapalloc` is fairly easy to understand. If the `addr` is zero, it understands "I don't mind where the memory is allocated". In general, you don't mind. But sometimes a particular driver needs a particular region of memory to interact with the device (e.g. a video frame buffer). Note also the usage of the parameter `align`, which can be used to allocate aligned memory (For instance, a page frame could be allocated by using the page size as `align`). By the way, is it first fit?, best fit? worst fit? Hint: they kept it simple.

  In lines `:196,198` they release the (prefix) unused portion of the allocated map. The routine is more simple than it could be because it behaves like allocating from the starting of the map to the end of the actually allocated memory, and then it uses `mapfree` to release the memory going from the starting of the map to the start of the allocated memory. They could call `mapfree` another time to do the same with the trailing unallocated portion of the map, but looks more simple to adjust the existing map to represent that trailing portion. The data structure is unlocked before the (possible) call to `mapfree`, which acquires the lock on its own.

- `memory.c:106,148`
  After searching for the position in the list where the memory should be placed, three things can happen:

  1. The previous map ends right before the memory being released: opportunity to recombine and avoid fragmentation. Add the memory to the end of the previous map in line `:118`, and move following fragments to the left in the map list. It stops on a map with length 0. `map` structures are neither allocated nor deallocated, their length will be zero if they are not useful (cf. lines `:188,189`).
  2. The released memory ends right before the next map. Recombine the memory into the next map.
  3. No luck; must insert the map by moving next ones to the right in the list. When there is no room for more maps the last one is dropped: the list is overflowing to the right. If you see the "loosing" message, you should increase the number of entries in the map and recompile the kernel: your system may need more maps of that kind.

  Given the small number of maps in rmaps, won't the kernel run out of maps early? Not so easily. Maps are used either by devices (and they allocate only a few ones) or to allocate dynamic kernel memory. What the author is doing is to use these low level "ram maps" to keep track of what memory banks are installed on the system, and what kind of memory they keep. For dynamic kernel memory, another allocator of smaller grain is built on top of the memory supplied by the rmap. So, you won't run out of maps easily. The source of the dynamic kernel memory allocator is at `../port/xalloc.c`.

Memory initialization

Now, let's get back to `meminit`. It must fill up the RAM maps and set page translations for existing memory (only 4M mapped by now).

```
main
    confinit
        meminit()
```
Fills up `Rmaps` *and builds an initial page table.*

- `memory.c:421,428`
  First, entries in the page table used by this processor (`m->pdb`) are updated for the upper memory. The range for video memory is set write-through (you don't want the cache to retain just written pixel values). The range used by ROMs and devices you want to be uncached, to interact with the devices directly. The routine `mmuwalk` returns a page table entry (`pte`) given the virtual address. Remember that below 4M physical addresses were mapped one-to-one at `KZERO`? `KADDR` remembers: The author knows it cannot use physical addresses, because the kernel is also running with paging enabled (i.e. using virtual addresses). A physical address is converted to a kernel virtual address by using `KADDR` (i.e. by adding `KZERO`). That is because for kernel usage, physical memory is mapped at virtual addresses starting at `KZERO`. Although right now not all physical memory is mapped at `KZERO`, that is going to change soon.

The page table being updated it the one for the boot processor, it will be used later as a template to setup new page tables.

- `memory.c:429`
  Every protection change on the page table requires a TLB flush. Otherwise, until the next context switch, entries within the TLB would retain the old protection flags. You will see `mmuflushtlb` in the virtual memory chapter.

- `memory.c:431,432`
  These two routines scan available memory and fill up `Rmaps` accordingly. The author probably wrote two routines because unlike regular RAM, upper memory must take into account video memory and the like. These routines update the page table for the kernel to make `KZERO` be the starting of a map for all physical memory. Bby now, `KADDR` can be used only for physical addresses within the map at `KZERO`, which are just 4M. Starting to fix that now.

- `memory.c:440,452`
  `conf` is updated to keep the address (base) and number of pages (npage) of the first RAM map; it is also updated to keep the address and number of pages of the biggest map. Hopefully, on the PC, the first map would be conventional memory, and the second one will contain all extended memory.

But, how do `umbscan` and `ramscan` work?

Filling up allocators

```
main...
    meminit
        umbscan()
```
Scans for UMB blocks.

- `memory.c:208,253`
  `umbscan` starts looking past the end of video memory up to the ROM signature at `0xc0000`. It does not go up to `0xf0000` because a two-byte check at `0xc0000` can tell us if there is a ROM mapped there by the hardware or not.

- `memory.c:228,229`
  At each pass, it writes the first and last byte of every 2K chunk with `0xcc`.

- `memory.c:230,233`
  If reading back those bytes does not yield the just written value, it is not real RAM. It must be a ROM then. So rewrite `p[0]` and `p[1]` just to be sure that if they were on registers (as dictated by the compiler), their values are in sync with the real value in memory. Not sure why they write `p[2]`, but could be for a similar reason. `p[2]` seems to contain the number of 512 byte blocks at the ROM scanned.

- `memory.c:234`
  If the two bytes starting the 2K block match the values just written (the signature of a ROM), skip the number of 512 byte blocks recorded by the ROM in the third byte. This portion is not kept in the allocator.

- `memory.c:238,239`
  If the two bytes are `0xff`, make the memory available for read-only allocation by calling `mapfree` (Note the `rmapumb` map and not the `rmapumbrw` map). Remember that it is not regular RAM because the read value was not what the routine wrote. But blocks marked with `0xff` seem to hold device RO memory for us to read.

- `memory.c:241,242`
  It was regular ram, so make it available for allocation. Adjacent maps will be coalesced.

- `memory.c:246,252`
  Finally, looks like if the first two bytes at `0xe0000` are `0xff`, not signed by a ROM, and not regular RAM, indicate that there is device memory (64K) for us to read. Place the memory in `rmapumb`.

```
main...
   meminit
      ramscan()
```
Scans for regular RAM blocks and updates the page table.

- `memory.c:256`
  `umbscan` was easy, tricky because of PC messy memory management for IO devices, but easy. Now, `ramscan` has the important task of updating the the kernel page table to reflect the installed ram. Besides, it fills the ram map as memory gets scanned.

- `memory.c:274,276`
  The routine leaves untouched the range from 0 up to `0x5fff`. If you look at `mem.h:33,39` you will see some stuff, going from the interrupt descriptor table, and information from `9load` up to the `Mach` structure for this processor at `0x5000` (see figure 3.2). Therefore, start by putting into the `rmapram` allocator memory going after the `Mach` structure up to the end of conventional memory (640K). To determine the end, it reads BIOS information at `0x400`—again, a very low address better left untouched.

- `memory.c:278,280`
  From 640 up to 1M we had the UMBs scanned before, and from 1M on we had the kernel loaded. (Looking that the `mkfile` you see how the image is linked to start at kernel virtual address `0x80100020` which leads to `0x00100020` physical address). So, only memory from the end of the kernel upwards may be used now. The author gives to the allocator the memory starting at the end of the kernel. The linker places the symbol `end` at the very end of the kernel image, past the data and bss segments (note that our stack "segment" is actually a bunch of bytes after our `Mach`). How much memory do you have? By now, the author places in the allocator at least `MemMinMB` MBytes. That has to be a reasonable low value. By allocating at least that, the author can use the allocator in the following code that scans for more available memory.

- `memory.c:290,301`
  If no hint was given about how much memory is installed, it makes a guess

by reading from CMOS the configured value. In any case, pretend to have at least 24 MBytes. The PC is not so good at letting the system know how much memory is installed, there are many variants out there. Most of the complexity of `umbscan` and `ramscan` has to do with the idiosyncratic nature of the PC.

- `memory.c:309,314`
  Starting at the page after `end+MemMaxMB`, it is going to check one MByte at a time if the memory is there or not. The trick is to write a silly value (line `:312`) at the first word of the Mbyte being tested, and see if we can read it back. If the write did work, there is memory. The author is saving the value actually stored at address `KZERO`, can you guess why?

- `memory.c:320,321`
  While it scans for memory, a page table reflecting the actual memory installed is built. So, the physical address scanned is converted to a kernel virtual address, and used to index into the first level page table. `table` points to the entry in the first level page table (page directory, or PD) corresponding to the virtual address scanned. Now going to update the "image" of physical memory mapped at address `KZERO` to reflect the memory actually installed.

- `memory.c:322,328`
  If the entry is null, there is no secondary page table, and it must be allocated. In line `:326` the entry in the PD is updated to be valid and point to the secondary page table (PT) just allocated. By zeroing it, the routine invalidates all its entries—valid bit is zero. Line `:327` is reseting a counter that we will discuss below. Saw how it can allocate memory while filling up the allocator? It was convenient to place a few Mbytes there at line `:280`.

- `memory.c:329,330`
  Now getting a pointer to the secondary page table entry for the virtual address scanned. The macro `PPN` gives the physical page number (a physical address, actually).

  When the author gets a pointer he pretends to use, it must be a kernel virtual address (i.e. bigger than `KZERO`). However, page tables keep physical addresses. Do you get the picture?

- `memory.c:332,332`
  Establish the mapping by setting the physical page address, the valid bit, and write permission. The flush must be done too, remember why?

  Since the `PTEUSER` bit is not set in the entry, only the kernel (ring 0) can access this virtual memory page.

- `memory.c:344,372`
  Here is the actual scan for memory. The commentary is a "must read". `mapfree` is used to place memory under the allocator; its parameter is one of `rmapumb` (UMB memory), `rmapram` (Real memory for use), and `rmapupa` (Memory that seems not to be there). For regular RAM, enable write permission for the MMU; for UMBs (i.e. device memory), set it uncached to get straight to the device memory; and for "phantom" RAM, clear the map.

The `*pte++=`... is used to update the mapping and advance the pointer to the next page table entry. Each of the three `if` arms map a whole Mbyte at a time once the check for the first word of the Mbyte tells us the kind of memory there. To know why one MByte at a time, ask yourself whether you can install on your PC less than a MByte of extra memory. Another thing to note is that the author is counting in `nvalid` how many pages are available. The counters are reset at the beginning of a 4MByte block, i.e. at the beginning of a page mapped with the first entry of a secondary page table.

- `memory.c:383,393`
  And here is why. On the PC, a first level page table entry can be used to map a whole bunch of 4MBytes (called a "super-page"), without using any second level page table. If the page address starts at a 4Mbyte boundary, and the pages on the 4Mbyte block just scanned are of one kind, the entry in the first level page table (`*table`) can be used to map the 4Mbyte block. `nvalid` knows how many pages there are of each kind.

- `memory.c:392`
  When the "super-page" map is not used, the pointer to the secondary page table at `map` is cleared. Next time it is checked at line `:323`, a new (secondary) page table is allocated for the next 4Mbyte chunk. If a super-page map was used, the pointer is not released and the memory used by the old secondary page table (now unused) will be reused for the next non-existing but needed second level page table.

- `memory.c:398,399`
  Perhaps the routine used a "super-page" and saved an allocated second-level page table that now is unnecessary.

- `memory.c:340,401`
  And perhaps `maxmem` is not page-aligned. Place the remaining part of `maxmem` within the not-existing memory allocator.

- `memory.c:402,403`
  In any case, ensure that from `maxmem` to the end of the memory range the "memory" is registered as not backed up. Imagine that later someone plugs in a PCMCIA memory card, memory will be moved (allocated) from `rmapupa` to (deallocated) `xrmapupa`. Now the kernel can ask for memory at `xrmapupa` and use it.

- `memory.c:406`
  Finally, restore the word at `KZERO` messed up previously for memory checking purposes. Kernel allocators are filled up reflecting the actual memory installed in the system, and the mapping starting at `KZERO` has entries appropriate for the whole physical memory.

Dynamic kernel memory

As we saw, the RAM map allocator is not enough to provide dynamic kernel memory. It was good at registering (un)allocated contiguous portions of memory areas in the system, but it is good just for big chunks of memory. Also, it is not to allocate and free repeated times a given portion of memory because it would not tolerate fragmentation—remember that it may even drop the last fragment?

The map is used during boot to allocate UMB areas for devices, and to collect available RAM for dynamic memory allocation. It is `xalloc` that collects that RAM for later use by the dynamic memory allocator.

- `main.c:142`
  the call to `xinit` initializes the allocator.

`main`
  `xinit()` *Initializes `xalloc` with blocks from the `Rmaps` and tells `palloc`.*

- `/sys/src/9/port/xalloc.c:45,84`
  It initializes a list of "holes" (i.e. memory to allocate) given the information in `conf`. It uses the first and the biggest chunk of RAM, as recorded in `npage0/base0` and `npage1/base1`. Perhaps it would have been better to turn `Rmaps` into the interface between the machine dependent and the portable part; the PC part could fill it up, and `xalloc` could collect those RAM maps found in the appropriate `rmap`. Probably the author's reason not to do so is that some other architecture may not need resource maps at all and its implementation (`conf` memory banks) is admittedly more simple.

  Although the collected memory should be enough, `xalloc` could run out of memory, yet have more memory available in `rmaps`—despite the recombination of fragments done by `rmapfree` will make its best to end up with a big map holding most of the installed memory. In any case, you better avoid dynamic memory if you can allocate an array of structures and keep on using it. Memory fragmentation is not to be underestimated.

- `xalloc.c:57,66`
  Pages (page frames) are removed from the bank 1 in `conf` and added with `xhole` to the allocator. `palloc.p1` is initialized with the starting address for the pages removed; this is not relevant for us now, but `palloc` is the source for page (frame) allocation in the system, and fields `p0`, `np0`, `p1` and `np1` are used for that. As it happens in `conf`, the author maintains information about just two memory banks.

  At most `conf.upages` are placed in `xalloc`. Those pages are to be used for user stuff.

- `xalloc.c:68,75`
  Pages are removed from the bank 0 in `conf` and added with `xhole` to the allocator. `palloc.p0` is initialized too.

- `xalloc.c:77,78`
  The number of pages placed under allocation is recorded in `palloc`'s `np0` and `np1`.

- `xalloc.c:79,82`
  Until now, `conf` recorded physical addresses for banks 0 and 1 (`xhole` receives

physical addresses, as `rmaps` do). But from now on, `conf` records kernel virtual addresses for both banks. One point here is that from now on `npage0`/`npage1` do not keep the number of pages in each bank, but the uppermost bound for each back; perhaps this is a bit confusing.

Try to understand yourself how other `xalloc` routines work. Take a look first to the data structures near the top of the file. While reading them, note how `ilock` is used to prevent race conditions—you will learn why in the process chapter. I suggest you start by reading `xalloc` (`xallocz`, actually), then`xfree`, finally the other ones.

There are 128 (i.e. `Nhole`) memory pools. Are they enough to use `xalloc` as a generic allocator? No. `xalloc` should be used just for big, long-lived, kernel data structures.

- `../../libc/port/pool.c`
  For actual dynamic memory the kernel uses pools. `pool.c` implements generic memory pools where memory can be allocated and deallocated. By using different pools for different purposes, fragmentation can be fought. Pools are like "arenas" in some programming languages. In fact, `pool.c` is in `libc/` and is used by user programs as the C library dynamic memory provider. Pools are appropriate for allocation of even small and short-lived data structures. Fragmentation would be contained within the pool.

- `/sys/src/9/port/alloc.c`
  As the pool interface is generic, hence more complex than it ought to be, a regular `malloc` interface is built on top of the pool interface, in `alloc.c`. `malloc` is very much like `poolalloc`, but allocates memory from a 4MBytes pool. There are a few differences regarding the C library `malloc` interface.

- `alloc.c:21,36`
  A pool is declared for `malloc`. Note the generic programming again. Routines are placed into the pool to allocate more memory for the pool, merge, lock, unlock, print, and panic on the pool. `xalloc` is used as the allocation routine. You already saw a bit of generic programming before, when the routines to process `plan9.ini` worked independently of how to read and seek on particular devices. The trick was to use pointers to functions and stick to a well defined interface. The code called read and seek using the interface (the function prototype) without knowing what was the actual function used.

  As pools are generic, they use routines noted in `lock` and `unlock` fields to allow concurrent usage of the pool. For malloc, the routines end up using `ilock` on a `Private.lk` field. This contortion is needed because it is malloc who knows how to lock things in the kernel, yet pools must be able to acquire locks.

`smalloc()` *Allocates zeroed dynamic memory (must succeed).*

- `alloc.c:172,177`
  A `smalloc` routine is included to request dynamic kernel memory when the allocation must succeed. When there is not enough memory, it makes the caller sleep for a while and tries again; and so on until it gets the memory. There are several places in the kernel where the author prefers to wait for memory instead

of reporting a "not enough memory" error; `smalloc` is used there instead of `malloc`. `smalloc` zeroes the memory allocated.

`malloc()` *Allocates zeroed dynamic memory.*

- `alloc.c:186,199`
  `malloc` is like `malloc`, but it zeroes the memory allocated. That is both for security issues and to be sure that anything allocated there gets a reasonable initial value: nil pointers, zero counters, etc. Shouldn't `malloc` just call `mallocz`?

  In all these routines, `setmalloctag` is used to record the PC that called the allocation routine. That is to find out who is guilty for bad usage of memory allocation routines; and also to know who is using a given portion of memory. All in all, for debug purposes.

  Apart from these details, the code should be easy to understand. While reading the file, ignore the `pimagmem` pool, used for program Images but not related to the malloc interface. It seems to be declared in `alloc.c` to reuse the locking and debugging routines that operate on pools near the beginning of the file. By making them receive a `Pool` and not use directly the `mainmem` one, they can be applied to `pimagmem` too. Shouldn't these generic routines be moved to `pool.c`?

So, regular dynamic kernel memory (i.e. `malloc`ed one) comes from `malloc` (or `smalloc`, or ...) in `alloc.c`, which in turn uses a pool (initially 4MByte) supplied by `pool.c`, confining fragmentation to be inside the pool. Several pools are used for different things (`malloc` and `Images`, by now). Pools are fed from RAM maps corresponding to installed memory. The lowest level is necessary to discriminate RAM of different kinds, the next one to reduce fragmentation, and the upper one as a convenient interface.

3.7 Architecture initialization

- `main.c:141`
  `archinit` sets up a generic interface to operate on this particular architecture model. Yes, it is a PC, but there are very different kinds of PC.

- `dat.h:216,229`
  This is the interface to highly architecture dependent routines, they vary from one PC model to another or they must be performed only at particular PC models—this is very useful to make the code work for both multiprocessor PCs and regular PCs.

`main`

`archinit()` *Initializes architecture specific procedures.*

- `devarch.c:535,542`
  The way to select concrete implementations is to scan a table of known architecture models and set the global `arch` pointing to the right entry. A generic entry is used if the exact model is unknown.

- `devarch.c:544,556`
  Conventions are used to make more simple the table of `knownarchs`. If any routine is not specified by the selected entry in `knownarchs`, it is defined as the generic routine. Looks like, in Object-Oriented words, `knownarchs` is *redefining* routines for concrete archs and *inheriting* everything else from the generic one. Simple yet effective.

- `devarch.c:560,563`
  Pentiums and above have a Time Stamp Counter (tsc) builtin that counts the number of cycles gone in the processor. It is better to perform time measures than the external programmable clock. If you know the Hertz the machine is running at, you know the exact time since the last time you reset the tsc counter. Forget everything else in `archinit` by now.

3.7.1 Traps and interrupts

Traps are important because system calls, page faults, and other traps (exceptions) are used to request some service from the kernel (perform a system call, repair a page fault, etc.). The intel has several protection rings. Plan 9 uses 0 for kernel and 3 for users. Each ring has its own stack. The stack used for ring 0 is the kernel stack. When the hardware detects a trap, it saves the processor context in the kernel stack. After that, what happens depends on the entry for the given trap number in a table called IDT (interrupt descriptor table). That table contains “pointers” to routines that handle the trap. As the Intel has segmentation, each entry contains a small number to describe on which segment the routine resides. The small number is used as an index into a Global Descriptor Table (or GDT) that contains descriptors for segments in the system (base address, length, protection). The whole picture was seen at figure 1.2. You already knew this, right?

Remember that code segments determine the protection ring you are running at. There are different text and data segments (as well as other extra segments courtesy of Intel) for users (ring 3) and kernel (ring 0). Other than protection, *hardware* segments are used almost for nothing else in Plan 9. The paging hardware is all the kernel needs.

By the way, you know which one is your current kernel stack, but beware that it would be a different one as soon as you get real processes running and such processes issue system calls. Each process has its own kernel stack used by the kernel to service its traps.

I won’t say more about how the Intel hardware deals with traps and interrupt as I feel this is enough to understand the code.

- `main.c:143`
  `trapinit` is called to initialize trap handling.

`main`
    `trapinit()` *Initializes interrupt and trap vectors.*

- `trap.c:142,163`
  `trapinit` fills up the IDT with entries for the 256 trap numbers. Each entry

in the IDT has several fields. Fields `d0` and `d1` hold the address for the handling routine. Plan 9 keeps in `vectortable` the routines handling traps and interrupts. Lines `:145`, `:160`, and `:161` store the routine address (`vaddr`) using the two fields `d0` and `d1`. Ask Intel why you must use two fields to store one address. The `KESEL` at line `:161` is specifying that the routine address refers to the kernel executable code segment; i.e. the processor will jump to ring 0 and execute the routine in kernel mode. See figure 1.2.

- `trap.c:145`
  For all traps, set the "present" bit in the entry (`SEGP`). That tells the hardware that the entry is valid.

- `trap.c:148,154`
  Why not fold these two branches? For breakpoints and system calls the entry is set up as an "interrupt gate" at privilege level 3, that is, user code is allowed to issue an `int` instruction to perform a system call or to notify a breakpoint.

- `trap.c:156,159`
  For any other kind of trap, the privilege level is set to 0 (kernel). That means that those traps should not be "called". Well, the page fault trap and others are among them. And they should not be *called*. It is just that the hardware can generate them, but users cannot use an `int` instruction to request such traps. If users try to do so, they will get a protection fault—which is yet another trap generated by the hardware.

- `trap.c:162`
  Use the next entry in the vector table for the next trap. The pointer is incremented in 6 characters each time, not in 4. Why?

- `l.s:549,805`
  In `l.s` you see that the vector table does not contain pointers to handlers, but instead, it contains binary code to call the handler (the byte after the call is a parameter specifying the trap number). The code calls `strayintr` (or `strayintrx`) in `l.s`. These are the interrupt handling procedures pointed to by IDT entries. By using the table, `trapinit` can forget about which traps get an error code pushed on the stack, and which ones do not get it. Depending on that fact, the author calls `strayintrx` or `strayintr` to ensure the kernel stack has always the same layout after a trap. The latter pushes the "error code" (the interrupt number, actually) by software, as the hardware did not do so.

  Should the intel hardware push always the error code, `vectortable` could go away and entries in the IDT point just to `strayintr` or to `syscallintr`.

- `l.s:614`
  For system calls, the `vectortable` does not call `strayintr` (common trap handling), but `syscallintr` instead. Having a common trap handling piece of code simplifies things (it avoids duplicated code), but it can make you run slower. For system calls, the call path continues at `syscallintr`, in `plan9l.s`—it does only the strictly necessary to prepare for calling `syscall` and proceed with the system call. More on this later, let's get back to regular traps.

`strayintr()` *Interrupt handler (no error code pushed by the hardware).*

- `l.s:514`
  `strayintr` simply provides common code to jump into `intrcommon`. It pushes the trap number so that all traps have an error code pushed in the stack together with the saved processor state.

`intrcommon()` *Entry point for interrupt/trap handling.*

- `l.s:521`
  Once the stack looks the same for all traps, `intrcommon` saves the data segment (which may be the user data segment) and loads the kernel data segment (Remember that the text segment is already ok, because the hardware did set it up as described in the IDT.) Afterwards, data memory references refer to the kernel data segment. This can be done because after the trap, the processor is running at ring 0; the user cannot load segment registers because that are privileged instructions.

- `l.s:528,534`
  Now, after other segment descriptors are saved (user's) and loaded (kernel's), and after the whole set of registers is pushed in the stack,`intrcommon` prepares for calling `trap` to do the trap processing. The stack at line `:535` has the saved processor status (made by the hardware) together with the registers just saved. If you look at `/386/include/ureg.h` you will see how:

  - `l.s:534`
    General registers (top of stack) were last pushed.
  - `l.s:532,533`
    Some extra user segment registers (fs and gs) were pushed before.
  - `l.s:524 and :528`
    Another user extra segment (es) and the user data segment (ds) were pushed before.

    And before that, as briefly pointed out before, the trap number and error code where pushed, either by the hardware or by `strayintr`.

    Finally, even before that, the hardware pushed the processor context that was going to be clobbered when the hardware calls the trap service procedure.

  The kernel has now in the kernel stack an`Ureg` for the saved context.

- `l.s:536,537`
  By pushing the stack pointer, the author sets up the `Ureg*` argument of `trap`, and finally calls `trap`.

  The just saved user register set will be used when returning from the trap to restore the user processor context. By restoring it, you jump back to user code. If any register is modified in the `Ureg`, it will be modified the next time the user process runs, after returning from the trap.

`forkret()` *Returns from interrupt/trap handler.*

- `l.s:539,547`
  Although a routine on its own, `forkret` is the code executed if `trap` returns. If you look at it, it restores the processor context from the `Ureg` pushed by the hardware and `intrcommon`. The `IRETL` instruction is a very interesting one because it reloads the processor with the context saved by the hardware on the trap. This means that it also restores the code segment and the stack segment and places the processor back in protection ring 3 (i.e. userland).

By now we skip trap handling. We will get back to it when discussing processes. But we had a pending discussion of `syscallintr`.

`syscallintr()` *Entry point for system call trap handling*

- `plan9l.s:31,52`
  This is the call and return path from user to kernel and vice-versa during system calls. After the `vectortable` dispatches to `syscallintr` there is lean (i.e. fast) code to get the `Ureg` on the stack; after the call to `syscall` there is again lean code until the `IRETL`. Compare this call path with the one the processor would follow by going from `strayintr` down to `trap`, and then switching on the trap number and checking some stuff, calling `syscall` and back. System calls are discussed later. The code is in `trap.c:471,539` though; and it dispatches using a system call table found at `../port/systab.c`.

The point is that after `main.c:143`, the kernel is prepared to service system calls as well as other exceptions. After reading all this code, you now know what that means.

3.7.2 Virtual Memory

We skip `printinit` by now, to continue with low-level glue initialization.

- `main.c:148`
  Interesting things begin to happen. `mmuinit` will initialize the MMU data structures. Yes, the kernel already has a page table, but it would not even be able to do a context switch. Let's see how this thing works.

`main`
  `mmuinit()` *Initializes the MMU.*

- `mmu.c:48`
  The intel is quite bizarre at supporting processes for systems software. It tries to do it all, and most operating systems have to do some contortions to use what it provides whenever they prefer to implement a different thing. This line is allocating a "Task state segment". That is the data structure used by Intel to describe a Task. It is needed because the processor uses that "segment" to switch to a different protection ring on a trap. Remember that I said that the hardware saves the processor context on the kernel stack when a trap happens? How does the hardware know what is the kernel stack? It might not be your current stack when you have a process running.

At any time, the "Task Register" is loaded with a descriptor into a task state segment (TSS). It is a memory segment as any other, but it describes the current task for the hardware. The selector loaded into the task register selects a descriptor from the GDT (Global descriptor table) that points to a TSS.

- `dat.h:109,136`
  The TSS contains among other interesting things, `esp0`, the stack pointer to be used at ring 0. When a trap places the processor into kernel mode, the hardware obtains the `esp0` in the TSS loaded at the task register, and uses that stack to save the faulting context. If the processor was already in kernel mode, the context is saved in the current stack. Remaining fields of the TSS contain the supposedly last saved context for the task. On Intels, you can use a call or a jump instruction into a TSS to switch from one task to another, and the hardware will save the previous task context, and load the next one. This is so slow that almost nobody uses the TSS to implement the context switch for OS processes. In fact, in `mmu.c:20` you see how there is just one TSS descriptor for all processes, which means that the TSS is used just to make the hardware work.

- `mmu.c:51`
  The GDT used until now, just for booting, is copied into the machine structure for the current processor. The kernel is getting out of the initial (and weird) data structures used just to boot. The current flow of control is on its way to become a regular process on a regular processor.

- `mmu.c:53,34`
  The descriptor in the GDT for the TSS is updated to point to the just allocated TSS. It will run at ring 0.

- `mmu.c:56,60`
  Concoct a descriptor for the GDT and load it into the GDT register. Now using the GDT for this processor. The kernel is almost prepared to officially switch to the new TSS.

- `mmu.c:62,66`
  Remember that the IDT was initialized at address `IDTADDR`? Now the kernel loads the IDT register. Until now the kernel were not really prepared to service traps nor interrupts—I lied. But now the hardware knows that the IDT has some new stuff in, because the register was reloaded. Did you notice that interrupts are still disabled?

- `mmu.c:68,74`
  Protections for the kernel text pages are changed not to be writable. This could be done before while the page table for the kernel was initialized during memory scan, but that's not a big deal.

- `mmu.c:76`
  `taskswitch` loads in the TSS of the current processor the given stack pointer for all protection rings (and also sets the stack segment as the kernel data segment there). It also loads in the TSS `cr3` register the given page directory pointer.

`cr3` is the pointer to the page table. Besides noting it within TSS, `taskswitch` loads the `pdb` into the processor `cr3` register. Note the "kernel stack" passed to `taskswitch`. It is the address of the `Mach` processor plus the page size. The kernel stack resides after the machine data structure. Not a big kernel stack. Remember that such stack is given to `taskswitch` so that it could initialize `esp0` to point to the kernel stack for the current task.

- `mmu.c:77`
  Finally, the task register is loaded with the selector for the TSS just created, and the kernel becomes an official task for Intel. This just means that when we get up to user level, the processor will use the right stack to service traps and interrupts: the kernel stack specified in the TSS.

3.7.3 Traps and interrupts (continued)

- `main.c:149,150`
  If the `arch` structure filled up by `archinit` has a routine to initialize interrupt handling code, it is called now. If no such routine exists, it is assumed that interrupts are already initialized.

```
main
   i8259init()
```
Initializes the PICs.

- `devarch.c:377`
  For the "generic" PC, `i8259init` is used as `intrinit`.

- `i8259.c:33,102`
  This file contains code for the i8259 programmable interrupt controllers (PICs). In the PC, there are two ones routing 8 interrupts each. The two i8259s are cascaded to dispatch up to 15 interrupts to the processor (one of the 16 ones is used to cascade the chips). Even though the programmable timer was initialized time ago, no timer interrupt ever reached the processor because the intermediate i8259s were not initialized. See figure 3.4.

- `i8259.c:37,38`
  Allocate control ports for both chips.

- `i8259.c:39,102`
  Comments make the code self explanatory. In any case, by the end of the routine both chips route interrupts in their way to the processor; The PIC is supposed to dispatch its 16 interrupts starting at `VectorPIC`. The hardware uses the interrupt vector offset to index into the IDT. Therefore, interrupts numbered `VectorPIC` through `VectorPIC+15` correspond to PIC dispatched interrupts.

- `i8259.c:29`
  The mask (`i8259mask`) programmed on the i8259s and the cleared interrupt enable flag in the processor status word are avoiding interrupts from happening. But since the kernel already has a working TSS, IDT, and handlers for the IDT entries, it is mostly ready to service interrupts.

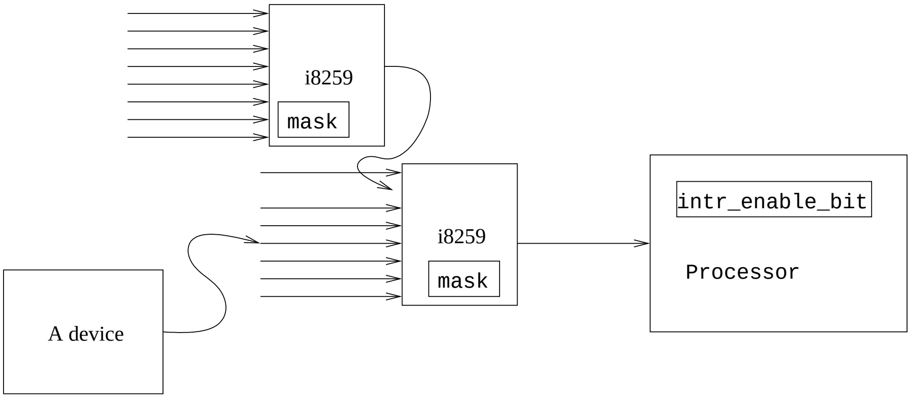


Figure 3.4: Interrupts arrive from the device though the PICs. The PIC may mask each of the interrupt lines. The processor must have interrupt enabled in `flags` to notice interrupts.

- `main.c:151`
  The PICs just initialized are used by `ns16552install`, which shouldn't be called here, but by `chandevreset` instead. My guess is that for `kbdinit` and to allow a serial line based console, the serial line (i.e. UART) initialization is being done here, or maybe this is a fossil if any time back in the past devices were initialized right from `main`. Although the code is still clean and easy to follow, it could be an alternative to place most regular initialization routines under the control of `chandevreset`, and let it resolve dependencies. But that could also be a recipe for disaster if the (generated) `chandevreset` code would not honor dependencies. Forget this brief guess if you didn't understand, it is not relevant.

```
main
   ns16552install()
```
Sets up the uarts.

- `ns16552.h:82,111`
  `ns16552install` allocates ports for the serial lines (two ones, `eia0` and `eia1`). Then `ns16552setup` initializes the two UARTS.

- `ns16552.h:99 and :103`
  `intrenable` is called to request that `IrqUART0` and `IrqUART1` interrupts be enabled (not masked by the PICs) and handled by the `ns16552intrx` routine. We'll get to `intrenable` a bit later.

- `ns16552.h:106,110`
  If `plan9.ini` said to use a serial console, setup the specified serial port so that its input and output queues are used for `kbdq` (the queue for keyboard I/O) and `printq` (the queue for console output I/O, as we saw before). `ns16552special` sets the pointers passed (`kbdq` and `printq`) to point to the `Uart` input/output queues. So, anyone reading from the "keyboard" will be reading from the Uart input queue, that will in turn be written by the serial line driver as characters

come in. The rest of `ns16552intall` simply initializes the several kinds of serial boards you may have installed. We will ignore that.

```
main
   ...
      intrenable()
```
Enables an interrupt and installs its handler.

- `trap.c:19,57`
  Going down to `intrenable`, it enables the interrupt `irq` after setting up an `Vctl` structure for the interrupt. Although we did not look into `trap`, note that it is also called by interrupts, which are handled like traps. The `trap` function uses a `Vctl` array to index with the trap number and obtain a "vector control" structure to learn how to handle the trap—more generic programming.

- `io.h:44,56`
  Among other things, the `Vctl` structure holds the interrupt number, a `tbdf` field which identifies the place of the device in the bus hierarchy, and a pointer for the interrupt handler `f`. The handler admits an argument and there is a place for the argument (`a`) in the `Vctl`. When an interrupt (or trap) happens, `trap` takes the `Vctl` and calls `f(a)`. That is an easy way to reuse a given `f` by supplying different arguments to it.

- `trap.c:24,31`
  A newly allocated `Vctl` is initialized. The name supplied to `intrenable` is stored in `v->name`. That way, the kernel (and its users) can know who allocated the interrupt. The interrupt number, handler, and its argument are stored too. In our case, name would be `eia0` or `eia1`, and the argument for the handler would be 0 or 1—telling `ns16552intrx` which one of the two UARTS is interrupting.

- `trap.c:33,34`
  Locking the `Vctl` to avoid someone changing the handler under our feet, call the architecture specific `intrenable` supplying the `Vctl`.

```
main...
   intrenable
      i8259enable()
```
Enables an interrupt in the PIC.

- `i8259.c:131,147`
  For the "generic" architecture, `intrenable` is `i8259enable`. It updates the `i8259mask` and installs it on the PICs. The interrupt number is checked to be valid.

- `i8259.c:148,152`
  Also, if the interrupt is not level-triggered (it is edge-triggered) it is not allowed to be shared. Shared? Yes, look at the `Vctl` and see how there is a pointer for a next handler. On the PC, interrupt numbers are scarce, and devices may share them. The kernel will call the drivers sharing the interrupts, and they should cooperate to determine which one was actually responsible for the interrupt.

- `i8259.c:153,157`
  Here is where the new interrupt mask is programmed, and the interrupt is enabled. It is now passing through the PICs from the device to the processor. If the interrupt enable flag is set, the processor may be interrupted now by this interrupt and `trap` will call its handler(s) using the `Vctl(s)`.

`main`

- `main.c:152`
  `mathinit` enables some traps and interrupts, used by the coprocessor to notify FPU (Floating point unit) errors.

  `mathinit()` *Enables FPU traps/interrupts*

- `main.c:552,559`
  Calls to `trapenable` and `intrenable` would be setting up `Vctl` structures to let `trap` know that the kernel is prepared to service FPU related events.

- `main.c:153`
  `kbdinit` (in `kbd.c:397`) allocates ports for the keyboard, as well as the keyboard interrupt. After consuming any character from the keyboard (from the impatient user) it enables the keyboard interrupt. The keyboard interrupt handler translates keyboard generated keycodes into `Runes`, that are the "characters" of the Unicode standard (Plan 9 uses unicode instead of ascii, have you Japanese friends?)

  `i8253enable()` *Enables the clock interrupt*

- `main.c:154,155`
  If a `clockenable` exists for the current architecture, it is called. For us, it is `i8253enable` that enables the clock interrupt using `clockintr` as a handler. Clock ticks from the programmable timer are now arriving to the processor. `clockintr` will be discussed later; it is the heartbeat of the system.

3.8 Setting up I/O

`main`
  `printinit()` *Initializes console output.*

- `main.c:144`
  the call to `printinit` initialized the queue used for `print` in the console. What? the queue? Let's see that.

- `/sys/src/9/port/qio.c:25,49`
  Time ago, Plan 9 used Streams [15] to do I/O. The data structure found here, is the distilled replacement for Plan 9 3rd edition. It defines a `Queue`. A queue is the structure used to read/write bytes from/to a device or any other kernel beast. The author uses that because it would not be good to block a process writing on the console just because it takes a long time to put bytes in the serial line. It is better to place the characters in a queue and, when the line is ready, process them and put the bytes through. This is an example, but there are many similar situations and I hope you get the picture.

- `qio.c:29,30`
  Queues hold `Blocks`. A `Block` is a buffer waiting for I/O. Some part of the kernel puts a buffer in a queue, and another part is expected to consume the buffer some time in the future.

- `portdat.h:123,135`
  `Block` is defined here, and provides pointers (`next` and `list` to link up blocks sitting in a queue). `base` points to the start of the buffer, and `lim` determines the end of the buffer. The other two pointer `rp` and `wp` are the read pointer and the write pointer. Routines writing to the Block advance the write pointer, and routines reading advance the read pointer. The portion of the buffer still to be read lies between `rp` and `wp`. Note the pointer to a `free` routine. The allocator of a Block can supply the buffer from whatever memory allocator it chooses (maybe just static memory), and set `free` appropriately so that when the buffer is no longer used memory is released.

- `qio.c:32,35`
  These values summarize the memory held by the queue (i.e. by the blocks in the queue).

- `qio.c:43,46`
  These locks are used to queue processes waiting for data to read in the queue, as well as processes waiting for buffer space in the queue so they could write.

  Now let's see a bit of the implementation.

- `qio.c:74,364`
  Several routines provide the programmatic interface for `Blocks`. They are all you need to manipulate and access the buffering provided by the `Blocks`. They know that blocks are linked together.

- `qio.c:370,403`
  `qget` is a routine called by readers of a queue.

- `qio.c:375,383`
  After locking the queue, it sets its state to `Qstarve` if there is no data to read, and returns a null block.

- `qio.c:384,387`
  If there is a block to read, it is removed from the queue.

- `qio.c:390,396`
  When the state is `Qflow` (the writer was stopped because it was writing too fast), and the queue has less than half its capacity, the writer is awakened (it will continue and write) and the `Qflow` flag is removed so that any other write can proceed.

We saw this to get a flavor of I/O using queues, but we'll see queues in chapter 5 that discusses files and I/O.

3.9 Preparing to have processes

The kernel has almost booted, and we have gone a long way already. You now know a bit of the data structures used and how the system glues to the hardware. The things to come are more interesting and a bit higher-level than what is past.

- `../pc/main.c:156`
  `procinit0` should be called actually `procinit`, but there is another routine with the same name.

  `procinit()` *Initializes the process table.*

- `../port/proc.c:386,400`
  it creates a process table containing the number of processes initialized previously in `conf`. All process table entries are linked into a free process list. Each entry contains what the system knows about a particular program in execution.

  `initseg()` *Initializes allocation for segment (images).*

- `../pc/main.c:157`
  Processes need (program) images to run. `initseg` initializes the `Image` allocator in `../port/segment.c` by doing the same other allocators do: `Images` are allocated and linked into a free list. You will learn later what is an image.

3.10 Devices

- `main.c:158`
  `links` is called to initialize devices. However, there is no C source file with a `links` function definition. What happens here?

- `../port/portmkfile:45,46`
  the script `mkdevc` is called to generate `9pcdisk.c` from `9pcdisk` or any other configuration file.

- `mkdevc`
  generates a source file from the configuration file (i.e. the value of the `$CONF` variable as given to `mk`).

  What does it generate? Let's look at both `pcdisk` and `pcdisk.c`.

- `../pc/pcdisk.c:9,31`
  First, external `Dev` structures are declared for entries under (i.e. indented below) "dev" in `pcdisk`. You see, `rootdevtab` for `root`, `consdevtab` for `cons`, etc. What is a `Dev`? By now, think of it as a bunch of procedures (i.e. pointers to functions) describing how to operate on a device. The interesting thing for you now is that they have a `reset` procedure. To pick up one, `etherdevtab` is declared in `devether.c:431,450`, and it contains a reference to `etherreset`.

- `pcdisk.c:32,57`
  Now, a `devtab` array with pointers to `Dev` structures is built by `mkdevc`, it is null terminated.

- `pcdisk.c:59,66`
  For entries in the configuration file named `*.root`, `*code`, array declarations are generated together with a length variable (The actual arrays are not being declared now). Looks like "`.roots`" need this to get initialized; but forget this now.

- `pcdisk.c:67,76`
  This is the interesting thing for us now. For each `thing` in the conf file under `link`, an external `thinglink` function is declared. That is our initial entry point for ethernet and communication devices. The convention is that a `thing.c` file would contain the `thinglink` function. By generating this code automatically, to add a new "linked" device, the author only needs to write a new C source file, add the entry for the device in the configuration file, and recompile the kernel. Of course, the scripts won't work unless the author follows name conventions.

`main`

`links()` *Links device drivers into other drivers.*

- `pcdisk.c:77,92`
  The generated `links` function, calls the link procedures for the "link" devices configured into the kernel. What are links? Well, you see how most entries under `links` are for ethernet cards; concrete ethernet drivers can be linked to a generic driver supplying the common functionality. You get the picture. (Note also how for `*.root` entries, a call to `addrootfile` is generated. That is initializing some kernel-supplied "files" used to get the system working; More on this later).

  `ether8003link()` *Links the WD8003 ethernet driver*

- `ether8003.c:267,270`
  For `ether8003`, you only have to go to file `ether8003.c` and look at function `ether8003link` (saw the name convention?). If calls `addethercard` supplying a card name and a pointer to a `reset` procedure.

  `addethercard()` *Links and ethernet ethernet driver*

- `devether.c:302,311`
  `addethercard` is simply registering (linking!) the card into a table with configured cards. Later, the `reset` procedure of the card will be called to prepare it for use. We don't discuss it here, but `devether` is a generic ethernet device that uses services of concrete ethernet card drivers. For instance, the kernel uses `devether` to start and stop ethernet cards, and `devether` uses `Ether` structures to locate procedures for starting and stopping the concrete ethernet card involved.

  How is the `Ether` structure being filled up? When later, `devether` calls the `reset` procedure registered by `addethercard`, it is supplied an `Ether` structure that it must fill up. Starting to see how things fit together?

- `pcdisk.c:95,98`
  Just for curiosity, see how the `knownarchs` array mentioned before is also generated. For our local configuration, the only specific architecture is that for Intel based multiprocessors; everything else is a regular PC.

- `main.c:160`
  now that generic devices have their concrete devices linked into, call `chandevreset`.

`main`
  `chandevreset()` *Resets device drivers.*

- `../port/chan.c:50,56`
  Forgetting about the "chan" thing, `chandevreset` iterates through the `devtab` generated by `mkdevc` and calls all `reset` procedures. That prepares each device for operation. As one device may depend on another, the configuration file has (at the right of the line configuring a device) the name of "it-depends-on" devices. `mkdevc` must honor dependencies when generating `devtab`.

`etherreset()` *Resets the ethernet ethernet driver*

- `../pc/devether.c:337`
  Using again our ethernet card as an example, `etherreset` may look rather complex for us now, but it just tries to detect (linked) ethernet cards and prepare them for operation.

- `devether.c:361,362`
  This is where `reset` for the ethernet card linked before is called. The purpose of this routine is to try to detect cards, and initialize any interface (i.e. shared memory between the host and the card, or I/O ports, or whatever) used to talk to them. If a card is detected, `reset` should say so with its return value so that `devether` knows whether there is yet another ethernet card or not.

- `ether8003.c:71,142`
  Most of the code of `reset` has to do with playing the dance from the card manual to determine if the card is there and what kind of card is it.

3.11 Files and Channels

.

First a quick remark, I am still discussing initialization carried out by `main`. But since files and channels are so central in Plan 9, let's say a bit about them before continuing with the source. You are advised to read `intro(2)` until the point where processes are discussed. Manual pages for system calls mentioned below are also relevant.

In Plan 9, a file is an entity serviced by a server process over the network. That is a generic definition. Of course, the "server process" can be the kernel, or a user process, and the "network" may be some kernel code to glue a local, in-kernel, file server with a local user of the file. Note that "local" here means "within the same node".

Files are used with the traditional unix interface: `create`, `open`, `close`, `dup`, `read`, `seek`, and `write`. And files are still (as they were in UNIX) a (named) sequence of bytes. Files are deleted with `remove` (kind of UNIX's `unlink`, but a bit different). `stat` and `wstat` are used to read and write file attributes. Directories are read like files, but they are written either by using `dirwstat` to change attributes of a directory entry, or by using `create` or `remove` (they add and delete directory entries for the file affected).

This (procedural) definition of what is a file, refers to procedures that can be applied to files, either to obtain new files or to manipulate and destroy them. However, when there is a network between the file and the file user (the caller of the procedures), something has to be done instead of calling the procedure with a procedure call.

What Plan 9 does is to translate calls to file procedures in the client machine to RPCs to the server. In case you didn't know, an RPC is a remote procedure call. It works by sending a message from the client to the server when a procedure is called, and receiving later another message with the procedure result. The steps are mostly as follows:

1. The client (the caller of let's say, `write`) calls the procedure (`write`).
2. A piece of code (stub) in the client implements the local procedure actually called by the client, but that is not the real procedure being called. The client stub builds a message with the identifier of the procedure being called, and a copy of the parameters passed to the procedure.
3. The client stub sends the message to the server process (the one implementing the procedure being called).
4. The server process is a process listening for messages requesting procedure calls. It receives the message sent by the client.
5. The server process unpacks the message and determines the procedure to be called. Depending on the procedure being called, the server knows what parameters are in the message, and unpacks them.
6. The server process calls the actual procedure (`write`) with the parameters just unpacked.
7. The procedure returns, yielding some results.
8. The server process builds a reply message with the procedure result.
9. The server sends the reply message back to the client
10. The client stub receives the reply
11. The client stub unpacks any output parameter and result received, and returns pretending that it was the stub the procedure that computed the result.

Now, the protocol defining the request (called transaction in Plan 9) and reply messages to perform operations on Plan 9 files is called 9P. It is defined in section 5 of the manual. It is a protocol, and not a bunch of unrelated RPCs, that means that both the client and the server using 9P are expected to follow 9P rules. You can take a look at `intro(5)` to see what's going on.

How does the client get in touch with the server to issue 9P requests? 9P does not specify that—read: 9P permits you to use any way you can imagine to get in touch with the server. You are expected to get a network connection between the client and the server by any other means. Once you have the connection, the client and the server can talk 9P on it.

For remote files, the connection is likely to be either a TCP or an IL stream (IL is the Plan 9 transport of choice) over an IP network. For local files, you still have a "connection" between the client and the server. I'll now describe this one.

3.11.1 Using local files

The client is your local machine. Consider a process calling a file procedure like `write`, it specifies a small integer (a file descriptor) representing the file where to `write`. File descriptors are obtained with `open` as in UNIX.

- `/sys/src/9/port/portdat.h:555`
  The kernel knows which one is the current process, and locates the `fgrp` field of its `Proc` structure. The `fgrp` points to a File Descriptor Group structure. See figure 3.5.

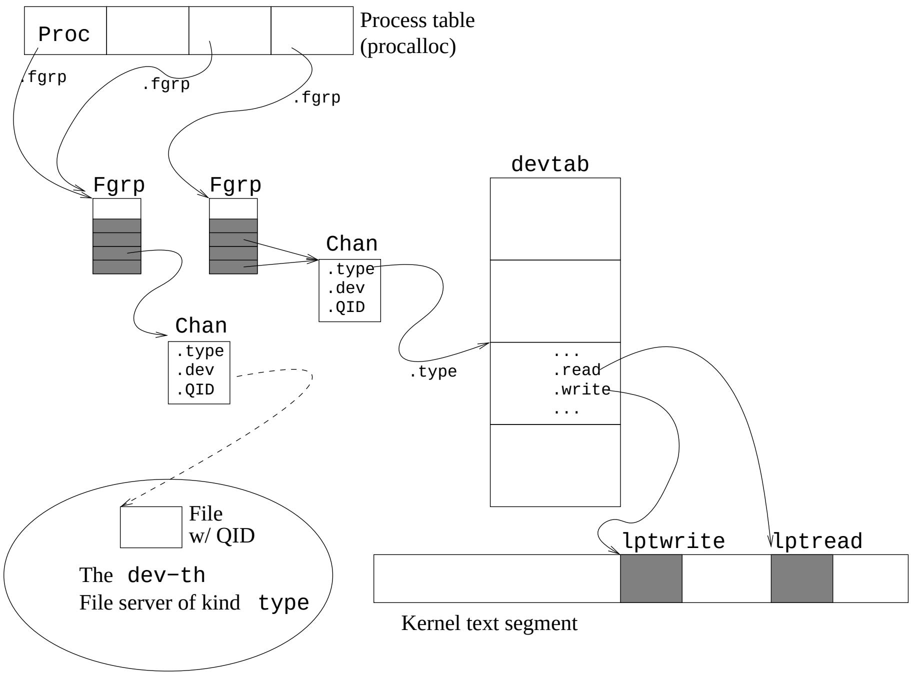


Figure 3.5: The user uses file descriptors (indexes into the `Fgrp` descriptor array) to specify files; but the kernel uses channels to point to routines knowing how to perform file operations.

- `portdat.h:431,437`
  The `Fgrp` contains an array of pointers to `Chan`. Every `Chan` represents a file being used, and every file descriptor is just an index into the array in the `Fgrp`. `open` allocates a new entry in the `Fgrp` and places a Chan on it. The index for the entry is given to the user as the descriptor for the file just opened.

We know what is a file from the point of view of the client (a descriptor), but where is the server? and where is the file on the server? The answers reside in the `Chan` structure, with a bit of help from other structures elsewhere. To answer the questions, let's follow a bit of the path that a `write` system call walks.

`syswrite()` *`write` system call.*

- `sysfile.c:444`
  `syswrite` is called, with `arg` holding a file descriptor, a pointer to a buffer, and a number of bytes.

- `sysfile.c:452`
  `fd2chan` translates the (integer) file descriptor into a `Chan` structure, by looking at the `Fgrp` for the current process.

- `sysfile.c:453,468`
  Ignore this by now. It is for handling directories and checking errors.

- `sysfile.c:469`
  Here it is! The `type` field of the `Chan` structure is used to determine the kind of device implementing the file (the device can be just "software", of course). Now, by indexing with `type` into `devtab`, the author gets a reference to the `Dev` structure for the device implementing the file.

  Say that "file" is a printer, the pointer to a `Dev` structure found at `devtab[type]` would be a pointer to `lptdevtab`, the `Dev` declared at `../pc/devlpt.c:209,228`. This is because the `type` in channels pointing to local printers is simply the index in `devtab` for the `lptdevtab` entry.

  Now, still in `../port/sysfile.c:469`, the procedure pointed to by `write` in `devtab[type]` is called. If you look at `lptdevtab`, it is `lptwrite` the procedure actually called. `lptwrite` is given a pointer to the `Chan` for the file being written. Besides, note how the offset where to write in the file is taken from the `offset` field of the `Chan`. You can see how a `Chan` in Plan 9 is a reference to a server file. You also start to see some implications, like that to share a file offset, the `Chan` must be shared, which means that processes sharing the `Chan` must reside on the same machine.

`syswrite`
  `lptwrite()` *Writes on a file in a `lpt` device.*

- `../pc/devlpt.c:148`
  In our example, `lptwrite` gets called, with the `Chan` for the file.

- `devlpt.c:155`
  At this point, something interesting happens. The `lpt` driver (the file server in this case) takes the `qid` field from the `Chan`. The QID identifies the file in the server. It only has sense within the server. Here, the server is just the local printer driver, and the driver is checking the QID to see which files should be written (it services several files).

The QID is made of two numbers, `path` and `vers`. Actually, it is `path` that identifies the file. In this case, `path` holds the value `Qdata` for the printer data file. But it could be any of `Qdlr`, `Qpsr`, and `Qpcr` for files with names `dlr`, `psr`, and `pcr` also serviced by the printer driver. Two files (within the same server) are the same file if they have the same `path` value in their QIDs. This means that a client of the file server can check if two `Chans` refer to the same file by checking their `paths` (assuming the files are within the same server. If a file is removed and created, it should get a new `path`.

The `vers` field of the QID is used to distinguish different versions of the file. It is useful because someone might be caching a file, or might like to know if the file has changed since it last got its QID.

Two files are exactly the same file, and thus have the same contents if they reside on the same server and have the same QID. This is also important for caching. If a cache has a copy of a file, and the server still has the same QID for that file, there is no need to refresh the file copy in the cache: it is the same one!

- `devlpt.c:165`
  Another interesting thing happens. There can be several printers. Which one is the one used for `write`? The `Chan` structure has a `dev` field that specifies which particular device is being used. So, `type` and `dev` fields together identify a device in the kernel. The `type` field of the `Chan` is used to select the appropriate implementation for file operations, and the `dev` field is then used to select the appropriate device of that kind.

So, what is a file for the client process? A file descriptor that leads to a channel. Where is the server? The `type` and `dev` fields of the channel know. Which one is the file? The `qid` field of the channel knows.

An what about remote files? For remote files, (not discussed now), the `type` field would select `mntdevtab` among `devtab` entries. `mntdevtab` (`devmnt.c:920,939`) is the `Dev` for the mount driver, a driver that implements device operations by issuing RPCs to the server process. The mount driver uses the connection to the server supplied by the the caller of `mount(2)` to talk 9P with the server implementing the remote file.

3.11.2 Starting to serve files

Going back to `main`, the kernel is already servicing some files (see `root(3)`).

- `../pc/main.c:158`
  `main` calls `links`

- `../pc/pcdisk.c:78,81`
  which calls `addrootfile` for `cfs`, `kfs`, and `ppp`.

- `main.c:160`
  `main` also calls `chandevreset`, which calls `reset` for configured devices, including a call to `rootreset`: the `reset` procedure for `rootdevtab`.

- `../port/devroot.c:78,90`
  `rootreset` is simply calling to `addrootdir` and `addrootfile` several times.

So, `main` makes multiple (indirect) calls to `addrootfile` and `addrootdir`. Let's discuss them now.

- `../port/devroot.c`
  This is the "device driver" for the root of the file system. It services a flat directory implementing the well known "`/`" directory.

```
main
  ...
     addrootfile()
```
Adds a new file to the `root` device file free.

- `devroot.c:62,66`
  `addrootfile` "creates" a new file in the directory serviced by the root device.
  `addroot()` *Adds an entry to the tree.*

- `devroot.c:46,47`
  At most `Nfiles` can be placed in the directory serviced.

- `devroot.c:48`
  This is a admittedly simple file system. When other parts of the kernel create a file into `devroot`, they supply file contents as well. Remember buffers named `cfscode`, `kfscode`, etc. declared by `mkdevc`?

  `rootdata` (`devroot.c:19`) is an array of pointers to the data of the (at most `Nfiles`) files serviced. So, `addroot` uses the first free entry to plug the file contents in. There are `nroot` files, from 0 to `nroot-1`.

  Where do file contents come from? Consider for example the `kfscode` array. The `mkfile` is calling `../port/mkroot` using `kfs` as an argument, which is calling `data2s`. `data2s` takes a binary from the running system where you are compiling the kernel (`kfs`, which is a program), and generates an assembly file (`kfs.root.s`) with the definition of an array (`kfscode`). The array contents are the contents of the file. That file is later assembled and linked into the system. That is how regular Plan 9 binaries are linked into the kernel to be used as "root files". You can imagine that root files are files needed to boot the system (i.e. to connect to a true file system, etc.) and cannot be loaded from the disk/network file system (chicken and the egg problem).

- `devroot.c:49`
  That was the contents, and the name, permissions, etc? `rootdir` is an array of `Dirtab` structures, containing attributes for the files serviced. In Plan 9, attributes reside within directories (well, they can reside at any place the file server wants to put them at). So, `d` is the pointer to the directory entry for the file being "created".

- `devroot.c:50,52`
  File name, length, and permissions are set in the directory entry for the file.

- `devroot.c:53`
  Crucial! a QID for the file is invented. For `devroot`, files are numbered 1 to `Nfiles`, so that file `i` resides at `rootdata[i-1]` and `rootdir[i-1]`. Clients using the directory serviced will obtain QIDs for their files and pass them back to `devroot` when requesting a file operation. The server must be able to locate the file quickly given its QID. An index is a just fine way.

- `devroot.c:54,55`
  In Plan 9, directories are also created with `create`, as files are. When the `CHDIR` bit is set in the permissions given to create, the file server understands that it must create a directory; not a file. It is the convention in 9P (yes, 9P) that the path component of QIDs have the `CHDIR` bit set for directories. So, these two lines of `devroot.c` are adjusting the QID of the file to look like a directory in case the permissions said to create a directory.

- `devroot.c:56`
  The number of created files is incremented. The next time a file is created, it would be placed in the next entry.

- `devroot.c:62,75`
  Pay now attention to the arguments given to `addroot` in both `addrootfile` and `addrootdir`. Understood? Well, it's true, I didn't say that directories have by convention 0-length.

- `devroot.c:80,90`
  The calls being made by `main` to the root driver are simply populating the root device with empty directories holding nothing. But now that there are directories, they can be populated!

  However, although `devroot` is prepared to service its files, no one has a "connection" to `devroot` yet. Well, as `devroot` is local, I mean that no one has a `Chan` with the `type` set so that `devroot` will service the file at the other end.

3.11.3 Setting up the environment

Most of the environment for the first user process, and the ones to come is provided by files. For instance, environment variables, names for the host, the user, etc. are provided by files serviced from the kernel. Now that you understand a bit of what does this mean, let's enumerate some files implementing part of the user's environment. If you want a user's description of what is provided, refer to manual section 3. It describes devices that service file trees from the kernel, as `root` does. You should read `intro(3)` at least.

Take a quick look at any of them and don't be worried too much if you don't understand what is going on. Some of them (eg. `env`) are fairly easy to understand though.

- `../port/devcons.c`
  implements the console device. It supplies files for console I/O and also for other diverse tasks like user authentication, time of day, rebooting, etc.

- `../pc/devarch.c`
  We saw a bit of it. It supplies files to identify the processor, see allocated irqs and I/O ports, and files to do I/O from user processes.

- `../port/devenv.c`
  Provides environment variables. They are serviced from the kernel as files in Plan 9.

- `../port/devpipe.c`
  Pipes

- `../port/dev....c and ../pc/dev...c`
  and many others.

I will comment some kernel drivers through this commentary, as I did with the `root` driver.

3.12 Memory pages

After our incursion into the files serviced by the kernel, we are back to `main`.

You are assumed to have at the very least some concepts on virtual memory from a basic operating systems or architecture course. You don't? I think you will survive, but study that topic a bit...

- `../pc/main.c:161`
  Just initialized the device files, but have more work to do. Now `pageinit` initializes page (frame) allocation, to prepare for paging.

`main`
    `pageinit()` *Initializes the page allocator.*

- `../port/page.c:13`
  `pageinit` has the important task of initializing the `palloc` page allocator. Right now the kernel cannot create a process because it can not allocate pages for its code, data, and stack; not yet.

- `portdat.h:444,459`
  The page allocator holds a `Page` structure for each available page frame known by the system. You can image that it is a big data structure—in fact, remember that when calculating available kernel memory the author had to take into account the size of this array. By now, see how it looks: looks like the allocator implements LRU for free pages.

- `portdat.h:279,293`
  The `Page` structure keeps for each one the physical address in memory, virtual address for user, and disk address on a file. It also has a virtualized copy of the modified/referenced bits maintained by the MMU. Page contents come from either a file or a swap file, `image` points to that. If you look at `Image` you see how it contains `Pte` structures that keep pointers to `Pages`. Don't be worried. It's simple: pages are either in use by in-memory images of files or they are free in the `palloc` array.

- `page.c:19,20`
  A big `Page` array is allocated at once for the number of pages defined in `np0` and `np1` in `palloc`. Both `np0` and `np1` were initialized at `xalloc.c:77,78` with the number of pages "eaten" by `xalloc` from the two memory banks defined in the `conf` structure.

  See how the author fights fragmentation? It has no sense to allocate and deallocate `Page` structures as they are needed. There will be at most as many free pages as free page frames at boot time. The author allocates a big array, and then links `Pages` on the appropriate list—`Images` or `palloc` as they are allocated/released.

  > **Lesson:** To avoid fragmentation, avoid using dynamic memory whenever possible. Allocate many resources at once and then keep on using them.

- `page.c:21,22`
  More defensive programming. You must have enough memory for the page array, but maybe you don't.

- `page.c:24,36`
  One page at a time, bank 0 pages are linked into the free list. `p0` is kept pointing to the physical address where a page is being "moved" to the free list; `np0` has the number of pages in bank0 not in the free list; and `freecount` has the number of pages in the free list.

  `color` has to do with caching. Two different pages can have a cache conflict because both ones collide on a cache sitting between the page and the page user. For instance, think that even pages use entry 0 in the processor cache, and odd pages use entry 1. It is better to use pages 3 and 4 than it is to use pages 3 and 5. Got the picture?

  The algorithm uses `NCOLOR` "colors" to "paint" the pages. If a page is ever allocated for a given user, and that user has color "1", it is better to give him a page of color "1". That is because pages of the same color are far apart in the memory and the author assumes that the bigger the distance two pages are at, the less the chance they will collide.

- `page.c:37,47`
  The same is done for pages on bank 1.

- `page.c:48,50`
  The list is set.

- `page.c:52`
  The number of pages for the user is the distance between the head and one past the tail.

- `page.c:52`
  The size of the physical memory in Kbytes is computed, just for letting the user know.

- `page.c:53`
  Remember the `nswap` "limit" set when the kernel first knew how many pages would be available? That was actually the maximum number of pages not found in-memory. By adding the physical memory size, you get the virtual memory size. This is just to let the user know.

- `page.c:57,58`
  These two values are used for paging. Eg. when less than a 5% of pages are free, the kernel should be moving pages to disk, to make more room. It is not healthy to let all the pages be occupied. It could be that (due to some bug) the kernel needs free memory for the algorithm that gets more free memory...

```
main
    swapinit()
```
Initializes the page allocator.

- `../pc/main.c:162`
  Back to main, `swapinit` initializes swapping.

- `../port/swap.c:21,33`
  Set `swapimage.notext`; which means that there is no swap file yet. Perhaps an explicit initialization would make things more clear: `swapimage` has been initialized to all-zeroes by the loader. Its `c` member that points to the `Chan` for the swap file is still zero, which means that there is no channel to the swap file. In any case, a "swap map" (`swmap`) is allocated. It reflects the state of portions in the swap file. A page array for the max number of pages being paged out to swap (`conf.nswppo`) is also allocated.

3.13 The first process

You should know already what a process is, at least from a theoretical perspective and as a user. To remind you, Plan 9 can use a machine with just one CPU and pretend that there are multiple programs executing at the same time. The way to do it is to let each program run a small fraction of time. As the processor is fast enough, if programs are given small fractions of processor time repeatedly, they would appear to be executing all at the same time. A program in execution is called a process.

A process is more than a program, it has open files, is run on the name of a user, has memory for data allocated while the program is running, a stack to keep track of nested procedure calls and to maintain local variables, etc.

To multiplex the processor among processes, a timer interrupt is typically used. The system arranges a timer to interrupt the processor, say, 200ms in the future, and loads the processor context with register values appropriate for executing the user process code. When the interrupt arrives, the hardware (and the interrupt handling software) saves the context of the processor as it was right before the interrupt. That set of registers, if reloaded, would permit the process to continue where it was. But usually, the kernel loads the context of a different process instead, letting it run for another quantum of 200ms in this example. At some time in the future, the saved context for the process in this example would be reloaded and it will not even know that another process was using the processor before.

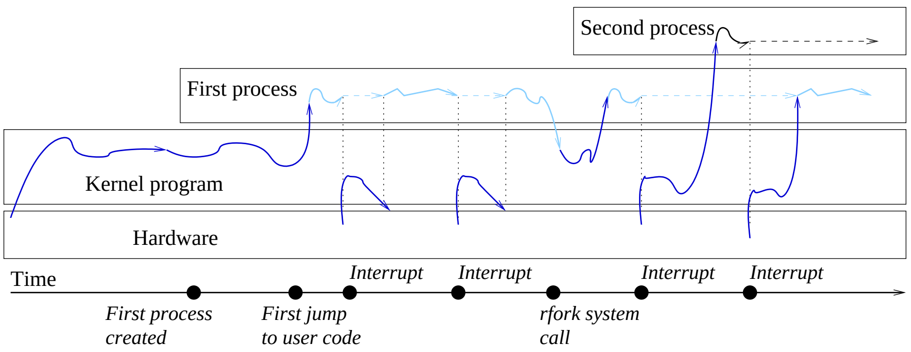


Figure 3.6: The kernel is just a program, and there is just a flow of control. However, users believe that their processes are executing sequential programs.

In a previous section, you were advised to read `intro(2)` until the point where processes are discussed, now go and read all of `intro(2)` and `fork(2)`. Most of what the code you are going to read is doing is also described at `boot(8)`. You are also advised to refresh your "theory" of what a process is in case you are lost by this point.

From now on, I will be describing some important fields of the process data structure. I suggest you try to guess how they are used. One fine way of doing it is (after reading the manual pages I suggested) using `grep` on the source to see where are those fields used. Try to guess what is the field being used for.

3.13.1 Hand-crafting the first process: The data structures

Right now, Plan 9 is just a flow of control running a program (the kernel) loaded into memory. That is not enough to provide system services, there should be a real process that could later use `rfork`/`exec` to create new ones and execute other programs.

You can see figure 3.6 to get a glance of how our current flow of control will evolve into a set of processes. In the figure you see how initially, the kernel started executing and then it creates the memory image for a first process. It will later jump to protection ring 3 to execute the user code for it. After that, the kernel would only execute to service interrupts and system calls—(in the mean time the process would be ready to run, but would not be running). This first process would make `rfork` system calls to create new processes, and the clock interrupt will be used to multiplex the processor among existing processes.

- `../pc/main.c:163`
  About to call `userinit` in `main`. It is the procedure that creates the first user process. After that is done, what remains is to jump to that process and let the system run when interrupts, exceptions, or system calls request any system service.

`main`

`userinit()` *Initializes the user environment and creates the first process(es).*

- `main.c:236`
  Previously, the kernel allocated an array with `Proc` entries. That was the process

table. Each entry is a `Proc` struct with the information needed to implement processes. For example, the `Proc` structure is used to get to the saved processor context needed to put that process back in the processor. Now the author allocates a free entry in the process table. That entry will be filled up to initialize (create) a new process.

```
main
  userinit
     newproc()
```
Allocates a process table entry.

- `../port/proc.c:307,314`
  `newproc` waits until the free list has at least a free entry. The `resrcwait` call would make the current process wait due to the specified reason ("no procs"); it would be preempted or moved out of the processor. Think that the current process, willing to create a new one, has to wait until a free entry be available. By the way, what is the current process? By now, none. However, there are free entries and we break the loop at line `:309`. The kernel is creating the first process and by now there is no current one. Note the use of the loop around the `wait`, because there is no guarantee that by the time the current process gains the lock in line `:313` any free entry be still there. Another process could run instead, gain the lock, and allocate the just released entry.

- `proc.c:315`
  The process entry is removed from the free list. `p` is the process begin created.

- `proc.c:318`
  This will be clear when we discuss processes in chapter 4. Processes have a "state". The process state tells the system on what condition the process is in. In this case, the process is being handled by the scheduler, the part of the system that decides who runs next and switches from one process to another.

- `portdat.h:493,504`
  You should recognize at least the typical states `Ready` and `Running`. To get an idea of how `state` is used, consider that dead processes (those that terminate or abort) have their `state` set to `Dead`; functions doing things to processes while they are alive can check that the `state` is `Dead` and do nothing to dead processes.

- `proc.c:319`
  This is the state name as printed by `ps(1)`. It is a new process. `psstate` holds a representative string for the state the process is in. That can effectively turn `ps(1)` into a debugging tool. It also permits you to inspect a process from a different machine, `psstate` would make sense even on a different architecture: It is just a string that is understood everywhere.

- `proc.c:320`
  `mach` points to the `Mach` structure of the processor where the process is running. Did not run yet. It is necessary to know on which processor the process is running at, because on that processor is where the actual process context is (while a process is running, the set of registers last saved for him by the hardware

is irrelevant). The convention is that `mach` is zero only when the process context is saved elsewhere and its user code is not running.

- `proc.c:321`
  Not on the free list anymore.

- `proc.c:324`
  This is discussed in the chapter for processes. It is used to make the process wait due to some reason. Not waiting anything now.

- `proc.c:325,328`
  Some resources used by the process cleared. Let's say something about it now before continuing.

Take a coffee, go read `rfork(2)` (reread `intro(2)` if you forgot), and come back later.

- `proc.c:325`
  Processes have a name space. More precisely, processes are within a process group and process groups have a name space. `pgrp` is a pointer to a `Pgrp` structure representing the "process group". For me, the initial "p" is confusing because it sounds like "process" and not "name", but the author would have a good reason for the name. The name space determines what names lead to what files. Actually, the name space is very similar to that of a UNIX machine. In UNIX, there is a mount table (for the whole system) that specifies paths that lead to file systems. For instance, "`/`" leads to the root file system, "`/cdrom`" may lead to a file system on a cdrom, etc. Now, back to Plan 9, every process has a name space. The name space is made of a mount table that associates paths to file systems. For example, one thing the first process will do is to associate "`/`" to the `root` file system implemented by the root device.

  The interesting thing in Plan 9 is that every process (group) has its own name space and can add and remove entries without affecting other processes. Adding a new entry is called *mounting* a file system or `binding` a file (see `mount(2)`). Deleting entries is called *unmounting* a file system.

  This is very important because it provides an engineering solution to several important problems on distributed systems. For instance, by binding `/$objtype/bin` onto `/bin`, each process will find appropriate binaries for the current architecture; binding files under `/dev` is a fine way to make a particular process see a different device–perhaps at a different node!

  Processes can share their name space (i.e. their process group) too. That means that several processes may have the same value in their `pgrp` field.

- `proc.c:326`
  Processes have environment variables. Both variables and values are strings. An example of an environment variable is `PATH` (which, by the way, is mostly unused in Plan 9). Again, each process has its own environment (made of a set of variables). `egrp` points to an Environment Group or `Egrp` structure.

- `portdat.h:413,429`
  It maintains variable names and value strings. Several processes may share their environment; `egrp` may be the same for more than one process.

- `proc.c:327`
  Processes have open files. As we saw before, each process has a file descriptor group. `fgrp` points to an `Fgrp` structure.

- `portdat.h:431,437`
  It has the set of file descriptor entries for the process. You already know that each file descriptor leads to a `Chan` structure that is all you need to perform an operation to the file referenced by the `Chan`. Again, processes may share their file descriptor group. Several processes may have the same value for `fgrp`.

- `proc.c:328`
  Processes can synchronize by using `rendezvous(2)` . To rendezvous, processes call `rendezvous` supplying a tag that must be the same for the two processes doing the rendezvous. Tags are actually local to a "process rendezvous group". `rgrp` points to a `Rgrp` structure.

- `portdat.h:407,411`
  This implements the rendezvous group. It is obvious that processes can share their `Rgrp`.

- `proc.c:329`
  Sometimes, a process gets broken and has to be debugged. The debugger is another process that controls the execution of the debugged process by placing breakpoints, altering the debugged process state, and processing events of interest for debugging. `pdbg` in a process being debugged points to the debugger process `Proc` structure.

- `proc.c:330`
  This has to do with how math coprocessors work. It is really expensive to save and reload the coprocessor context. Therefore, the system tries to avoid unnecessary coprocessor context switches by not loading the coprocessor when a process does not use it. `fpstate` records the state of the coprocessor. The author uses it to know if the coprocessor is being used. By now, it was not even initialized to a reasonable state. `FPinit` says so.

- `proc.c:331`
  You probably think that all processes execute user programs, running with the processor in non-privileged mode (ring 3). And that these processes enter the kernel only to service an interrupt, a trap, or a system call. That is not true. There are processes that spend their lives within the kernel. These are called kernel processes. For example, in Plan 9, there is a process called "pager" that has the important task of doing page outs when low on free page frames, by stealing page frames from other processes. This is better implemented as a process with its own (sequential) flow of control that mostly sleeps until a point when it starts moving pages to disk. You can imagine that to move pages to disk, the process must do I/O and must have special privileges, since it must

have access to the pages for the processes involved. For both reasons, it is implemented as a process that never leaves the kernel, it starts executing a at a given function within the kernel. After that, it is handled as a regular process—with special privileges, admittedly. `kp` is true when the process is a kernel process.

- `proc.c:335`
  Processes move. They move from one processor to another. This is important because if a process has just run at a given processor, the processor's cache will still have cached memory for the process. It is much cheaper to run this process again on that processor than it is to run it at a new one: The new processor will remove from its cache other memory and it will have to move in new entries for this process; besides, the old processor has now unuseful cached memory. `movetime` keeps the earliest allowed next time for a move, to help in taking into account "processor affinity": a process has affinity for the processor where it did run last time.

- `proc.c:336`
  Processes can be wired to a processor, so they always run on it. `wired` points to the `Mach` structure for the processor where the process is wired to.

- `proc.c:337`
  `ureg` points to a saved `Ureg` for the process used for notes. Forget this now if you don't understand.

- `proc.c:338`
  `error` maintains the error string for the process (see `errstr(2)`). Errors are represented by strings (portable, readable) in Plan 9. They are set either by `errstr` or by a failed system call.

- `proc.c:339`
  Processes have a virtual address space. That means that different processes have different page tables and their virtual to physical address translations differ. In Plan 9, the user part of the address space (what the user process sees) is organized as a set of segments. Every process has a `TEXT` segment with the code, a `DATA` segment with its data, and a `STACK` segment with the stack. There are other segment kinds like `BSS`, for uninitialized global data, etc. Some of them are initialized later for this handcrafted process.

  Processes may share segments, although some segments, like stacks, are not shareable—that would make no sense. The chapter on virtual memory discusses this. Each process can have up to `NSEG` segments, and at this line, `newproc` is clearing all the pointers to the `NSEG` segments.

- `proc.c:340`
  Processes have a pid that identifies them. The `pid` is unique within a node. `incref` takes a `Ref` structure, that contains a counter, and increments it. The author uses `incref` and not `++` because some other processor might be creating a process—well, not now. `incref` uses locks to avoid races. So, in this line, a new `pid` has been allocated. It is "1" because `pidalloc` was initialized to zero early during kernel bootstrap.

- `proc.c:341`
  `noteid` identifies a "process note group". Processes may be posted notes, which are kind of UNIX signals. Notes are strings, though—and may be posted through the network using `proc(3)`! The system itself posts notes to processes when they get an exception, and they can prepare to handle the note as said in `notify(2)`. You can post a note to a group of processes (a set of processes with the same `noteid`), and all processes with that `noteid` will get the note. A new note group has been allocated.

- `proc.c:342,343`
  If so many processes have been created that `pidalloc` or `noteidalloc` get wrapped down to zero (by a set of increments), panic.

- `proc.c:344,345`
  If there is no kernel stack for the process (which is the case) a new stack is allocated. Remember that `smalloc` keeps on trying until the stack can be allocated. Right now the kernel is running into a rather borrowed kernel stack. The current stack belongs to the current processor, not to the current process, and each process should have its own kernel stack where the hardware will push the processor context on traps and interrupts, and where kernel procedures will be called during system calls made by the process.

  The author tries not to waste time by keeping the kernel stack for a process that existed before; its `kstack` would be non-nil and the one used in the previous life of its `Proc` would be reused for the new process.

- `proc.c:347`
  All set. If you take a look to it all, the author has initialized everything to just nothing—except for the kernel stack, which comes along with the allocated `Proc`. The process must now be given some resources to start working.

```
main
  userinit
```

- `../pc/main.c:237`
  Back to `userinit` in `main`, it starts to give resources to the process. At this line, create a new namespace `pgrp`.

  `newpgrp()` *Allocate a new process group*

- `../port/pgrp.c:43,51`
  `newpgrp` simply allocates a `Pgrp` structure with everything set to null. It also allocates a new process group id for the just created group and sets `ref` to 1.

- `portdat.h:49,53`
  Perhaps it's time to say a bit about `Refs`. a `Ref` structure contains a lock and a counter. It is used to do reference counting—hence the name—although it seems to be reused to allocate pids, etc. too. A reference counter (`Ref`) is simply a counter with an integer value that corresponds to the number of users (i.e. references) of the data structure the counter is associated with. For example, the `Pgrp` just allocated is used only by the process being initialized. Therefore, the

counter must be one. Whenever a new user of the referenced structure appears (e.g. a new process starts sharing our `Pgrp`), the `Ref` is incremented. When a user disappears (e.g. a process using the `Pgrp` ceases to exist) the counter is decremented. If the counter ever reaches zero, the data structure is no longer used and can be deallocated. The `Lock` is necessary because multiple processes (and processors) could be updating the counter at the same time. You might say that it would have been better to implement reference counters by atomic increments and decrements on word-sized values as the Linux kernel and some others do. However, the Plan 9 approach has two advantages: it is portable and simple to understand. By not optimizing what does not need to be optimized, the machine dependent part can be kept small, and the code can be kept more understandable—ever looked inside of the Linux kernel? give it a try.

> **Lesson:** Do not optimize; ever. Do optimize only when you really have a performance problem. And be very reticent regarding what is a performance problem: processors are increasing speed dramatically.

Another reason I did not say is that the "atomic increment" done by Linux is not so atomic—not a single instruction; at least not in all the cases and for all the architectures.

- `../pc/main.c:238,239`
  The `Egrp` (environment variables) is also allocated—initialized to zero, and its reference counter set to one. It puzzles me why there is no `newegrp` routine. Admittedly `newpgrp` assigns an id, but it is definitely not more complex than it would be "`newegrp`".

- `main.c:240`
  `dupfgrp` (which duplicates an `Fgrp`) is called with a null value to create a new file descriptor group for the process. Again, maybe a `newfgrp` routine would avoid surprises for the reader, although the code is pretty clean and the author knows the good reason that `newfgrp` does not exist.

  `dupfgrp()` *Create/duplicate a file descriptor group*

- `../port/pgrp.c:170,183`
  First, a new `Fgrp` is allocated and if no `Fgrp` to duplicate was given, the `Fgrp` is initialized by allocating a chunk of entries. The `DELTAFD` name suggests that the `fd` array is resized on demand to contain `DELTAFD` extra entries. That is in fact the case. It is a usual technique employed to avoid fragmentation and save time to extend arrays dynamically with "delta" new entries every time they run out of entries—instead of allocating a rather big array or refusing to admit more entries, or allocating one extra entry at a time.

  It returns with a brand new file descriptor group with no open file (no channel linked into) and `DELTAFD` ready entries. The reference counter is again 1.

- `../pc/main.c:241`
  `newrgrp` creates a new rendezvous group for the process.

  `newrgrp()` *Create a redezvous group*

- `../port/pgrp.c:53,61`
  Again, this just allocates the `Rgrp` and sets the reference counter to 1.

- `../pc/main.c:242`
  `procmode` is set to 640 (octal). Processes are handled by files—like everything else. `devproc` implements the driver supplying `/proc` files, that represent processes. This field is simply the permissions for the process, in the file system view. Using a file for everything is a nice way of fixing how permissions are checked: in the same way file systems check permissions on real files.

  Although processes are files, there is no way to create new processes by using file system operations. The author considered that it wouldn't pay the effort, which is reasonable since processes are created locally, within the same node. The same happens to several other system calls that cannot be performed using the file system.

- `main.c:244`
  `text` contains the name of the file where the process text (code) comes from. In this case the author uses the string `*init*` because there is no file.

- `main.c:245`
  Positive discrimination at work. `eve` is the name of the user who boots the machine. Well, `eve` is the name of the array in `../port/auth.c:37` that holds the name of the user who boots the machine. Conventionally, in Plan 9, that user is referred to as "eve". You know, "Adam and Eve". In Plan 9 there is no "root" (superuser), but "eve" is given more privileges than other users. The first process certainly runs on the name of the user who booted the machine, hence this line.

- `main.c:246`
  Ok, the kernel did that before. But the code is cool, isn't it?

  `fpoff()` *Places the FPU into an "inactive" state*

- `main.c:247`
  This executes some instructions to place the coproprocessor in an "inactive" state. `FPOFF` is defined at `l.s:373,378`. By the way, this and the previous line should be replaced by a call to `procsetup` (`main.c:564,569`).

3.13.2 Hand-crafting the first process: The state

```
main
   userinit
```

- `main.c:250,256`
  Preparing to switch to the first process. `sched` is a member of `Proc` of type `Label`. A `Label` is like an oversimplification of a `jumpbuf` in C. It is buffer where a copy of a program counter and stack pointer is kept. Labels are used to implement coroutines. More soon.

- `main.c:255`
  The program counter registered in `sched` is the address of the `init0` routine. That function will be the very first thing executed by the process being created and its purpose is to do some final arrangements before jumping into the user program for the process.

- `main.c:256`
  The stack pointer saved in `sched` is the address of the allocated kernel stack (`kstack`) plus the size of the allocated stack minus something. On the Intel, stacks grow downwards; to lower addresses. The address of the allocated stack is the lowest address in the stack, but the stack pointer for the empty stack would point to the biggest address of the allocated stack on an Intel. The "minus something" thing is to reserve some space at the bottom of the stack. What's the reservation for?

- `../port/portdat.h:91`
  `MAXSYSARG` is the maximum number of arguments for a system call. Here, the assumption is that each argument occupies at most a word (if it occupies more a pointer would be used).

- `../pc/main.c:256`
  The number of words for arguments is multiplied by the number of bytes in a word, because adding to `kstack` makes it move counting characters, not words. By the way, the "-12" in the comment seems to be a relic: 5 words times 4 bytes per word is 20. I guess at some point there was a `kstack-(12+something)` and the 12 was removed and the comment kept obsolete. But who knows.

  Taking into account that the reservation is for system call arguments, for me it looks like the author wanted to ensure that the (theoretically) pushed arguments for a system call made within the kernel always are valid memory. There is a check in `trap.c:504` that precisely ensures that. For a kernel process, the arguments would not stand in the "user stack", but in the kernel stack instead.

  What is needed is a kernel stack that has a fake return PC on it so that `gotolabel` could overwrite it to "return" to the PC in the label. But I am not sure more extra room be needed.

- `main.c:261`
  The first segment for the new process! (Not to be confused with a hardware segment) It is going to be an `STACK` segment. Must start at `USTKTOP` (initial top of user stack) minus `USTKSIZE` (the size of the stack: it grows downwards!). The number of pages in the segment must be the number of pages to get `USTKSIZE` bytes. Forget a bit about how is the segment created by now. But think that the `Segment` structure is created and it contains in the end a bunch of (indirect) references to `Page` structures. By now all the pointers to `Pages` are null. Note that addresses mentioned here are virtual addresses.

  The virtual memory for this process will look like the one depicted in figure 3.7 (in that figure you can see the virtual memory for a second process too, so you could get the feeling of how this works). Right now the kernel is running on the

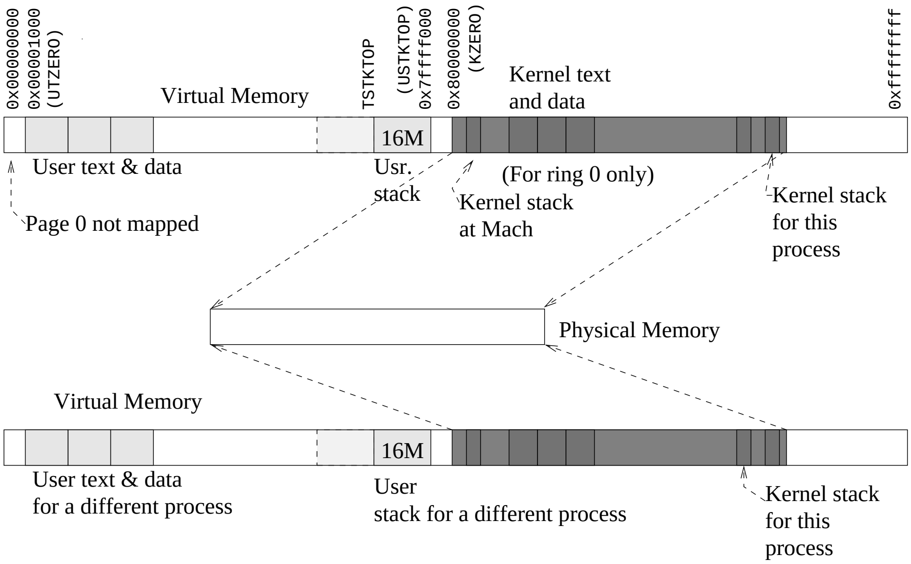


Figure 3.7: Virtual memory for user processes. Permissions in the hardware page tables will be set so that user code (ring 3) is unable to access the last two gigabytes of the virtual address space.

stack near the `Mach` structure; although it will later run at the kernel stack for the process being created.

- `main.c:262`
  The stack segment is placed into the `seg` array for the process. The stack segment always resides in the `SSEG` entry.

- `main.c:263`
  Got the segment, but it has no memory. The call to `newpage` obtains a new fresh page; later `segpage` plugs the page into the segment. The page is requested to be cleared (the "1" for `newpage`). `UTKSTOP-BY2PG` is the virtual address for the page. The "0" for `newpage` is a pointer to the Segment reference and we are not (yet) interested what that is for. Ignore how `newpage` works by now, but take into account that a page is allocated from the page (frame) allocator, its reference count gets incremented, its referenced/modified field (`modref`) is cleared, and its virtual address (`va`) is set to the one given to `newpage`.

- `main.c:264`
  `segpage` attaches the `Page` into the segment at the virtual address specified by the `va` field in the page. One of the pointers of the `Segment` points now to the page. The segment has memory! And the user has an initial stack to issue procedure calls on it.

  One thing to note is that the MMU knows nothing about this page yet. Only the kernel data structures know that the segment has a page there. The MMU page table will be updated when the page is first used.

- `main.c:265`
  The page is "mapped" for the segment, but we are running in kernel now, and not even within the context of the process holding the segment (i.e. not using its address space). `kmap` maps the page for the kernel and returns the address the kernel should use to access the page.

- `dat.h:202`
  Nice. `kmap` only has to add `KZERO` to the physical page address. Remember that the kernel did set up a one-to-one mapping for physical memory at address `KZERO`? That's why. No need to add temporary entries to the current page table just to access physical memory. But, why is `kmap` named "map"? It is conceptually mapping the page for kernel usage. It doesn't matter if all the feasible maps were done before. The use of `kmap` instead of just ORing `KZERO` also allows the author to change his mind and use another implementation in the future without changing the code that depends on `kmap`. Interfaces are important in that they isolate pieces of code.

  > **Lesson:** Keep clean interfaces between different components of your programs. Interfaces allow you to modify one component without affecting the others.

- `main.c:266`
  `VA` gives the (user) virtual address for a given kernel address, and `bootargs` places arguments for the initial process in the page whose virtual address was given. By convention, Plan 9 processes receive two arguments for their main program, the argument count, and the array of arguments. Each argument is just a string. And by convention, the argument zero is the program name. See `exec(2)`.

  By using arguments and environment variables, users can control what the executed program will do.

- `dat.h:201`
  Because on PCs, the kernel shares the virtual address space of the process running, `VA` simply returns its argument. .

```
main
   userinit
      bootargs()
```
Sets up arguments for the first user process.

- `main.c:305`
  `bootargs` is building arguments for the user process entry point in the user stack. `sp` is set to the top of the (empty) user stack. Again, space is reserved for `MAXSYSARG` words.

- `main.c:307`
  `ac` is the argument count. zero by now.

- `main.c:308`
  first argument. `av` is the argument vector (i.e. like `argv`) and its first entry points to the string "`/386/9dos`" just pushed on the user stack by `pusharg`.

The first argument is the program name. In this case, it is customary that the program be called `9dos`, the Plan 9 version for PCs.

`pusharg()` *Pushes an argument in the user stack.*

- `main.c:286,294`
  `pusharg` only advances the stack pointer to make room for the string given, and copies it there. It returns the new stack pointer (which points to the first character in the pushed string).

- `main.c:309`
  `cp` was set to point to the address where the kernel loader placed the bootline. Ensure that it is null-terminated.

- `main.c:310`
  In `buf`, the author is going to adjust bootline. The adjusted version will be pushed as an argument. The `64` in the declaration should probably be replaced by `BOOTLINELEN`.

- `main.c:311,318`
  If bootline started with `fd`, place the canonical full name for that: “`local!#f/fd0disk`”. That saves typing for the boot user and keeps the first process happy. The “`#f`” is the path to the kernel floppy device. The name is pushed on the user stack and placed into the argument vector.

- `main.c:314,316`
  The same for hard disks.

- `main.c:317,318`
  If option “n” is given to the first process, it understands we are booting from the network. Surprisingly, option “n” is missing from the `boot(8)` manual page (boot is the first process).

- `main.c:319,324`
  If the argument was a floppy or a hard disk, put the name of the disk into the `conf` structure as if the user said `bootdisk=...` in `plan9.ini`.

- `main.c:328`
  The stack pointer points to words, but `pusharg` puts characters in the stack. This is ensuring that dummy bytes are pushed on the stack in case that is needed to complete a word. But all the arguments are now pushed.

- `main.c:331`
  Make room in the stack to put the argument vector—with pointers to the arguments just pushed. The `+1` is because it has to be null-terminated (did you read `exec(2)`?).

- `main.c:332,335`
  `lsp` is the argument vector passed to the user program. It is built by pushing in the stack each of the recorded arguments in `av` and null-terminating it. `av[i]` was the kernel virtual address for the pushed argument. The added expression

is to "move" that address into the one seen by the process. It is a translation. The length of the translation is the difference between what the user code thinks is the start of the page allocated for the stack, and what the kernel thinks is the start of that page. This addition is necessary: remember that the kernel is using a virtual address that depends on the `KZERO` mapping for the stack (e.g. bigger than `KZERO`), but the user will use its own user virtual address for the stack (e.g. lower than `KZERO`). The `KZERO` map has no permissions to be used by code running at ring 3. Otherwise, a user program would be able to access all physical memory installed.

Perhaps the usage of a "kernel-virtual-to-user-virtual" or "kv2uv" macro would make this more evident.

- `main.c:336`
  The same adjustment has to be done to the stack pointer itself. Beware, I did not tell, but `sp` is a global variable pointing to the user stack for the boot process. That is why it is adjusted and the routine just returns. It is probably a global because `init0` is the one using `sp` to jump into the user code. Because we are going to loose our stack when jumping to `init0` later, the author cannot easily pass the argument using the kernel stack of the new process. The `sizeof ulong` adjustment is for the argument count. By the way, what happened to the argument count? Does boot not use it and nobody noticed? Or are we missing something?

```
main
  userinit
```

- `main.c:267`
  The user stack is all set. So release the kernel map for the stack page; i.e. do nothing.

- `main.c:272`
  A segment of type `TEXT` is created for the process. It starts at `UTZERO` (the zero address within the text segment). Just one page suffices (that's 4K).

- `main.c:273`
  `flushme` has to do with caching. Forget it.

- `main.c:274,277`
  The segment is placed in the list of segments for the process (slot `TSEG` for the text segment). A page allocated at address `UTZERO` and linked into the segment as before. Ignore the `memset`; it is honoring `flushme`.

- `main.c:278,280`
  The text segment for the process is initialized with the contents of `initcode`. That is the program run by this process. If you look into `../pc/initcode` you'll see that it simply execs `boot`.

  Hint: You run new programs in new processes by doing a `fork` and then an `exec`. By hand-crafting the first user process, the system is doing the `fork` (It cannot use `fork` because there is no process to fork). The best way to avoid hand-crafting an `exec` is to use a silly user program that calls the regular `exec` system call. You're done, and you can start as the first process any program you want!

- `main.c:282`
  `ready` changes the process state to `Ready` (ready to execute) and links the `Proc` into the scheduler ready queue—the queue of processes ready to run. More on that later.

3.13.3 Starting the process

- `main.c:164`
  `userinit` is done, and `schedinit` starts up the scheduler now that there is a process to schedule. `schedinit` never returns, so `main` is really done.

`main`
  `schedinit()` *Initialize and start scheduling.*

- `../port/proc.c:57`
  `m` points to the `Mach` structure for the current processor. In `../pc/dat.h:160` you can see how it contains a `Label` (you know, PC and SP buffer). `setlabel()` *Record PC and SP.*

- `../pc/l.s:497,498`
  `setlabel` receives a pointer to a `Label`. It first loads `ax` with the pointer to the label passed. `FP` is the "frame pointer" that points to the activation record for `setlabel` in the stack.

- `l.s:499`
  the current stack pointer is copied into `label[0]` (considering `label` as an array of integers now).

- `l.s:500,501`
  The stack has the typical layout for a function call. The function right now is `setlabel`. The top of the stack contains the "return pc", i.e. the address where to continue executing upon return from `setlabel`. That address is stored in `BX`, and then `BX` is placed into `frame[1]` (considering `label` as an array of integers now, assembly uses offset 4 because an integer has 4 bytes). The return PC points to the instruction in `schedinit` right after the call to `setlabel`.

  So, `m->sched` has the PC and SP corresponding to the point of execution right after `setlabel` was called.

- `../port/proc.c:58`
  `up` is a pointer to the current user process. Right now, it has a null value. So, why does `schedinit` check for `up`? You will know in the next chapter; You are not expected to, but can you guess why?

  It is obvious that `up` cannot be a global variable, because, how could then different processors have different user processes? `up` is defined at `../pc/dat.h:263` to be the `externup` field of the `Mach` structure for the current processor. For each processor, that field points to its current process. On uniprocessors, a global `up` would suffice.

- `proc.c:83`
  The scheduler is called. `sched` picks up a process and switches to it. Right now there is only a first (boot) process.

```
main
   schedinit
       sched()
```
Schedule a process.

- `proc.c:93`
  If `up` is not null, there is a current process and the routine must save its state; otherwise the kernel won't be able to come back to the process when it be allowed to run again. But there is no current process yet; forget this now.

- `proc.c:107`
  `runproc` chooses among the set of ready processes one to be run. `up` is set to point to it. Until the next time `sched` runs it will be the current process (for this processor).

```
main...
   sched
       runproc()
```
Elect a process to run.

- `proc.c:201`
  Interrupts allowed. From now on, the timer, and any device can request our attention. Interrupts will be serviced using the kernel stack of the boot processor, by now.

- `proc.c:202,203`
  `idlehands` does whatever the kernel should be doing while there is nothing to run on the processor. It is defined as "do nothing" at `../pc/fns.h:43`. So, whenever no process is ready to execute the kernel would be looping here doing nothing until an interrupt (or another processor) changes things.

- `proc.c:204,243`
  The loop keeps on looping until a process is found for running. Whenever it gets that process, it jumps to the `found` label, where the fortunate process will be put on the processor. The only thing to notice right now is that there is an initial process and the jump to `found` is done. Although it is also interesting that the loops leaves `rq` pointing to the run queue where the selected process resides. There are several run queues as we will see in the next chapter.

- `proc.c:246`
  Interrupts cleared: The routine is messing up with the queue of ready processes, if an interrupt arrived and could change things, the kernel could crash.

- `proc.c:247,248`
  What if another processor is also picking up a process to run? The routine should wait until that processor be done with the scheduler queue. So, if it

cannot gain the lock of the run queue, it goes back to the loop. Interrupts were disabled before locking the queue, but that affects just to the processor running this code, other processors are free-running. The goto will again enable interrupts and reenter the idle loop that searches for ready processes. The routine will again detect that a process can be run, and try to lock again the scheduler queue (the process that it found before the `goto loop` might still be there if not picked up by another processor).

This is busy waiting (i.e. no semaphores), but since the routine cannot even know who should run, what else can it do? You might say that the author could let the old current process run a bit more time, but it could be that 1) there is no such process or 2) such process is now blocked waiting for something to happen.

This is not the case now. There is one process (`boot`) for us to run.

- `proc.c:250,255`
  Got the lock, now search for a process `p` that runs at this processor (affinity!). `head` is the head of the processor ready queue and the `rnext` field of `Proc` is used to link processes into this queue. The loop selects any process `p` that either

  1. did run in the processor running `runproc`, for affinity. The routine can know that because the `mp` field of `Proc` has the number of the field `machno` in the `Mach` structure where it did run last.
  2. did not move recently (to avoid trashing, i.e. moving a process repeatedly between several processors). The code knows that `p` did not move recently because in `movetime` `p` has the earliest time when it is allowed to move; more below.

  `l` points to the `Proc` right before `p` in the run queue.

  For us, `p` points to the "`*init*`" process just created by `main`.

- `proc.c:260,263`
  If the run queue is empty, go back to the loop. It could be empty because all processes can be blocked (not the case). It also goes back to the loop when the process is running on a different processor (the process state is at the processor, and not saved within the kernel; see the comment to learn how to know that).

- `proc.c:264,271`
  Note how easily the process is removed from the run queue. It is no longer ready, it is going to be running. The counters for the number of processes in the run queue and the number of ready processes are updated.

- `proc.c:272,273`
  This should not happen. The kernel (as any program) tries not to flood the user with messages. Whenever something important has to be said, it does so; otherwise it remains silent, because, who cares? Guess why this shouldn't happen?

- `proc.c:274`
  Done with the run queue, let other processors use it.

- `proc.c:276`
  The process is being handled by the scheduler. Perhaps this could be moved right before line `:274`, to make sure that if by any bug, the process gets linked back to a run queue, everybody will see that it is scheding and report the bug in line `:273`.

- `proc.c:277,279`
  Must honor conventions. `mp` must have the number of the `Mach` (processor) where the process last run on. `mach`, the pointer to the structure is a different thing and may be null. Also, `movetime` is set to the current time at the boot processor plus 1/10 of second. `HZ` has the number of ticks per second and `ticks` gets incremented every tick. The author uses the time at processor 0, to make all processors agree on the current time on an MP machine (every processor has its own vision of time depending on the exact point when it was initialized).

  So, `p` is removed from the ready queue and attached to the processor where it is going to run (it is not going to be ready, it is running).

```
main
   schedinit
       sched()
```
Schedule a process.

- `proc.c:107`
  Back in `sched`, the pointer to the current process is setup as I said before.

- `proc.c:108,109`
  Update the state and set the `mach` pointer. Its state is going to be in the processor. Also, the `proc` field of `m` is updated. You can go from `Mach` to the current `Proc` and vice-versa.

- `proc.c:111`
  `mmuswitch` changes the address space to that of the new process.

`mmuswitch()` *Switches the MMU to another page table.*

- `../pc/mmu.c:126`
  This is the only line executed now, it is not really switching the "Intel task". It will load the page table given: the page table for the processor. If the process has its own page table (lines above) that one is used instead; not the case! The kernel stack for this process—used by interrupts and traps that occur while running at user level—will be setup to the top of the currently empty kernel stack of the process. `kstack` points to the memory allocated for the stack but that is not the top of the empty stack.

`taskswitch()` *Switches the Intel idea of the current task.*

- `mmu.c:27,40`
  It is updating the TSS used by this processor to ensure that the kernel stack pointer will be set at `stack`, within the address space determined by the page table pointed to by `pdb`. Besides, it loads the new page table pointer into the MMU. The kernel has just switched to the address space of the just created process, but it is still running using our boot processor stack. Don't worry, the

kernel is mapped the same way at all page tables used for Plan 9 processes, so our (kernel) virtual addresses keep on pointing to the same place in memory. The chapter on virtual memory will make this more clear.

- `../port/proc.c:112`
  Our mind is going. Remember that `sched` was setup to have the PC for `init0` and the SP pointing to the end of `init0`'s parameters pushed manually by `main`.

`gotolabel()` *Jumps to a saved context*

- `../pc/l.s:489,490`
  `gotolabel` resets the stack and program counter with those of `label`. It now fetches the pointer to the label into a `AX`. We consider `label` as an array of int, although it is not.

- `l.s:491`
  The stack pointer is set to that in `label[0]`, the old boot processor stack is gone. At this point the processor starts using the kernel stack for the new process.

- `l.s:492,493`
  The trick!, when `gotolabel` returns, the machine jumps to the return PC (theoretically) pushed last on the stack by a `call` instruction. By pushing the `PC` saved in `label[1]` (i.e. the start of `init0` code), the `ret` at line `:495` "returns" to the PC in the label.

- `l.s:494`
  The convention is that functions returning an int use register `ax` to pass the value back to the caller. Usually, the `1` just returned will appear to be the return value of `setlabel`, but that is explained in the next chapter.

`init0()` *(Kernel) entry point for the first process.*

- `main.c:190`
  We are running at the address space of `*init*`, using its kernel stack, and starting at the first instruction of `init0`. Now we are a regular process executing within the kernel. Not just the flow of control taken after the machine was reset.

- `main.c:197`
  Interrupts were disabled during the context switch, but now the kernel can handle them again.

- `main.c:199,206`
  The comment states that `chandevinit` will not call any `rootinit`—there is none. Remember `chandevreset`? The same thing again. The comment regarding early kernel processes means this: you do not have root and current directories for this process, and although you want to setup such things for the user code about to run, it is good to setup them now so that kernel processes started before jumping to user code could at least have silly root and dot directories, provided by the root device. I don't know why there is no `rootinit` function with this code in, the author knows.

- `main.c:203`
  `slash` is a pointer to a `Chan` structure for the root directory. It is set to point to a channel given by `namec` ("name channel"), which corresponds to a file with the given name in the current name space. What? I said namespace and there is no root yet? Yes. Plan 9 have absolute paths, relative paths, and paths for kernel device files. "`#/`" refers to the root of the file tree serviced by the root device. Devices service file systems named by "`#`*character*". "`/`" is the character for the root device. The reason to have device paths is that no matter how many adjustments a process makes in its name space, it still has access to kernel devices and can access system provided files—this is not the whole truth.

  So now `slash` points to the directory serviced by `root`—which is the root of `root`'s file tree. You will see how this happens in the chapter devoted to files. But you already got a glance about it when we discussed channels and `root`. Try to read the code anyway.

- `main.c:204,205`
  `cnameclose` simply removes the given name—not the whole truth. Right now, it is "`#/`", and that's a funny name for our `/`. So the author creates a new name for the channel. What? channels have names? Yes they do. Think that they are caching the file name.

- `main.c:206`
  `cclone` clones (dups) a channel. Now `dot` "points" to the file where `slash` points.

- `main.c:208`
  `chandevinit` calls the `init` routine for every `Dev` configured in `devtab`. We are now a regular process, have a regular (reasonably sized) kernel stack, are handling interrupts once in a while, and devices can be initialized. I are not going to describe how the various `init` functions work. Only when they be relevant for the topic of one of the given chapters I'll do so.

- `main.c:210,223`
  Various environment variables are set up for the current process. The lines for setting the `terminal` environment variable are generating a suitable value from the current architecture name. The kind of `cpu` is set to `386`, you know that, right?. This is very important, because `$cputype` is used among other things to choose what binaries are appropriate for this architecture (saw `/386` on your Plan 9 box?). The environment variable `service` is also important, because `boot` uses it to choose the rc script used to start system services. On terminals you want rio, on cpu servers you probably don't. For configuration (ini) parameters with names not starting with `*`, the author sets environment variables to reflect their settings. This is very important, because if the `boot` initial program is ever updated to take into account some peculiarity of the Plan 9 installation, an environment variable can be used to indicate that; just by adding a line to the `plan9.ini` file.

  Ignore the `waserror` and the `poperror`. They are described in the next chapter.

- `main.c:224`
  A kernel process is created named "alarm". It will start executing at the kernel function `alarmkproc`. That function iterates through the list of alarms searching for expired alarms. Then it posts "alarm" notes to the processes with expired alarms and sleeps until the next alarm expires. This is really good because it turns alarm handling into a sequential activity: not all processes must be setting up timers for pending alarms while in the kernel. But we are not interested on that now.

- `main.c:225`
  `touser` transfers control to the user program within the given process context. The global variable `sp` is used to tell `touser` what should be the current user stack pointer. This was initialized before by `bootargs`.

`init0()`
  `touser()` *Perform an upcall from the kernel to the user code.*

- `plan9l.s:12,25`
  The trick is to pretend that we return from an interrupt suffered while running the user program. The processor is silly and obeys, reloading the processor context with that of the "previously" running user program.

- `plan9l.s:13,19`
  All these pushes are building the fake stack frame after the (not-existent) interrupt: User PC, user code segment, user flags, user stack pointer, user stack segment. The code and stack segments are the `UDSEL` and `UESEL`, which are the appropriate selectors for `UDSEG` and `UESEG`.

- `plan9l.s:16,17`
  A fake flags word with just the interrupt enable bit set is pushed on the stack (as if it was a real flags pushed during the interrupt that saved the frame being built). After the `iret`, we are sure that interrupts will be enabled. It would be a disaster if that was not the case, because no timer could ever preempt the process.

- `mem.h:75,76`
  Users run at ring 3, with non-privileged mode. Interrupts and traps lead to a switch back to kernel mode as dictated by the IDT entries.

- `plan9l.s:20,24`
  The user data segment descriptor is copied into the descriptors for all other user's extra segments. These are not loaded by the `iret`, so they are updated by hand.

- `plan9l.s:25`
  Up to userland. After this instruction, the program in `initcode` is running at user-level, with the kernel fully initialized below to handle traps and interrupts.

- `initcode:14`
  When the program reaches this point, a trap is raised with number `0x64`, and

the `exec` system call to execute the program in `/boot` is issued, with arguments for `/boot` taken verbatim as given to us. Remaining initialization is up to `boot`!

Now, go read again `boot(8)` and take a look at what it does. That is so clearly stated there that I'd rather interfere your learning by repeating it here.

One final consideration. Most of the work done by `userinit` in `main` would be portable and could be said to belong to the `port` directory. However, some machine dependent assumptions (e.g. the growing direction of stacks) are being made. The code is more simple having a single `userinit` than it would be having a machine independent `userinit` and then a machine dependent `userinit` with the code that cannot be made portable. Portability refers to the difficulty of porting the code to a new system, not to the number of lines of the `pc` directory. If the `pc` directory holds more files, but is easier to understand, and easier to rewrite/adapt for a new architecture; that's fine!

So you now know how the kernel boots, you have an operational system and everything else is done by issuing system calls to the kernel. In the following chapters we will read the code related to servicing system calls for the major components in the system.

Chapter 4

Processes

On this chapter, we will read the code related to processes. As I will do with following chapters too, I try to follow the execution path for system calls related to the chapter topic; but I make exceptions to this discussion order whenever I feel its necessary.

Although I am not discussing source code files, one file at a time, I suggest you still try to read files as they appear. The worst thing that can happen is that you don't understand the code and come back to the commentary; but another thing that can happen is that you understand most of it or it all on your own! That would be a big progress! During this chapter, you will be reading these files:

- Files at `/sys/src/9/port`:

  `portdat.h`
  : Portable data structures.

  `portfns.h`
  : Portable functions.

  `proc.c`
  : Processes.

  `pgrp.c`
  : Process groups.

  `devcons.c`
  : Console device.

  `devproc.c`
  : Process device.

  `alarm.c`
  : Alarm handling.

  `tod.c`
  : Time of day.

  `taslock.c`
  : Test and set locks.

  `qlock.c`
  : Queuing locks.

`sysproc.c`
: Process system calls.

- Files at `/sys/src/9/pc`:

  `trap.c`
  : Trap handling procedures.

  `dat.h`
  : Machine dependent data structures.

  `fns.h`
  : Machine dependent functions.

  `clock.c`
  : Clock handling.

  `main.c`
  : Machine dependent process handling.

  `l.s`
  : coroutines, locking and other low-level routines.

. . . and several other ones used as examples.

4.1 Trap handling continued

Before looking at how processes work, let's revisit trap handling once more. I will be using the clock interrupt as an example of the trap source. The clock interrupt is important because it is the mechanism used to multiplex the processor among processes. In what follows, assume that a user process was running while a clock interrupt happen.

`trap()` *C entry point for traps.*

- `/sys/src/9/pc/trap.c:218`
  `l.s` dispatches the interrupt to the `trap` procedure, supplying a pointer to the `Ureg` structure.

- `trap.c:225`
  `intrts` is set to the result of `fastticks`. This `Mach` field records the time stamp for the interrupt. That is necessary for `devintrts` that handles interrupt time stamps.

- `devarch.c:597,601`
  The time stamp is the result of the architecture specific `fastclock` routine, which is defined to be `cycletimer`

- `devarch.c:586,595`
  that returns the time stamp counter of the processor. (In case you don't know, Intel processors from the Pentium up have a `tsc` register which is incremented by the hardware at every clock tick).

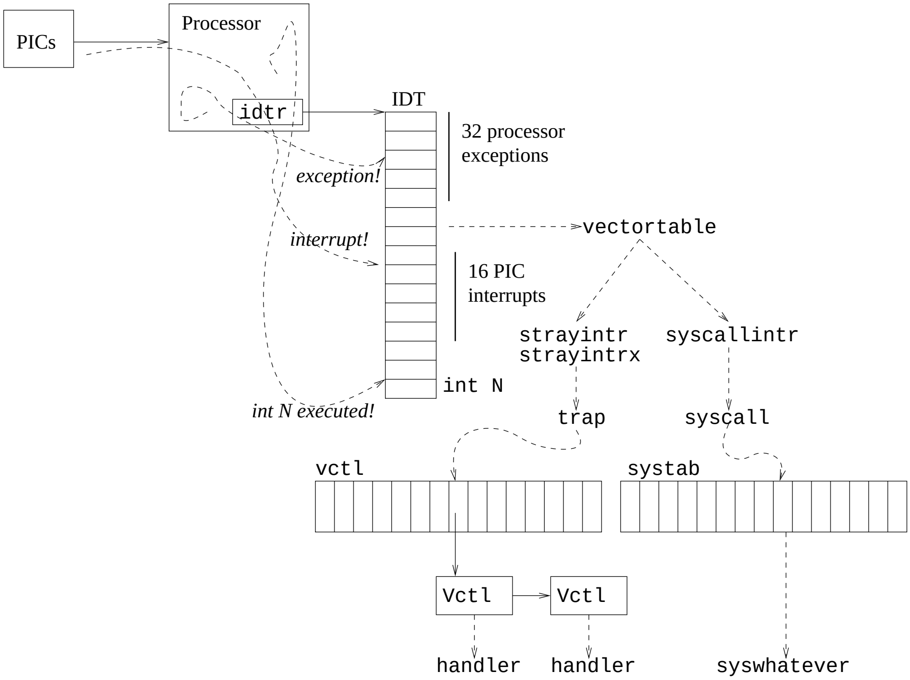


Figure 4.1: External interrupts (from the PICs) are dispatched by IDT entries at `VectorPIC`; processor exceptions (caused by a faulting instruction) are dispatched by the first 32 IDT entries; other exceptions can be "called" by `int` instructions. In the end, either `trap` or `syscall` service them.

- `trap.c:226,230`
  `user` is set to true if the interrupt (or trap) happened while running user code. It was so, if the saved code segment within the `Ureg` is the user code segment.

- `trap.c:232`
  The trap number is kept in `vno` (vector number). It is going to be used heavily and caching it in a variable will both enhance the readability of the code and make it run faster.

  What can be the value of `vno`?

- `trap.c:175,208`
  `excname` contains names for the first 32 trap numbers, which are generated by the processor. If `vno` is less than 32, it must be an exception generated by the processor. See figure 4.1 if you got lost.

  Otherwise, `vno` is either within `VectorPIC` (32!) and `VectorPIC`+16, or it is above `VectorPIC`+16. In the first case, the event corresponds to an external interrupt routed through the PIC; in the second case, it must be a "software generated interrupt" (i.e. `int n`).

- `trap.c:233`

Back to the `trap` routine, `ctl` is the vector control structure for this trap. If there was no `Vctl`, we are in trouble: only traps and interrupts that the kernel is prepared to service had their `Vctl` set up by either `intrenable` or `trapenable`. Ah, and beware of the assignment!

- `trap.c:234,266`
  The kernel is handling a trap or interrupt that someone enabled before. Things go well...

- `trap.c:234,238`
  Only for interrupts, increment the number of interrupts recorded in the `Mach` structure. And if the trap number is a "real interrupt" (not a processor exception, and not a system call), take the interrupt number from the `Vctl`. `lastintr` holds the value of the last interrupt. That is used for debugging. Perhaps a local variable would suffice at the expense of reporting unwanted interrupts only when they happen—see below.

  The check against `VectorSYSCALL` seems to be unnecessary, since the trap for system calls is routed through `plan9l.s` to enter `syscall` (below in `trap.c`) directly.

- `trap.c:240,241`
  `isr` is the interrupt service routine. If the `Vctl` has one, call it with the trap number. But, what is an ISR?

- `i8259.c:159,162`
  Our PIC is the `i8259`. When we did enable it, the `i8259elcr` (edge/level control register) had one bit set per edge triggered interrupt. The `i8259isr` interrupt service procedure was set as the service procedure for level triggered interrupts, and it was set as the end of interrupt procedure for edge triggered interrupts. Note how fields `eoi` and `isr` are used to turn `Vctl` into a programmable interrupt handler.

  `i8259isr()` *Interrupt service routine for the 8259.*

- `i8259.c:105,128`
  All `i8259isr` does, is to tell the `i8259` that the interrupt has been serviced by acknowledging the interrupt. The chip now knows that, and forgets about the interrupt until it is raised again. If the interrupt is not acknowledged, the i8259 would ignore that interrupt when raised again, because it would assume that the processor is still handling the previous one and is not prepared for servicing it again. The real interrupt number is not the one coming in `vno`. `vno` was set to `VectorPIC`+number because PIC serviced interrupts start at `VectorPIC` in the IDT. By adjusting the number, `irq` contains the interrupt number that the `i8259` knows about: from 0 to 15.

- `trap.c:242,245`
  Here is where the functions registered as interrupt handlers (or trap handlers) are called. For the clock interrupt, the handler will be `clockintr`, as said in `i8253.c:130` by `i8253enable`. Let's defer a bit what `clockintr` does, but note how this is the point where the trap (or interrupt) is actually serviced.

`Vctls` for the same trap are linked through the `next` field—there can be several handlers for the same interrupt. The check in line `:243` shows that a vector number can be enabled (the kernel knows it is ok to get that trap) even when there is nothing to do to handle it.

- `trap.c:246,247`
  You now know what this does. When necessary, it calls the "end of interrupt" routine.

- `trap.c:250,265`
  The author preempts processes here. This is discussed in the next section.

- `trap.c:267,271`
  There is no `vctl` for the trap number: no part of the kernel requested to handle it. Besides, the trap number has a name in `excname`, and the trap comes from the user program. The user did something weird and got a trap generated by the processor. After indexing with the number to get a name for the trap, a note for the trap is posted to the process. The process is likely to die. Interrupts (disabled since the trap) are enabled before posting the note.

  You can get into these lines when any of the bad things named in the `excname` array happen to the process.

- `trap.c:272,294`
  Not a "handled" interrupt, and not a processor exception while running user code. The condition checks that `vno` corresponds to an interrupt and not to a processor exception. Again, is the check for `VectorSYSCALL` necessary?

  In this case, the number of spurious interrupts for the processor is incremented and `trap` returns doing nothing more. A message is printed because this shouldn't happen. Now that we bothered the user, the `for` loop also reports other unwanted interrupts at different processors. If you read the comment, you see that the author is suspicious of IRQ 7. Most PCs keep on delivering that interrupt under certain circumstances even if you are not allowing interrupts.

- `trap.c:295,308`
  Really into trouble. It must be a processor exception while running kernel code. So the kernel has a serious bug. The `dumpregs` call prints processor registers to aid in debugging the kernel, and then the kernel calls `panic`. Only the boot processor (`machno` zero) panics, other processors sit in a loop until the panic at the boot processor causes the system to go down.

- `trap.c:310,313`
  If something was posted for the process, honor it. More on that when we read note-handling code. When `trap` returns, `l.s` will reload the processor context from the `Ureg` and it will resume the activity previous to the trap.

`panic()` *Issue a message and halt.*

- `../port/devcons.c:183,204`
  Just to satisfy your curiousity, `panic` prints the given message (dumps the stack to aid in debugging the kernel) and halts the system by calling `exit`.

`exit()` *Stop system operation.*

- `../pc/main.c:606,630`
  `exit` prints the "exiting" message once for each processor (note the use of the `active.machs` bit field. Then it waits for a while to let the console print the panic and exiting message (important if going through the serial line). Finally, if running at the boot processor on a terminal, it loops forever. For CPU servers, seems like a reboot (`reset`) is preferred to restart CPU server operation—after giving some time to let a human read the message. The `exiting` field of `active` is set to true.

  As each processor exits, its bit in `active.matchs` is cleared, and other processors will exit too because they notice the `active.exiting` bit (see `../pc/clock.c:53,54`). This is shown later.

4.2 System calls

Some system services (eg. clock handling) are requested by interrupts; some others are requested by explicit system calls. Let's complete now the discussion of system call handling.

- `/sys/src/libc/9syscall/mkfile:51,61`
  This script generates for all system calls listed in `sys.h`, a procedure that puts the system call number into `AX`, issues an `int` instruction, and returns. These procedures are called to issue system calls from user code, and are linked along with every user program. Although there are many system calls, all of them use the same trap number, and it is the parameter in `ax` that determines which one is being requested.

`syscall()` *C entry point for the system call trap.*

- `../pc/trap.c:471`
  System calls get routed to `syscall` in `trap.c` by `plan9l.s`.

- `trap.c:477,478`
  If a system call was issued from the kernel something is wrong. The saved `Ureg` CS selector is used to check if the system was running at ring 3 while the system call was issued.

- `trap.c:480,481`
  Accounting for number of system calls services, and noting that the process is servicing a system call.

- `trap.c:483`
  registers for the user program are those saved in the `Ureg`. The name is `dbgreg` because this is very useful for debuggers, to fix up a faulting program and let it continue. The last known PC for the process is that saved in the `Ureg`; remember that in the `Proc`.

- `trap.c:485`
  This line recovers the system call number from the `ax` register from the saved user context.

- `trap.c:487,490`
  Coprocessor stuff. If doing a fork (to create a new process) and the coprocessor was used, save its state in the `fpsave` for the current process. The new FPU state is inactive. By doing this, the author ensures that both the parent and the child process start with an `PFinactive` coprocessor state. The child may not use the floating point unit, ever; in that case, its FPU would be kept `FPinactive`.

- `trap.c:491`
  System call servicing may take some time; enable interrupts. All the information needed now is kept in `up` (and `ureg`). If an interrupt arrives, the current (kernel) stack will be used to service it and the current processing will continue after returning from that interrupt.

- `trap.c:497,502`
  (Ignore the error handling; just assume that line `:497` is entered). If the number is not that of an existing system call (between 0 and `nsyscall`-1), post a note for the process and report the error. As you will see, `error` would cause the routine to continue at line `:513` in this case.

- `trap.c:504,505`
  When the stack pointer for the user code (`sp`) is not in the first page mapped for the stack, or is so near the top of an empty stack that it seems to be no space for the system call arguments, check that the addresses going from `sp` to the end of the system call arguments are indeed okay. The first stack page is known to be okay because the kernel mapped it when the process was brought to life; but we cannot be sure otherwise that the user stack pointer looks fine and points into existing stack space. The kernel checks before accessing the user stack.

  > **Lesson:** Don't trust your users! If you implement any kind of service, assume that users would be as malicious as you can image. Usually, they will not be malicious, but they will have bugs that could make them behave as if they were really malicious.

- `trap.c:507`
  `s` in the `Proc` structure is set with the arguments for the current system call. `s` is of type `Sargs`, which holds as many words as `MAXSYSARG` says—i.e. as many arguments as a system call may take. The word in the top of the user stack is ignored; that would be the return PC for the "system call" assembler routine called by the user code.

- `trap.c:508`
  The string with the "ps" state of the process is updated to contain the system call name.

- `trap.c:510`
  And the system call is called. Note how the `Sargs` is supplied. Arguments reside within kernel memory and can be used at will, without checking that the stack addresses are still valid. While the system call is executing, the kernel could switch to a different process.

- `trap.c:530,538`
  The result of the system call is placed into `AX` (will be returned by the assembler user-level stub); and any note posted (will be discussed later) is handled.

4.3 Error handling

Errors are handled by several routines within the kernel. Error handling also includes routines used to report errors, be they fatal or not. See pages `perror(2)` and `errstr(2)`.

4.3.1 Exceptions in C

Errors are handled using `error`, `waserror`, `nexterror`, and `poperror`. Handled with care, these routines provide a clean and fast error handling mechanism similar to exceptions in other languages.

- `../port/portdat.h:593,595`
  Each process has an array of up to `NERR Labels` for error handling together with the number of entries in the array (`nerrlab`). There is also a place to put an error message of up to `ERRLEN` characters.

  To see how this is used, let's see what `syscall` does for error handling. In figure 4.2 you can see the whole picture. Initially, the user calls a library function to perform a system call, and it traps into the kernel (figure 4.2(a)).

- `../pc/trap.c:495,496`
  The return value for the system call is set to `-1`, which means failure. `waserror` is called inside a conditional. If it returns false, the system call will be called and return value set accordingly; otherwise, `syscall` would return the error condition.

  `waserror()` *Prepares for errors setting up an error label.*

- `fns.h:123`
  `waserror` increments the number of error labels for the process (initially zero) and fills up another label in `errlab` (initially the first error label).

- `trap.c:496`
  Go back to this line. When `syscall` was called and it called `waserror`, `setlabel` in `fns.h:123` returns zero. The expression `(a,b)` returns the value of `b`; therefore `waserror` returns zero, which means that the then-arm of the if is taken.

  During the system call the first error label is set and keeps the SP and PC for the kernel as they were in `trap.c:496` during the early system call steps (see figure 4.2(b)).

`poperror()` *Removes an error label.*

- `trap.c:511`
  The system call completed, and `poperror` is called.

- `../port/portfns.h:199`
  `poperror` simply decrements the number of error labels: it removes the label added by `waserror` in `trap.c:496`. The state is like that of figure 4.2(a) (of course, the PC and SP would be that for `trap.c:511`), and not the ones as of line `:496`.

  Now, imagine the system call number is wrong.

- `../pc/trap.c:501`
  `error` is called with `Ebadarg` to report the error. `error()` *Raises an error.*

- `../port/proc.c:1132,1138`
  `error` copies the given string into the `error` string for the process. (yes!, `Ebadarg` is a string with a descriptive text for the error: portable, human readable, simple). Once the error reason is noted, `nexterror` is called. `nexterror()` *Reraise an error.*

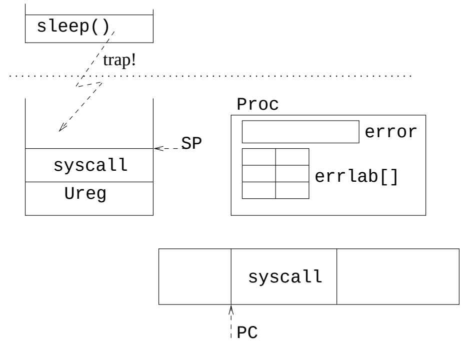
- `proc.c:1140,1144`
  `nexterror` is where things start to move. A `gotolabel` jumps to the kernel state as recorded in the last saved error label; and the number of error labels is decremented to 'pop' it off the array. In our example, the error label was set by `syscall` and both the PC and SP would be set as they were when `waserror` was called.

- `../pc/trap.c:496`
  this time, `waserror` returns true, because after `gotolabel`, the corresponding `setlabel` appears to return true. Therefore, the conditional is not taken. In effect, `nexterror` is "raising an exception", so that the flow of control resumes where it was at the last `waserror`. Stack (local, or automatic) variables are deallocated and function calls made since the `waserror` are gone without returning. Can you see how the combination of `waserror` and `error` forgets about an ongoing computation and resumes in the top-level routine where an appropriate action can be taken?

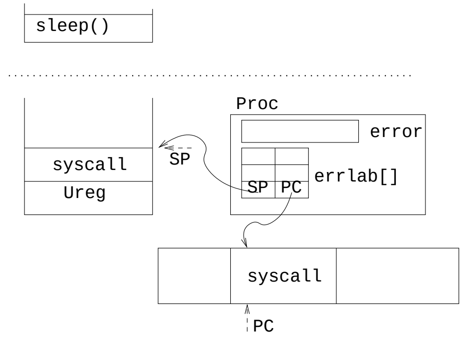
  In the figure 4.2, you can see how this works. To make it more clear, the figure corresponds to a situation where a system call is called (4.2(a)), `waserror` in `syscall` sets the error label (4.2(b)), another kernel procedure (`syssleep`) is called (4.2(c)), and this procedure calls `error` to raise an error (4.2(d)). Noticed how it works?

Things are more interesting because `waserror`/(`next`)`error` pairs can be nested. For example, the system call called in the previous example could call `waserror` again, and a routine called by the system call could call `error`. In this case, there would be two (nested!) error labels in the error stack. The first call to `error` would jump to the last label pushed by `waserror`; Then, a `nexterror` could be used to jump back again (re-raise the exception), or alternatively execution could proceed.

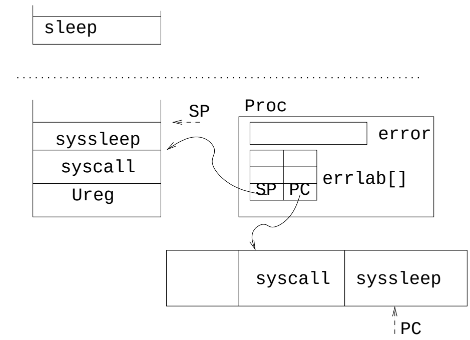
To clarify things, the scheme looks like this



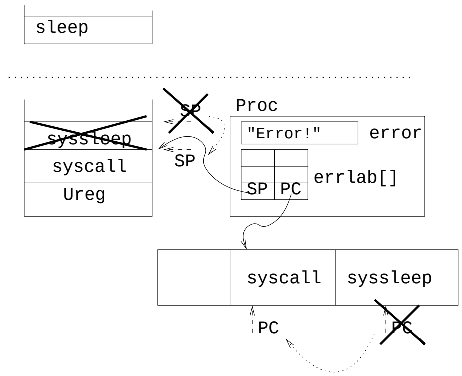
(a) Initial context: The kernel starts executing a system call. PC and SP correspond to the current kernel context.



(b) `Waserror` is called: A label is set to remember the context where to continue after any error.



(c) More calls: the kernel continues executing, more records pushed on the stack, etc.



(d) `Error` called: An error string is set, and the context is restored as it was when `waserror` was called. This time, `waserror` returns a different value.

Figure 4.2: Error handling: labels are set in `errlab` and used to raise "exceptions". `waserror` remembers the context; `error` notifies an error and restores the context.

```
top_routine() {
        // one typical idiom...
        if (!waserror()){           // (1)
                do_the_job();
                do other things;
                poperror();
        }

        // another typical idiom...
        alloc(some_memory)
        lock(a_lock_held);
        if (waserror()){            // (2)
                free(some_memory);
```

```
                unlock(a_lock_held);
                nexterror();
        }
        poperror();
}

do_the_job(){
        if (something fails)
                error(msg);
}
```

Things to note: `poperror` would remove the label pushed by `waserror`; `error` would jump right to the line of `waserror` again, but that would make `top_routine` follow the `else` arm in 1, and the `if` arm in 2; `nexterror` would jump to whatever outer routine called `waserror` before `top_routine` was called, telling the caller that we suffered an error. In the case `nexterror` is called, the reason for the error would still be `msg`. `do_the_job` does not need to return the error condition to the caller function, and how `top_Routine` does not need to check for any error condition returned by `do_the_job`.

One final note, special care has to be taken when calling `error` (or `nexterror`) because some resources could have been allocated (or locks acquired) since the last call to `waserror`. Take into account that `nexterror` does not know anything about either resources or locks; therefore, the routine that did allocate/acquire those resources/locks must call `waserror` to release them on errors (like 2 in the example). To pick up an example, add the line

```
char *p=malloc(BIGSIZE);
```

right at the beginning of `do_the_job`, and consider that "`something fails`". Got the picture?

4.3.2 Error messages

You already saw `panic` and a couple other routines. They are easy to follow, therefore I will not comment on them. Nevertheless, the `pprint` routine is curious.

The `pprint` routine reports errors not on the console as `panic` does, but on the standard error stream for the process. The kernel is assuming that `stderr` is descriptor number two and is opened for writing; not assuming too much. This is a detail that shows how Plan 9 was built with the network in mind from the ground up.

UNIX would print in the console any message about problems related to the current process but not causing a panic (e.g. an "NFS server not responding" when a network file cannot be used due to server problems). The message is of interest to the process but not to the whole system. By printing the message in the console, the user sitting there can see the message, but the process could be started from a terminal miles away! So it's better to print the message to wherever the process prints diagnostics (stderr) and let the process (owner) know. Besides, the user sitting at the console usually does not care of any problem for processes he has not started.

4.4 Clock, alarms, and time handling

The clock is used to maintain the system time (when the TSC is not available) and to implement alarms. An alarm causes a function to be called after an specified amount of time. There is a system call `alarm(2)` that can be used to request an "alarm" note to be posted to the process after the given period of time. By handing the note, the user process can achieve the effect of the `alarm`: calling a function after a period of time.

`time(2)`, `cons(3)`, `alarm(2)` are manual pages that can be of interest for you now.

4.4.1 Clock handling

Let's see how the clock works starting at the clock interrupt.

`trap`

- `/sys/src/9/pc/trap.c:218`
  The clock interrupt happens and `l.s` dispatches it to the `trap` procedure, supplying a pointer to the `Ureg` structure. Interrupts stay disabled.

- `trap.c:242,245`
  For the clock interrupt, the handler was `clockintr`.

`trap`
  `clockintr()` *Services the clock interrupt.*

- `clock.c:43,44`
  The increment notes in the `Mach` structure that another tick passed by. The call to `fastticks` updates the `fastclock` field of `Mach`; that is used by non-boot processors to update their own TSCs!. Looks like although the boot processor is in charge of time, other processors try to be in sync.

- `clock.c:45,46`
  The PC image in the `Proc` for the running process is updated to be real one. This is also done when entering a system call.

- `clock.c:48`
  Do some time accounting, as we will see in the next section.

- `clock.c:49,50`
  Record execution statistics for kernel profiling, if needed. `kproftimer` is a pointer to `_kproftimer` only when the `kprof` device has been init'ed. If not doing kernel profiling, the pointer will be nil and ignored.

- `clock.c:51,52`
  This processor is not really active. It may be exiting (or halted!) but it got yet another clock interrupt because interrupts are enabled. Ignore it.

- `clock.c:53,54`
  Some processor paniced (or started shutdown) and the kernel is exiting. If this is running at a different processor, it notices now and calls `exit` to terminate operation at this processor. `exit` resets the bit for the processor in `machs` so that other clock interrupts are ignored at lines `:51,52`.

  Noticed how one processor does not perform immediate actions on another processor? The best the author can do is to "kindly request" the other processor to do something: processors are a living thing.

  > **Lesson:** When using multiple processes (processors) to do something, do not directly interfere with the execution of others; ask them for anything you want instead. This prevents dangerous race conditions because only you can do things to yourself.

- `clock.c:56,62`
  Alarms and `clock0links` serviced here! (see next section).

- `clock.c:64,68`
  Will see in the virtual memory chapter. Some other processor asked we to flush our MMU by reloading our page table. We do so. The clock interrupt is a good place to see if anyone else is asking this processor to do anything: the requester does not need to wait too much (although it needs to wait!).

- `clock.c:70,75`
  More scheduling affairs. Forget this now.

- `clock.c:76,79`
  If the code interrupted was running at ring 3, account for another tick in a counter maintained in the bottom of the user's stack, and call `segclock` to do profiling on the user code. Interrupts will be reenabled after `clockintr` returns to `trap`, and `trap` returns to recover the user state.

  While kernel is servicing an interrupt or a system call, `ureg->cs` would not be `UESEL`, and no time will be charged to the (interrupted) user.

- `clock.c:11,18`
  To avoid synchronization problems due to multiple clock interrupts on machines with several processors, the boot processor does clock handling. That is the meaning of "0" in `clock0link`. To service "kernel alarms", i.e. stuff that needs to be done every tick for the kernel, links a established into `clock0link`. Lines :57,62 are servicing these links. A link is just a pointer to a `clock` procedure to notify of the clock tick. `addclock0link`*Establishes a procedure called at clock 0 ticks.*

- `clock.c:21,34`
  Which ones are the links? `addclock0link` inserts a new link into the list. So, `grep` for `addclock0link`!

  - `ns16552.h:104`
    The serial line UART wants `uartclock` be called every tick.

– `../port/devcons.c:1015`
  The console driver wants `randomclock` be called every tick. That is to maintain the random number generator (see `cons(3)`).
– `../port/devmouse.c:85`
  `mouseclock` should be called to redraw the cursor. The author knows the user can be kept happy if the cursor appears to be responsive, even if the machine is heavily loaded.

  It is important both to be fast, and to *appear* to be fast!
– `../port/tod.c:47`
  `todfix` should be called to maintain (fix!) the time of day.

4.4.2 Time handling

The clock ticks, and every tick the `tod` (time of day) module updates the system idea of the time of day.

```
main...
    consinit
         todinit()
```
Initializes time of day handling.

- `../port/devcons.c:471,475`
  The console driver init function calls `todinit` (and `randominit`). Remember that this driver was initialized during boot as every other driver configured into the system.

- `tod.c:43,48`
  `todinit` calls `fastticks` to update the `hz` field of the `tod` (time of day) structure. Perhaps a local variable instead of `tod.hz` would make it clearer that `tod` gets its `hz` when `todsetfreq` is called, and not now. `todsetfreq()` *Initialize TOD frequency.*

- `tod.c:54,60`
  `todsetfreq` is calculating the multiplier mentioned in the comment at lines `:8,24`; read that comment now.

  `todfix()` *Updates the time of day.*

- `tod.c:147,156`
  Once per tick, `todfix` gets called. The `last` variable retains its value from call to call, and is used to know if the last call was issued at least one second ago. The author does not want to do `todget` too often (to avoid wasting processor time), but he wants it to run often enough to avoid overflows in the counters used (note that there are many ticks per second). One second appears to be a reasonable compromise.

  `todget()` *Updates the time of day and returns it.*

- `tod.c:95,141`
  `todget` is doing the actual time of day updating. It is used both to update the time of day and to get the time of day. It is usual that a routine to get something can be reused to update such thing before accessing it.

- `tod.c:101,104`
  If not yet initialized, the routine initializes the module by calling `fastticks` and setting up the `hz` field. In any case, `ticks` has the value for our TSC based fast clock.

- `tod.c:105`
  `tod.last` has the time when `todget` was last called (1 second ago). Now `diff` has the number of ticks passed since then.

  Initially, `tod.last` is zero until either `todset` is called to set the time of day or `todget` runs and notices that `tod.last` is too far away in the past.

- `tod.c:108,109`
  `x` has the number of nanoseconds since the epoch time. Note how `tod.off` is added. It will be updated later.

- `tod.c:111,129`
  Only the boot processor does time of day handling. Other processors may be calling `todget` through `devcons` (`cons(3)`) or `devaudio` (`audio(3)`) to get the time of day.

- `tod.c:114,121`
  If the time of day must be adjusted, change it a bit at a time. Values `sstart` and `send` are set by `todset`. The `ilock` is used to prevent others from accessing `tod` in the mean time and also to prevent interrupts while it is being updated. `ilock` is discussed later.

- `tod.c:124,127`
  Not too often, `last` is updated to record the interval since the last call and `off` to record the time since epoch.

  Where are overflows? `last` is used to get in `diff` the number of ticks since the last adjustment. If `diff` gets too big, `x` could perhaps overflow. In fact it doesn't matter where would the actual overflow happen, what matters is that the author assumes that `diff` would never be too big and the algorithm is coded assuming that. The author is ensuring that the assumption holds.

- `tod.c:132,135`
  Time could go backward because time can be changed. In no case a time change will report an earlier time next time the user asks—that could really hurt programs that depend on the time behaving properly. Because of the same reason, the author adjusted time a bit at a time in the above lines, just to permit the user to adjust the time without big jumps into the future.

  `lasttime` holds the time reported by `gettod`. One thing is what the routine reports, and another what it believes.

- `tod.c:137,140`
  Time of day finally reported.

- `tod.c:66,89`
  Time is set by this routine. To see some call, look at `devcons.c:1211,1241`,

where the console driver accepts writes from the user to set or adjust the time of day. A `-1 t` is the convention for adjusting time a bit at a time: `todset` sets `sstart` and `ssend` (and `delta`) so that `todget` adjusts time slowly. If the time is not negative, the time is reset to the given value. `todset` can adjust the time in multiple ways. Go to `devcons.c` and try to see when is `todset` called. Correlate that with the `cons(3)` driver manual page.

`seconds()` *Returns the time of day in seconds.*

- `tod.c:158,168`
  Just to complete `tod`, `seconds` returns the time of day in seconds, and is used mostly by drivers for time outs and time stamps recorded in seconds.

4.4.3 Alarm handling

You now know how time goes by. User processes can know by reading from the console driver's time file. However, users also may want to be notified after a given amount of time. They also may want to sleep (i.e. be kept blocked and not ready to run) during a given amount of time.

- `portdat.h:79,89`
  Both `Alarms` and `Talarm` are headers for lists of processes.

- `portdat.h:75,576 and :585`
  The `Alarms` and `Talarm` lists are linked using the `palarm` and the `tlink` fields of the `Proc` structure—perhaps the names should be more uniform here. The author keeps the `Alarms` list holding all processes with alarms in the current machine. Each node (`Proc`) in the list keeps the alarm time in `alarm` and the list is kept sorted by call time. The list `Talarm` is analogous but maintains sleeping processes.

`syssleep()` *`sleep` system call.*

- `sysproc.c:478,493`
  `syssleep` calls `tsleep` to put the calling process to sleep.

`sysalarm()` *`alarm` system call.*

- `sysproc.c:495:499`
  `sysalarm` calls `procalarm` for the same purpose. In both routines, `arg[0]` is the period of time for sleeping or for the alarm. (see `alarm(3)`). Forget a bit about `tsleep` and look into `procalarm`.

`sysalarm`

  `procalarm()` *Sets up an alarm for the process.*

- `alarm.c:84,87`
  `old` set to the previously set value for the process alarm. (did you read `alarm(3)`?). The previous value is recovered by looking at time in the boot processor.

- `alarm.c:88,91`
  Canceling an alarm. `alarm` is set to zero, and it will be ignored later by `checkalarms`.

- `alarm.c:92`
  The absolute time for the alarm computed by looking at time in processor 0. Using absolute times for alarms lets the author compare the processor time with the alarm time without much arithmetic nor race conditions. It also avoids adjusting the `alarm` field as time goes by.

- `alarm.c:94,102`
  First the `Alarms` list locked, to avoid races with other processes using `alarm` or the alarm list—e.g. `alarmkproc` uses the list to notify expired alarms, and should not use the list while it is being modified. Then search any existing alarm entry for the current process. (Saw the pointer-to-pointer-to-node thing again?) Line `:98` removes the entry if it exists.

- `alarm.c:104`
  By this line, the process has no previous alarm registered in the system: not in the list, no pointer to any "next" alarm entry.

- `alarm.c:105,116`
  Alarms list not empty, the node `f` with the first alarm after the one being installed is located. That node is linked after the current process at line:109, and the current process linked in place of that node at line `:110`. If this alarm is going to be the longer one, `l` in line `:115` points to the "next" pointer in the last node, and the current process is linked there.

- `alarm.c:117,118`
  Alarms list empty, easy.

- `alarm.c:119,123`
  One way or another, the current process is now linked into the alarm list, the `alarm` time is recorded, and the list unlocked. The `goto` is used to share the code at `done`. It is breaking the loop and the conditional in a clear way. Remove the goto, and the code would become less clear.

```
trap...
  clockintr
      checkalarms()
```
Checks for expired alarms.

- `alarm.c:44`
  `checkalarms` will be called later by `clockintr`.

- `alarm.c:49,53`
  It looks at the head of the alarms list (no lock!) and calls `wakeup` if the first alarm expired. The list is not locked because the kernel still runs with interrupts disabled and because it is `checkalarms` the one removing alarms from the list. So, it is safe to look into the first node because it would not disappear under `checkalarms` feet. Can you tell know why to cancel an alarm the `alarm` field of `Proc` is set to zero?

By the way, what happens if while the alarm is pending the process dies? Hints: the alarm list points to processes; a zero `alarm` is ignored; `alarm` removes any pending alarm before installing the new one.

Ignore the rest of `checkalarms` by now.

`alarmkproc()` *Entry point for the* `alarm` *kernel process.*

- `alarm.c:12,38`
  Remember from the previous chapter that a kernel process is running `alarmkproc`? We have an endless process scanning the alarm list. The first time it entered the loop, it locked `alarms`, saw that no alarm was pending, unlocked it, and called `sleep` on `alarmmr`. So, by the time a process is setting up an alarm in the code just discussed, `alarmkproc` was probably "sleeping on `alarmmr`". The call to `wakeup` at line :53 awakes `alarmkaproc` and it continues running right after line :36. (The `return0` is a function that returns zero, but ignore that now).

  The author uses a kernel process to post alarm notes to processes with expired alarms. This process is subject to regular process scheduling and may sleep when locks cannot be gained.

- `alarm.c:18,19`
  The current time recorded in `now`, and the list locked. If the list cannot be locked because other processor is calling `alarm`, the kernel process will wait here until the lock is gained.

- `alarm.c:20`
  Pick up the head of the alarm list, and keep on scanning while the `alarm` time is past. The alarm does not happen at the exact time it was scheduled at; it may happen later.

- `alarm.c:21`
  If the alarm was canceled, `alarm` is zero and we must ignore the entry.

- `alarm.c:22`
  The lock on `debug` is needed to post the alarm note. If we cannot get it, better break the loop and go to sleep until next time `checkalarms` awakes `alarmkproc` again—yes, starvation is theoretically feasible, but so improbable that who cares?

  I hope you will appreciate that there are algorithms that are theoretically not good, but the author still prefers to use them than to modify them to keep theoreticians happy and incur into more overhead. This does not means that theory is not important; this only means that you have to balance theory with what you know from practice.

- `alarm.c:23,28`
  An `alarm` note is posted for the process, `debug` unlocked and the alarm reset (to zero). Any error during `postnote` is ignored (`poperror`!). The `alarm` kernel process better keeps on trying to post alarm notes, than die if an error happens while posting one. This is one place where `waserror` is called not to deallocate resources on errors, but to ignore errors.

```
trap...
    clockintr
        checkalarms
```

- `alarm.c:55,56`
  Back to `checkalarms`, it does by itself the processing of "sleep alarms" using the `Talarm` list. If the list is empty, nothing else has to be done.

  Can you guess why the author uses two different alarm lists?

- `alarm.c:58,73`
  For any `talarm` expired, a `wakeup` on `trend` is issued (and the process removed from the `talarm` list). When `twhen` is zero, the timer is ignored.

  This is less serious than posting a note, and I guess that is the reason why `talarms` are handled directly by `checkalarms`. Talking about reasons, there is one for having two lists (`Talarm` and `Alarms`), a process may setup an alarm and go to sleep waiting for the alarm to happen. So, two different lists have to be maintained. The `Alarms` list always has a postnote as the associated action and does not need to accept a user-supplied handler.

  `Talarm` keeps kernel timers used whenever the author wants to be notified after a while. `Alarms` keeps alarms set for user processes. More clear now?

`syssleep`
    `tsleep()` *Sets up a timer and puts the process to sleep.*

- `proc.c:525,570`
  `tsleep` is the one placing processes into the `talarm` list. A function and an argument is given. The "t" is `Talarm` is for "timer". The kernel uses `Talarms` to setup timers that call a given function when expired. After the discussion of `Alarms`, I think the code should be mostly clear. `tsleep` can be called several times for the (current) process (cf. lines `:538,548`) and the previous timer is canceled in that case. The `sleep` call in line `:567` is the one that actually makes the process sleep. I defer the discussion on `sleep`/`wakeup` until later.

`delay()` *Waits a bit.*

- `clock.c:83,100`
  To complete the discussion about timing, `delay` routines should delay for so few time that they loop to implement the delay. It would not pay to setup a timer because of the small delay time. (Well, admittedly, `delay` is called to make the kernel wait for long periods of time too; guess why?)

4.5 Scheduling

Plan 9 has preemptive scheduling, which means that from time to time processes are preempted and moved out of the processor. To now when a process should be preempted, the system clock is used. Most scheduling is done by `sched` and `runproc` in `../port/proc.c`, as we saw while learning about system startup. Interestingly, there is a `resched` function defined in `../port/portfns.h`, but does not seem to be defined; just a relic.

4.5.1 Context switching

`sched()` *Switches to another process.*

- `proc.c:91,113`
  We saw `sched` before. If you `grep` for it, you will find that it is called mostly whenever the current process should yield the processor to let another process run. In particular there are two points of interest where `sched` is called:

  - `../pc/trap.c:250,265`
    The author preempts processes here when a higher-priority process is waiting for a processor.

  - `../pc/clock.c:70,75`
    The author calls `sched` from the `clockintr` routine. Let's start here. Assume that a process is running as shown in figure 4.3 and look at that figure while reading below.

4.5.2 Context switching

```
trap
   clockintr
       sched()
```
Switches to another process.

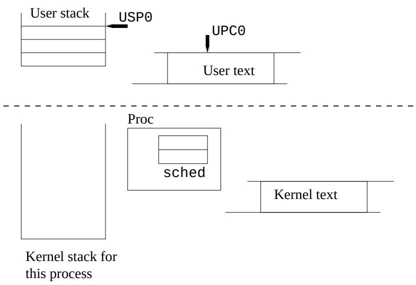
- `clock.c:70,71`
  Another clock interrupt caused a call to `clockintr`. After checking for `Alarms` and `Talarm`, if there is no current process (yet) or the state of the current process is not `Running` we return from the interrupt. If the state is not `Running`, it is likely that the process is being moved from one scheduling state to another; therefore, we better do nothing. For example, if the process is going into `sleep`, `sched` will be called by `sleep`. Routines like `clockintr` try not to interfere when the job will be done by someone else.

- `clock.c:73,74`
  If any process is ready to run, it calls `sched`. `anyready` checks the `nrdy` global in `proc.c:32`. Things look as shown in figure 4.3(b).

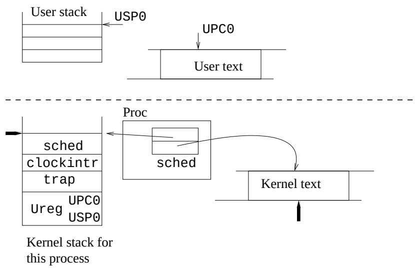
```
trap
   clockintr
       sched()
```
Switches to another process.

- `../port/proc.c:91`
  `sched` is called. Another process should run on this processor. This involves both choosing the next process and switching to it. The first task (implementing the scheduling policy) is done by `runproc`, the second task is done by `sched`.

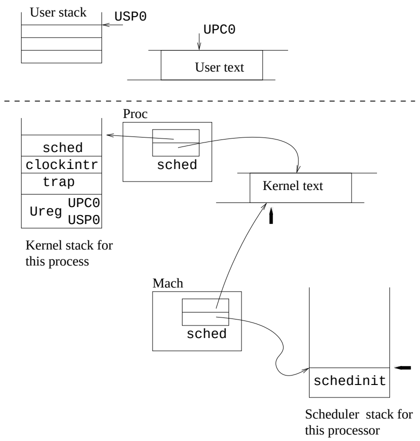
- `proc.c:93,94`
  Assume there is a current process (interrupted by the clock interrupt). We disable interrupts from now on.



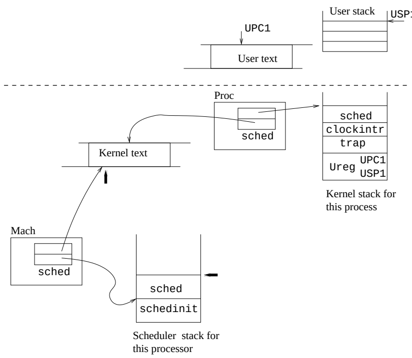
(a) The user code is running



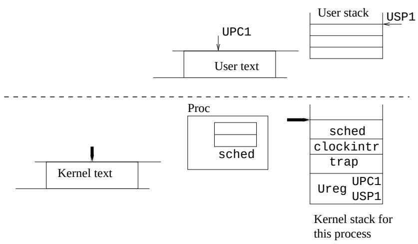
(b) An interrupt leads to `sched`. The user state is in the `Ureg`, and the kernel state is saved in `Proc.sched`.



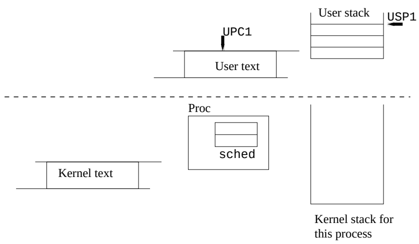
(c) Switching to the scheduler stack, using `Mach.sched`.



(d) The scheduler uses the `sched` label of another proc to switch to it.



(e) The kernel resumes within the new process context; it starts returning



(f) The new process user code runs

Figure 4.3: The kernel uses a scheduler stack to switch processes. There is one label per processor to switch to the scheduler and one label per process to remember where the process was within the kernel. Thick arrows show where the processor is running.

- `proc.c:97`
  Account the number of context switches.

- `proc.c:99`
  `procsave` saves the machine dependent part of the process. `procsave()` *Saves mach. dep. context.*

- `../pc/main.c:575`
  `procsave` starts running using the pointer to the current process as a parameter.

- `main.c:577`
  If the process used the coprocessor

- `main.c:578,579`
  If the process is dying, there is nothing to do but to reset the FPU.

- `main.c:581,589`
  But if the process is not dying, the FPU state is saved into the `fpsave` entry in the `Proc` structure. Next time we switch into this process, the state will be reloaded into the FPU. Pay attention to the comment.

- `main.c:591`
  Once saved, the process `fpstate` is inactive, meaning that there is no FPU state for this process in the real FPU. We will see more about FPU handling soon.

- `../port/proc.c:100`
  `setabel` called with `sched` for the current process. It saves the current stack pointer and program counter into label (see fig. 4.3(b)). And returns 0! (note the line `../pc/l.s:502`). The kernel stack for the current process has activation frames for `sched`, `clockintr`, `trap`, and the saved `Ureg` for the process moving out of the processor.

- `proc.c:105`
  the `gotolabel` reloads the program counter and stack pointer with the ones recorded in the `sched` label of the `Mach` structure for the current processor. If you remember from the starting up chapter, the label was set in `schedinit`.

`schedinit()` *Calls the scheduler.*

- `proc.c:58`
  And here we are!, the stack was the initial kernel stack used during boot (above the `Mach` structure), and the program counter was set to point right before line `:58`. The old process is mostly gone, although `up` still points to it—it is not null.

  Things are like in fig. 4.3(c); the scheduler stack is used to switch from one process to another.

- `proc.c:59`
  Starting to switch. We set the `proc` pointer in `Mach` to null.

- `proc.c:60,63`
  If the process is still runnable, but is being preempted, `ready` makes arrangements so it gets `Ready` to run in the future.

- `proc.c:64,79`
  If the process is dying, the state is adjusted, MMU machine dependent structures (page tables) are released (the prototype page table set up during boot will be used thereafter), and the process will be linked into the free process list. More about process death later.

- `proc.c:80,81`
  The process state is saved, so set `mach` to zero, and forget about the current process.

- `proc.c:83`
  `sched` called again, with no current process. We are still running in the scheduler stack, like shown in fig. 4.3(d). It is a good thing to have a scheduler stack. It allows procedure calls in occasions when the previous process kernel stack should not be used; i.e. to switch to a another process, the author does not need to keep on using the current stack. Besides, it is convenient when there is no current process.

`schedinit`
  `sched()` *Switches to another process*

- `proc.c:107`
  Back in `sched`, `runproc` selects another process to run. You know from the previous chapter that it loops when no ready process exists.

- `proc.c:108,110`
  The process is linked to the processor and set running.

- `proc.c:111`
  Switched to the page table for the next process. We are in the address space of the coming process, but kernel addresses are shared, so don't worry.

- `proc.c:112`
  the `gotolabel` reloads the saved kernel context (stack and PC) for the coming process. That context was probably saved at line `:100` when the new current process was last preempted. `gotolabel` reloads the the stack and PC, pretending to return 1 from the `setlabel` that filled up the label. the kernel switches to the kernel stack for the coming process. We end up as shown in figure 4.3(e).

- `proc.c:101`
  `setlabel` returned true, so `procrestore` is called to reload machine dependent processor context, interrupts are allowed again, and `sched` returns. For the PC, `procrestore` is defined to do nothing (`../pc/fns.h:98`). Probably, the `setlabel` for the current process was made while running `sched`, called from `clockintr`, called from `trap`, so the return starts the unwinding of the kernel stack. When `trap` finally returns, the `IRET` in `l.s` will reload the saved `Ureg` for the current process—see fig. 4.3(f).

One more note: the state of the new current process could be other than sown in this example execution. In general, that process could have a kernel stack corresponding to any path of execution leading to a call to `sched`. You will see more examples of that during this chapter.

4.5.3 FPU context switch

You saw how `procsave` and `procrestore` were used to save and restore the FPU state. But how is the FPU context really handled? `mathinit()` *Initializes FPU traps/interrupts.*

- `../pc/main.c:552,559`
  FPU traps were initially enabled by `main`. `procsetup()` *Initialize FPU for a new process.*

- `main.c:565,569`
  Initially, the `fpstate` is set to `FPinit`, as you saw in the previous chapter, and the FPU set to an offline state. `mathemu()` *Services the FPU emulation fault.*

- `main.c:517,520`
  If the process uses the FPU, it will get an emulation fault, because the FPU was set off. `mathemu` calls `fpinit` to initialize the FPU (enable it) and sets the `fpstate` to `FPactive` (because the process is known to use the FPU).

- `../port/proc.c:99`
  If a new process is getting switched in, the FPU state for the previous process (which used the FPU in our example) is saved...

- `../pc/main.c:575,592`
  because it was `FPactive`. Its FPU state is now `FPinactive`.

  If the next process does not use the FPU, its state will be `FPinactive`.

  When the first process (that used the FPU) is switched back again, `procrestore` does nothing!.

- `main.c:521,535`
  When the process starts using the FPU again, another FPU fault leads to `mathemu`. As it is `FPinactive`, its FPU context was saved and must be reloaded. Lines `:533,534` do that. The process is now `FPactive`, as it was before the first time it was preempted in our example. I hope you will see how the author avoids switching the FPU context when the process does not use the FPU.

  Some other OSes try to avoid even saving the FPU state, by keeping track of who did use the FPU last and saving that state only when it is absolutely necessary—i.e. before another process uses it. But, take into account that Plan 9 runs on MP, and the FPU state might be loaded into a different processor the next time. Things are more simple in the way they are in the code, and fast enough.

4.5.4 The scheduler

How does `runproc` select the next process to run? It applies a scheduling policy to select a process. Let's look at it. But we should start by the routine allowing processes to run.

Getting ready

- `../port/proc.c:24,33`
  Processes willing to run are either `Running`, or they are `Ready` to run. Ready processes are linked into `Nrq` scheduler queues. Each queue has processes of a given priority, and high-priority processes get more CPU than low-priority ones. Unlike UNIX[1], Plan 9 uses higher priority values for higher priorities!

`ready()` *Puts a process in the ready queue and recalculates priority.*

- `../port/proc.c:142`
  Before running, the process must be set `Ready` and linked into a ready queue—so that later `runproc` can pick it up.

- `proc.c:147`
  With interrupts disabled (because nobody should touch the scheduler queues),

- `proc.c:150,153`
  if the process was running (is being preempted), `rt` is incremented. `rt` counts the "running time" for the process measured in quanta. Every time the process goes from `Running` to `Ready`, `rt` is incremented. So `rt` measures how many *full* quanta the process just consumed. `pri` is set to the formula:

$$\frac{\frac{\mathtt{art}+2\mathtt{rt}}{4}}{\mathtt{Squantum}}$$

- `proc.c:153,157`
  if the process was not running (it is not being preempted), the average running time, `art`, is set to $\frac{\mathtt{art}+2\mathtt{rt}}{4}$, and `pri` is set to

$$\frac{\frac{\mathtt{art}+2\mathtt{rt}}{4}}{\mathtt{Squantum}}$$

  —`rt` is reset to zero. Wait a bit to understand what is going on and keep on reading.

  `Squantum` is set to $\frac{\mathtt{HZ}+\mathtt{Nrq}-1}{\mathtt{Nrq}}$ in `proc.c:138`. If the number of run queues is small with respect to the machine HZ (ticks per second), `Squantum` is almost $\frac{\mathtt{HZ}}{\mathtt{Nrq}}$, dividing the `HZ` evenly among the run queues. If `Nrq` is big regarding `HZ`, `Squantum` is almost $\frac{\mathtt{Nrq}}{\mathtt{Nrq}}$. But this formula is empirical and only the author knows how it was adjusted to yield the current expression. For our PC, `HZ` is 82 and `Nrq` is 20, yielding a `Squantum` of 5. To augment `pri` in one, `rt` has to be incremented by 5/2, or `art` has to be incremented by 5.

[1] In UNIX, priority 0 is better than priority 10.

So, every time the process gets `Ready`, (i.e. is preempted), `pri` increases slowly as `rt` and `art` increase. `rt` influences more `pri` than `art` does. Things can change because if the process leaves the processor voluntarily (i.e. gets blocked, and after a while it goes from a non-ready state to `Ready`) its `rt` is set to zero, and its `art` gets decreased. The order of lines `:154,157` is ensuring that changed values are used next time and not now.

- `proc.c:158,160`
  Here it is, `pri` is set to `basepri` minus the `pri` value just computed. If the computed `pri` was small, the process priority would be close to `basepri`; otherwise it can go all way down to lower values, or even zero. So, for the process, it is bad when the computed `pri` gets big. Should the process not block, neither `rt` nor `art` would be decreased, so the computed `pri` gets bigger and `basepri-pri` would be smaller; should the process block, `rt` will be reset, and `art` will be decreased so that the computed `pri` gets smaller and `basepri-pri` would get closer to `basepri`. If the same process keeps on blocking (e.g. gets `Running`, computes a bit, reads a file and blocks, gets unblocked, and so on), its final `pri` will end up being `basepri`. If the process keeps on running, its final `pri` will be very low. Interactive processes tend to block, and they are not penalized; CPU intensive processes are penalized. By the way, do not pay much attention to the exact details of the formulas, other similar ones are likely to work too; these things come out of the author's experience with the system: they are empirical.

  Did I already said that these things are empirical? Don't forget.

- `proc.c:162,170`
  If the process `basepri` was above `PriNormal`, avoid the priority decreasing too much. If the process is waiting for a lock, its priority would be just `PriLock`. Otherwise, its priority is the `pri` just computed.

- `portdat.h:522,525`
  `PriLock` is 0, a very low priority value. If the process is waiting to gain a lock, it is not doing anything useful (yet), so the system penalizes it. It is better to let the process holding the lock run, and penalizing ourselves is a fine way of favoring that. Besides, note that `PriKproc` (`basepri` for kernel processes) and `PriRoot` (`basepri` for processes running `/*` files) are above `PriNormal`. That means that both kernel processes and "root processes" will be kept above `priNormal`, no matter what they do. The system gives them priority over normal process, that are below `PriNormal`. Processes in both priority classes (above and below `PriNormal`) get their priorities adjusted during time, but they stay within the same class.

  Both root processes and kernel processes are working on behalf of the whole system. It makes sense to give them priority because otherwise the whole system would suffer. Try to find out which ones are the kernel and root processes in your Plan 9 box (hints: how are you using your local disk files? how are you talking to other nodes in the network? how are you using your swap file? did you look at the `mkfile`?)

- `proc.c:171`

There is a run queue for each priority level. Set `rq` to be the queue for the process `priority` this time.

- `proc.c:173,184`
  The process is set `Ready` and linked into its priority queue. `readytime` is set to the time the process was set `Ready`.

- `proc.c:185`
  If interrupts were enabled at line `:147`, they are enabled now; otherwise they stay disabled. The reason for using `splx` is that `ready` must both lock the run queue and disable interrupts; as `ready` can be called either with interrupts enabled or disabled, the author restores things as they were.

In few words, you now know that Plan 9 uses dynamic priorities within two priority classes.

Picking up a process

Finally, `sched` calls `runproc` to pick up a process to run. You already read `runproc` in the previous chapter, but let's look at some details now.

`sched`
  `runproc()` *Elects a process for running.*

- `proc.c:204`
  Once out of four times, `runproc` forgets about processor affinity and priority, and picks up the process waiting longer—trying to avoid starvation and to balance the load of processors here.

- `proc.c:211,219`
  Low priority queues are scanned first! `xrq` points to the run queue with the minimum `readytime` found for the head process, and `rt` is set to the minimum `readytime` for that process. By line `:219` the lowest priority process sitting at the head of a run queue that was ready before the other ones is located by `xrq`. Just the head is used, to avoid locking the ready queues while scanning for processes.

- `proc.c:220,226`
  If there is such a process, `rq` is set to the queue where it stands, and `p` to the process selected. The `goto` goes to the place where `runproc` tries to run it. If there is no such process, `runproc` loops again; but next time it will honor priorities and affinity. If the process is not `wired` to the processor, `movetime` is set to zero, that is a really small value and will allow the process to move.

- `proc.c:232,241`
  Three out of four times, run queues are scanned from high to low priority; as they should. If there are no processes, `runproc` keeps on looping. The process chosen is the first one that either

  - is the first in the queue that reached its `movetime` (did not moved recently; same reasons as above), or

  - is the first in the queue that did run on this processor (to keep processes running within their favorite processors).

  By the way, the author didn't really choose a process, but a run queue instead!

  The queues were not locked. But the worst thing that may happen is that the process gets removed from the ready queue (by another processor) and `runproc` will find its `rnext` pointer to be nil. Since `Procs` are allocated statically in a big process table, there are no worries about crossing a dangling pointer.

- `proc.c:245`
  got a process.

- `proc.c:246,248`
  Someone is using the queues (more than one processor), try again.

- `proc.c:250,255`
  Now a process is chosen, but using the `rq` selected before. If things have changed, the fortunate one may be different from the process than caused this ready queue to be selected—this is the price for avoiding locks before. `l` is set to the process before the one selected, and `p` to the first one with affinity for this processor: the selected one.

- `proc.c:260,263`
  No process selected, try again. If the reason was that no process had affinity for the processor, in a couple of loops `runproc` will ignore this fact.

- `proc.c:264,271`
  The process removed from the ready queue. It is no longer ready but running. `l` is used to remove it from the list. Counters adjusted accordingly.

  The process `rnext` is not cleared!, if any other processor is scanning through the ready queues, it could still jump over to the previously next process in the ready queue, even though the process is not ready. The author is careful to avoid unnecessary locks in places where locking would mean severe performance degradation.

- `proc.c:272,273`
  A scheduling bug?

- `proc.c:276,280`
  And there it goes. `sched` will switch now to it, as we saw before.

Processes tend to be chosen from the head of the list, and are inserted at the tail in `ready`. The effect would be a round-robin for processes with the same priority—if you forget about affinity or if there is only one processor.

There is one thing to consider. What if a high-priority process was blocked, and it suddenly gets `Ready`? That can happen because we are running a low priority process and an interrupt notified the kernel that the event the process was waiting for, just happened. The answer to the question is in `trap`.

`trap`

- `../pc/trap.c:255,265`
  Conditions say that: it is an interrupt but not the clock or a timer; there is a currently running process, and there are higher priority processes waiting. Interrupts happen often, but clock and timer interrupts happen really often. Checking for a higher priority process when a frequent (but not ubiquitous) interrupt happens is a way of checking often (but not always!) for a high priority process. The `preempted` flag is set when the author commits to preempting the current process in favor of the higher priority one, the check at line `:258` ensures that this code would be ignored if the process is already being preempted. A simple call to `sched` ensures that the high-priority process will be able to run.

  `sched` will not return before the other process runs. When it returns (the current process has been switched back to the processor), `preempted` will be reset, and another interrupt may yield to a new preemption.

  The `slphi` at line `:262` is problematic, because the new interrupt might cause a preemption before doing a return-from-interrupt for the current one. Think that each interrupt pushes more frames into the kernel stack, and they are popped only when returning from the interrupt. If enough interrupts arrive, the kernel stack would overflow and the system would probably crash. But the `preemted` check seems to suffice to avoid that.

You know affinity, but processes can be also wired to a processor.

`procwired()` *Wires a process to a processor.*

- `proc.c:354,383`
  `procwired` wires `p` to any processor (if `bm` is less than zero), or to the processor specified by `bm` otherwise. Most of the routine is picking up a processor to wire the process to. The one with less wired processes is used. Perhaps a new `m->nwired` field would save most of this code. A big `movetime` is given to the process so it never(?) moves.

Finally, I don't discuss it, but `accounttime` in `../port/proc.c:1210,1233` maintains values for processor load averages and process run times.

4.6 Locking

During the description of what the code is doing, you saw lots of locks and locking routines. I skipped all of them. Now it's time to discuss locking one you know about process scheduling and process priorities.

Because Plan 9 runs on MP machines, there are several locking primitives employed. Let's see them from the most simple to the most complex.

4.6.1 Disabling interrupts

Remember that within the kernel `setlabel` and `gotolabel` are used to provide coroutines. Therefore, within the kernel, the kernel decides when to leave the processor by using `gotolabel` in favor of another kernel routine. So, you could say that ongoing system calls for other processes are not an issue regarding mutual exclusion for critical regions within the kernel.

What? You don't understand the meaning of "mutual exclusion" or "critical region" (go, reread the material for the OS course you attended several years ago and come back later).

However, while the kernel is executing a routine in favor of a user process, an interrupt may arise. The interrupt will start a different routine of the kernel while the previous one is stopped. If a system call is being executed and an interrupt arrives, the interrupt routine can try to access resources used by the previously ongoing system call. Therefore, the kernel must prevent interrupts from happening while touching data structures that can be manipulated by the code executing after the interrupt. If you take into account that an interrupt can lead to the suspension of an ongoing system call and a context switch to another process, you can imagine that code executing after the interrupt can touch almost any data structure in the kernel.

According to what I just said, if there is a single processor, disabling interrupts would ensure mutual exclusion among processes while executing within the kernel. While touching an important data structure, the kernel can disable interrupts and it knows nothing will "preempt" it in the meanwhile. There can be more than one processor, but even so, there are structures that are handled (read: written) only by one processor (e.g. `Mach` for each processor) and can be protected by disabling interrupts. Other critical regions, like the code doing a context switch for a process, are also protected this way. How are interrupts enabled and disabled?

The abstraction used to enable/disable interrupts is the "processor priority level". Imagine that the processor is running at a given priority (0 or 1). While running at a low priority, interrupts (high-priority events) can interrupt the processor. While running at high priority, interrupts cannot interrupt the processor. This comes from the days UNIX was implemented because the processor used actually worked this way.

`spllo()` *Sets processor priority low.*

- `../pc/l.s:433,437`
  `spllo` (set priority level low) is used to enable interrupts. It pushes the flags word in the stack and pops it back into `ax` (the return value of `spllo`). The interesting part is `sti`, which sets the "interrupt enable" bit in the flags word. The author uses the stack to move `flags` into `ax` because the only way `flags` can be accessed on the Intel is by pushing/popping it on/from the stack. The value returned by `spllo` can be used to restore the previous "processor priority level" (i.e. the interrupt enable bit) as it was before the `spllo`.

`splhi()` *Sets processor priority high.*

- `../pc/l.s:423,431`
  `splhi` (set priority level high) is used to disable interrupts. Lines `:424,426` set in `m->splpc` the return PC as saved in the stack by the call to `splhi`. That is, `m->splpc` holds the PC of the instruction that called `splhi`; That is for profiling, but can be used for debugging too. If somehow interrupts are disabled and they shouldn't be, you could know who is guilty for that. The real work is done

at line `:430`: interrupt enable cleared. The old value of flags is returned as in `spllo`.

`splx()` *Sets processor priority as given.*

- `../pc/l.s:439,448`
  `splx` (beware that it continues until the `ret`), exchanges the flags and the value passed as a parameter. If the kernel calls `spllo` and, later, passes the value it returns to `splx`, flags would be restored as they were; i.e. the priority level would be restored. Lines `:440,442` are saving the caller's PC in `m->splpc` for the same reason as above. Line `:445` is taking the first parameter (FP is the frame pointer).

- `../pc/l.s:450,451`
  `spldone` is not used as a function. If `spllo` or `splx` is executing, the PC is between `spllo` and `spldone`. If you look at `../port/devkprof.c:38,43` you will see how are `m->splpc` and `spldone` used. The routine `_kproftimer` maintains statistics about what parts of the kernel execute during what times, the author seems to want to charge the `spl` times to the caller and not to the routines themselves.

4.6.2 Test and set locks

By using an atomic `tas` instruction, which tests for the value of a word and sets it to true value, mutual exclusion can be achieved even with interrupts enabled. That is important because it is overlay expensive to disable interrupts on all processors—and it is also complex. So, kernel routines can agree that when a `lock` word has a true value (non-zero), the critical region cannot be entered. By using an *atomic* `tas`, the previous value of the lock can be tested and set without race conditions. The first one to set the lock, acquires it. Another useful variant of `tas` is `xchg`, which exchanges a register with a memory position atomically.

`tas()` *Tests and sets a word.*

- `../pc/l.s:462,466`
  The Intel only has `xchg`, so `tas` uses the intel instruction to emulate a `tas`. The parameter passed is the lock being tested and set. The value set (`0xdeaddead`) is irrelevant. Line `:465` is atomic!

`xchgw()` *Exchanges a two words.*

- `l.s:472,476`
  `xchgw` is a wrapper for the Intel instruction `xchg`. Two parameters passed are the "register" and the memory position being exchanged atomically. By the way, `xchgw` seems to be used only the `astar` device, while `tas` is the routine actually used for mutual exclusion by the kernel. So, if `astar` had another way of doing its business, `xchgw` could go away—as seemed to happen with `xchgl`.

The kernel does not use `tas` directly, but uses routines provided by `taslock.c` instead. With the help of several machine dependent routines, most of test-and-set code is kept portable.

`lock()` *Acquires a test-and-set lock.*

- `../port/taslock.c:31`
  `lock` acquires a lock using `tas`. The `Lock` structure is defined in `../pc/dat.h:24,31`; you will see how it works now. `getcallerpc()` *Gets the PC after the instruction that called it.*

- `../port/taslock.c:36`
  `getcallerpc` uses the address of the `lock` parameter to locate the PC of the caller. If you now the address of the first parameter, you can know where in the stack is the return PC and obtain that value. The PC of the caller is stored in `Lock.pc` for debugging purposes. If a lock has problems, it is useful to know who did set the lock.

- `taslock.c:39`
  Here it is. `tas` tries to set the lock, which is actually `Lock.key`. If the lock was cleared before `tas` executed, the return value would be zero; otherwise, the return value shows that the lock is already held by someone else.

- `taslock.c:40,43`
  The `Lock` structure is updated to record the process setting the lock and the PC that called `lock`; all done. `isilock` is a boolean stating that the lock was set by `ilock` and not by `lock`. More about this soon. Most of the time, the lock would be not set and the kernel would execute these lines. If the lock is found to be set most of the times, that would show contention for a given lock within the kernel. The affected data structure would better split in several ones, or some of its parts locked separately, and perhaps a different kind of lock used (Noticed that `Proc` has several locks?).

- `taslock.c:46`
  `glare` counts how many times the lock couldn't be acquired at the first attempt.

- `taslock.c:47`
  Trying to get the lock. If there is a current process and it is running (i.e. the `lock` is not requested once the process was committed to block) the kernel can call the scheduler to wait for a while until the lock be released.

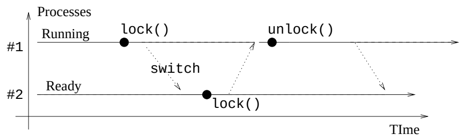
- `taslock.c:48,53`
  If the scheduler can be called, save the process priority so it can be restored later, and set the priority to `PriLock` (i.e. to zero, so that almost every other process would be preferred by the scheduler). If the process priority is kept as it is, and the holder of the lock has a lower priority, the scheduler could prefer to switch back to the current process instead; So, the author is making sure that such thing would not happen. `lockwait` is set in `Proc` to state that the process is waiting for a lock; the scheduler would maintain the priority set to `PriLock` no matter what the process does.

- `taslock.c:55,56`
  Now waiting while trying to get the lock repeatedly. The number of "attempts" is recorded in `inglare`. When contention for locks appear, the author can at

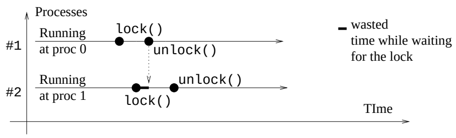
least now that in mean, `inglare`/`glare` loops are needed. If the relation gets too big, it can be a symptom that locking has to be adjusted.

- `taslock.c:58`
  Important loop! If there are several processors, the current one has `l->key` on its cache. If the processor does not write the lock, it will not interfere other processors because the lock value will be read from the cache. If the processor would `tas` the lock instead, it could lead to a high bus contention—because the value would be written and that usually means that the value has to be put into main memory, or that the processors must talk to update their caches. So, do not even try to set the lock until we know it is no longer set.

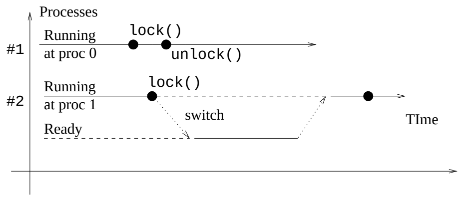
- `taslock.c:59,65`
  Just one processor, and can call the scheduler. Do so (see figure 4.4(a)). The `i` counter is used to report we are "looping many times" once per thousand iterations.



(a) test-and-set lock on uniprocessors. It is better to context switch if the lock cannot be acquired.



(b) test-and-set lock on MP. It is better to wait for a while.



(c) test-and-set lock on MP in case a context switch was made. The time used for context switching at processor one is wasted time, since no user process is running in the mean time.

Figure 4.4: Behavior of test-and-set locks.

- `taslock.c:65,70`
  More than one processor. It is better to wait for a while until the lock is released by another processor (see figure 4.4(b)). `tas` locks should be held only during small amounts of time, so it would not pay to do a context switch (see figure 4.4(c)). In the case of a monoprocessor, it would be a waste to keep on looping because unless we release the processor the lock holder would not run. Note the high number of iterations until reporting that we have problems.

- `taslock.c:72`
  Out of the while, so the lock was released when we last checked it in line `:58`. Now try to set it.

- `taslock.c:73,80`
  If got the lock, update the `Lock` structure and restore the previous priority in case it was set to `PriLock`. If did not get the lock try it again—note that no `tas` will be tried until the "lock-read-only" while finishes.

`canlock()` *Tries to acquire a test-and-set lock.*

- `taslock.c:136,145`
  `canlock` tries to set the lock but just once. It lets the caller know whether the lock was acquired or not. Routines willing to give up or to do something else when a lock cannot be acquired can use `canlock`. You already saw how `canlock` was used by the scheduler to give up and try again. The implementation of `canlock` uses `tas` as `lock` does, and initializes the `Lock` structure appropriately.

`unlock()` *Releases a test-and-set lock.*

- `taslock.c:148,158`
  Releasing a `tas` lock is easy, just set the lock (`l->key`) to zero and the next `tas` will return zero. Note how the lock is checked to be set. The call to `coherence` ensures that the lock value is seen by other processors—a "no-op" on Intels.

`ilock()` *Interrupt-safe version of `lock`.*

- `taslock.c:85,86`
  `ilock` is a variant of `lock`. The `lock` routine works fine when it comes to mutual exclusion between different processors. However, the kernel may also want mutual exclusion between a process running inside the kernel (e.g. a system call) and an interrupt handler. Imagine that the kernel gets a lock on the memory allocator, and an interrupt arrives. If the interrupt handler wants to allocate memory, it would try to acquire the memory allocator lock. You get a deadlock! This is shown in figure 4.5.

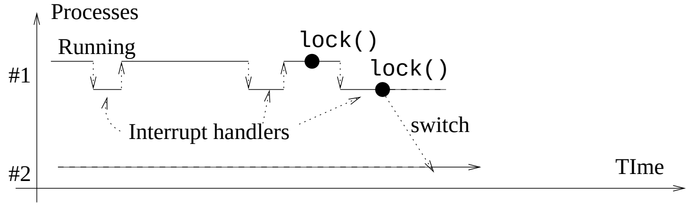


Figure 4.5: Acquiring locks without `ilock`. The process 1 acquires a lock, and gets interrupted. The interrupt handler tries to acquire the lock and has to block!. The lock will never be released.

The way to avoid the deadlock is to disable interrupts besides acquiring the lock. By disabling interrupts, no interrupt handler can request the lock because no interrupt handler will run (in the current processor) while the lock is held. Other processors are not an issue because they can try to acquire the lock without a deadlock.

The author could disable interrupts and then call `lock`, but if the lock was not acquired, interrupts might be cleared for just too long. The `ilock` routine tries to acquire the lock and restores the previous interrupt state when the lock cannot be acquired.

- `taslock.c:95`
  Interrupts disabled, the previous priority level kept at `x`.

- `taslock.c:96,102`
  If `tas` got the lock, return with interrupts disabled and the lock set. (`isilock` records that the lock disabled interrupts too). The previous priority level is saved in `l->sr`. The unlock routine needs that to restore interrupts to their previous state.

- `taslock.c:112,113`
  A single processor available, how could the lock be released if interrupts are disabled? This message would mean that usage of locking primitives has to be fixed.

- `taslock.c:117,120`
  While the lock is held by another process, restore the interrupt status. Once the lock is released, interrupts are cleared before trying again to acquire the lock.

`iunlock()` *Interrupt-safe version of* `unlock`*.*

- `taslock.c:161,177`
  Locks acquired with `ilock` are released with `iunlock`. Again, the author ensures that it is an `ilock`ed lock. The last two lines are manually setting `splpc` to be the PC of the caller of `iunlock`, and then doing the rest of `splx` (remember that `splx` did fall down to `splxpc`?).

4.6.3 Queuing locks

A `Lock` can be used to protect small critical regions. When the lock is being held for a long time, or when there is much contention on a lock, another kind of lock is needed. If processes would have to wait for a long time to acquire a lock, they better sleep while waiting. When there is much contention for a lock, it is also better to respect the arrival order of the lock requesters; otherwise you can get starvation.

- `../port/portdat.h:60,66`
  A `Qlock` is a lock that maintains a queue of processes waiting for the lock. The `head` and `tail` fields maintain the list, the `locked` flag shows whether the lock is held or not. And finally, the `Qlock` structure has to be protected for race conditions: `use` is a `Lock` that must be held to operate on the `QLock`. `use` will be held during a small amount of time—as soon as the process either gets the `Qlock` or gets queued, the `use` lock is released. The author is building locks appropriate for big critical regions and high contention using the simple `tas` locks as the building block.

`qlock()` *Acquires a queuing lock.*

- `qlock.c:17`
  `qlock` is the routine called to acquire a `QLock`.

- `qlock.c:21`
  Important! to protect `q`, a `tas` lock is acquired.

- `qlock.c:23,27`
  If the `q` lock is not set, set it and return. In this case the `tas` lock is released; it was held only while the `QLock` was manipulated.

- `qlock.c:28`
  The number of processes queued on a `QLock` increased. The author can know whether the queues of `QLocks` are really used or not. A kernel is a living thing, the only way to issue a good diagnostic is by asking it about its symptoms.

- `qlock.c:29,38`
  The process is queued in the `QLock`. There *must* be a current process, otherwise there is nothing to queue. See how the process is queued in the tail of the queue?

- `qlock.c:39`
  The process was not `Ready`, now it is `Queueing` and will not run again until extracted from the lock queue and placed back in a ready queue.

- `qlock.c:40`
  The PC of the caller recorded to find out guilties for locking bugs.

- `qlock.c:41,42`
  The `QLock` was protected by the `use` lock during all this time; now release the `tas` lock and switch to a different process. The current process is queued waiting for the lock.

`canqlock()` *Tries to acquire a queuing lock.*

- `qlock.c:45,57`
  Like `canlock`, `canqlock` tries to acquire the lock and returns reporting whether the lock was acquired or not. `canlock` is used on the `use` lock (because `canqlock` should not block).

`qunlock()` *Releases a queuing lock.*

- `qlock.c:59,76`
  `qunlock` releases a `QLock`. If there are processes in the queue, the one at the head is extracted and set `Ready`. That process has the lock (Hence `locked` is kept set to true). If there is no process waiting for the lock `locked` is set to false. `QLocks` are acquired in FIFO order regarding the request time.

There are many places where `QLocks` are used. They are used wherever the lock is likely to be held for a long time—and processes are likely to wait for the lock a long time. For instance, I/O devices like `cons`, `pipe`, etc. use `QLocks` because I/O operations are likely to be slow and take a significant amount of time.

4.6.4 Read/write locks

The locks discussed above could suffice. However, many times there are several processes accessing a data structure just for reading it, and there are several other processes writing the data structure. It makes no sense to *serialize* the readers of the data structure. When there is a single processor, readers would necessarily read the data structure once at a time; but on multiprocessors, several processors could be reading the same data structure at the same time without any problem. With the locks seen before, multiple processors willing to read the same data structure could be stalled, waiting for another reader to finish. That is why there are read/write locks. While you read below, see figure 4.6 as an example of how R/W locks would block readers/writers. The figure should be clear by the end of this section.

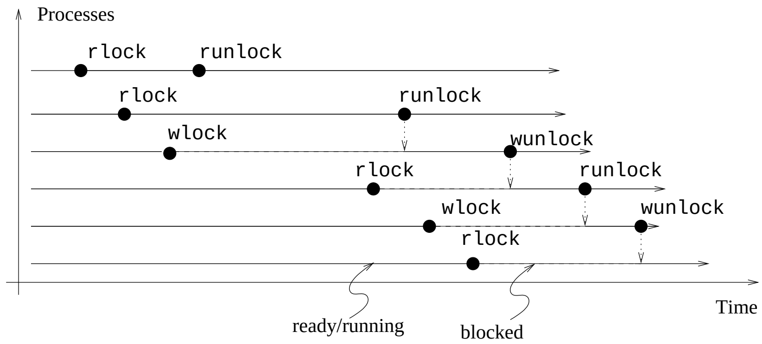


Figure 4.6: Read/Write locks. Typical scenario showing how readers/writers acquire the lock.

`rlock()` *Acquires a R/W lock for reading.*

- `qlock.c:79`
  `rlock` tries to acquire a `RWLock` for a reader process. If will be able to acquire the lock if there is no one writing the locked data structure—or waiting to write.

- `qlock.c:83`
  Again, using a `tas` lock to protect the lock data structure.

- `qlock.c:84`
  Now it is important to know an overall number of readers and writers for `RWLocks`; that can reveal symptoms that `RWLocks` are used where a more simple lock could be used instead.

- `qlock.c:85,90`
  `writer` is a counter for the number of writers. Perhaps it should have been named `writers`, but there is only one writer allowed at a time, hence the name. So, if there is no writer, and there is no process waiting for the lock, the lock is acquired. When there is no writer, but there are processes waiting for the lock, that processes would be writers waiting to acquire the lock. In this case, it is

important to queue and give priority to the writers who arrived before. Otherwise, a continuous flow of readers might starve a writer. `readers` is incremented to reflect that there is one more reader holding this lock. There is no "`locked`" field in the `RWlock`; if there are readers or writers, the lock may be locked or not depending on who you are and who is holding the lock.

- `qlock.c:92`
  One more reader had to wait, let the author know.

- `qlock.c:93,102`
  The process (and there must be one) is queued in the list of processes queueing at the lock. The queue is maintained using the `qnext` field of the `Proc` structure.

- `qlock.c:103,105`
  The process state shows that the queued process is queueing for reading. As with `QLocks`, the process will not run until dequeued from the lock queue and placed back in a ready queue.

`runlock()` *Releases a R/W lock for reading.*

- `qlock.c:109`
  Readers use `runlock` to release a `RWlock` for a reader.

- `qlock.c:115`
  If there are more readers holding the lock besides the one releasing the lock, nothing else has to be done: the lock is still held by remaining readers. Besides, if there is no process waiting to acquire the lock, there is nothing more to do because by updating `readers`, the lock is released when `readers` gets down to zero.

- `qlock.c:120,122`
  The last reader went away and there are processes waiting. The first process waiting in the queue must be a writer—note that should it have been a reader, it would have entered acquiring the lock, because there was no writer waiting by that time. There can be readers in the queue too, but they are queued because they saw there was another process queued (and a writer was among them) and decided not to pass it. The process state is checked to see whether the queued process is waiting for read or for write—see figure 4.7.

- `qlock.c:123,128`
  As the last reader is gone, awake the writer by removing it from the queue and setting it ready. The `writer` counter is set to one to reflect that one writer holds the lock. That would prevent further readers/writers to acquire the lock. Now that `p` is ready, the scheduler can elect it for running.

`wlock()` *Acquires a R/W lock for writing.*

- `qlock.c:132`
  Writers use `wlock` to acquire the `RWlock` for writing.

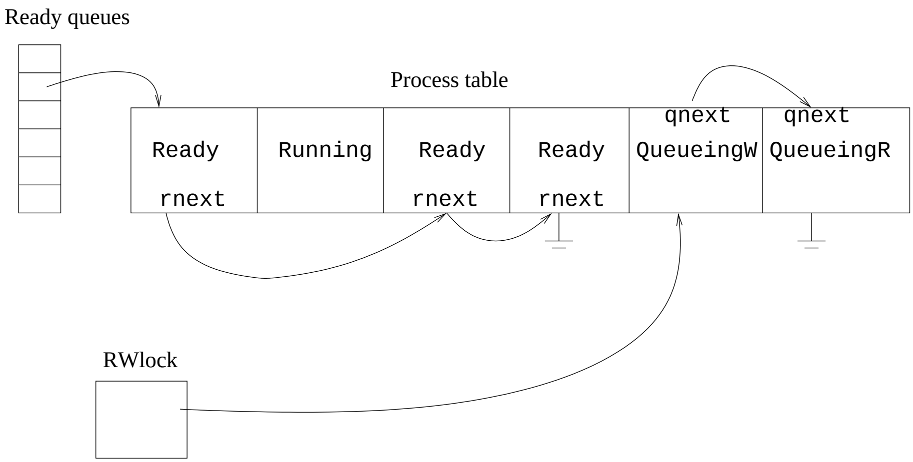


Figure 4.7: Queueing locks. The process is just removed from the ready queue while waiting to acquire the lock. In this case, the lock is a read/write lock and the process state is used to see what the process is waiting for.

- `qlock.c:138,145`
  If there are no readers and there are no writers (no writer, actually) the lock can be acquired. The PC that called `wlock` and the process that acquired the lock are noted, for debugging. For readers, the author thought it was not worth to record that information, probably because there are multiple readers.

- `qlock.c:148,158`
  The process must wait until either the last reader releases the lock, or the current writer releases the lock. The process state is updated to reflect "waiting to acquire a `RWlock`" for writing, and the scheduler is called. The process won't run again until dequeued by either the last reader or by the writer. Even if the lock is held by readers, further `rlocks` will have to wait.

`wunlock()` *Releases a R/W lock for writing.*

- `qlock.c:165`
  Writers releasing the lock call `wunlock`.

- `qlock.c:169,175`
  If there is no process waiting, the lock is released by setting `writer` to zero.

- `qlock.c:176,184`
  If the first process waiting is a writer, it is given the lock (note that `writer` is kept set to 1). The writer is allowed to run by removing it from the lock queue and setting it ready.

- `qlock.c:186,187`
  The process must be one waiting to acquire the lock for reading. Otherwise,

some bug caused other process to enter the queue, or some bug caused the waiting process to change its state.

- `qlock.c:189,197`
  The first process waiting is a reader, but as the lock is going to be acquired by a reader, any other reader waiting can acquire the lock at the same time too. The author scans the queue seeking for processes queueing for read. All of them are removed from the queue and set ready. `readers` is updated to reflect that the lock is held by that many readers. The scanning stops as soon as a writer process is found, because remaining readers should yield to the writer who arrived before.

- `qlock.c:198,199`
  It did not harm that `writer` was set until now, because the `tas` lock was held. But better update it now.

`canrlock()` *Tries to acquire a R/W lock for reading.*

- `qlock.c:202,216`
  `canrlock` is used as `canlock`, but for a `RWlock` locked for reading. Perhaps a `canwlock` could be added for completeness, although no one is using such a thing now.

One comment before proceeding. When a process is being awakened to acquire the lock, you saw how the author does the bookkeeping for the awakened process (updating counters et al.) in the process holding the `tas` lock. You will also see that when author implements note posting and notifying, whatever can be done easily within the notifying process is not done by the notified process. This is a general rule that when you hold a lock and some bookkeeping has to be done for the process being awakened, you better do it in the awakening process before releasing the lock. Otherwise, the awakened one would have *necessarily* to acquire the lock for the resource and that could lead to more race conditions. Keep the code simple. Of course there is a counterpart rule that if you can do something more easily in the process awaken than it can be done in the awakening process, you should do it where it is more simple. Keep the code simple, did I say that?

4.7 Synchronization

There are several forms of synchronization in Plan 9. User processes, to synchronize, can use `rendezvous(2)`.

Besides, the `OCHEXCL` bit for permissions given to the `create(2)` system call, allow processes to synchronize on their access to files: only a process may have the file open at a time. This simple mechanism avoids the need for complex file locking primitives found on other systems like UNIX. This is very important since there is distributed access to files (noticed the `nfslockd` process on your UNIX box?).

Moreover, the `CHAPPEND` permission bit, together with the guarantee that small writes are likely to be serviced atomically by the file server, can be used together to add new data to a file without race conditions.

While in the kernel, processes `sleep` waiting for an event and `wakeup` other processes, as you saw while discussing timers; and of course, you have the various lock primitives discussed before.

In this section you are going to read the code for `redezvous`, `sleep`, and `wakeup`.

4.7.1 Rendezvous

See `rendezvous(2)` before continuing.

`sysrendezvous()` *rendezvous system call.*

- `../port/sysproc.c:696`
  `sysrendezvous` is called with two arguments

- `sysproc.c:702`
  `tag` is `arg[0]` and `val` is `arg[1]`.

- `portdat.h:407,411`
  A `Rgrp` (rendezvous group) is a hash table with `RENDHASH` entries.

- `sysproc.c:703`
  `REND` is defined in `portat.h:393` to use the hash function to locate the entry in the table. The author just applies the modulus to the `tag`, that seems to be a good enough hash function for the case. `l` is our entry in the hash table.

- `sysproc.c:706`
  The list in the hash bucket is searched for a process with the same `tag`. You see how `Procs` doing `rendezvous` are linked into the `Rgrp` hash table using the `rendhash` field as the "next" pointer. The code is clear, but perhaps names like "`rqnext`", "`rendnext`", etc. would make it more clear.

- `sysproc.c:707`
  One process called `rendezvous` with `tag`!

- `sysproc.c:708,710`
  It is removed from the list. The value the first process gave to `rendezvous` will be the value returned to the 2nd process calling `rendezvous` for the same tag. The `val` supplied by this 2nd process, is passed to `rendval` for the first process.

- `sysproc.c:712,713`
  Waiting until the first caller of `rendezvous` stops running (may be at a different processor). This is busy waiting, because the author thinks it would be a waste to sleep until the first process stops, and wakeup later. On uniprocessors, the process will never be running.

- `sysproc.c:714,716`
  Now that the first caller of `rendezvous` is not running, we can mess it up. Remember, the first caller did call `rendezvous`. If it was running, it was about to stop, waiting for a second process calling `rendezvous`—Agree now that it would be a waste to sleep? As the first caller is now sleeping waiting for us; make it ready again. It will pickup `rendval` as the return value—It was sleeping also in `sysrendezvous`.

- `sysproc.c:722,725`
  This is the starting point for the first caller of `sysrendezvous` with a given tag. Record the tag and the value so the 2nd caller notices; and link the process in the `Rgrp` hash.

- `sysproc.c:726`
  The process calling rendezvous was running; hence it was not in the scheduler ready queue. Set the state to `Rendezvous` to reflect that it is no longer `Running`; later it will be moved to `Ready` as you saw before.

- `sysproc.c:729,731`
  `sched` yields the processor. Other processes will run. The current one will not because it is not linked in a ready queue. When the 2nd caller calls `ready` for us, this process will eventually be `running`, and continue by returning from `sysrendezvous` with the 2nd caller's value.

  One thing to note: `rgrp` is unlocked before returning or `scheding`. After `sched`, no lock is necessary to complete the rendezvous.

This is a common scheme that you already saw several times: a process is moved out of the ready queue, it runs and blocks due to some reason; so it gets linked into the structure representing the reason. Later, the structure will be scanned and the process moved back to a ready queue.

4.7.2 Sleep and wakeup

`sleep()` *Waits for something.*

- `proc.c:403,452`
  Sleep and wakeup are complicated, as the plenty of comments suggest. The call `sleep(r,f,arg)` puts a process to sleep on `r` due to a reason represented by `(*f)(arg)`. `wakeup(r)` wakes up a process on `r`. If `wakeup` is called, the reason for sleeping no longer holds. There are several problems though.

  – A process may call `sleep`, and in the mean time, right after starting to call `sleep`, another process can call `wakeup` for him.

  – Right after calling `sleep`, the condition may change and we may change our mind and don't sleep. Therefore, `wakeup` can be called for a process that is no longer sleeping.

  – While a process is `sleeping`, it can be posted a note with `postnote`. That may even make the process die. So once more, `wakeup` can be called for a non-sleeping process.

  Read this comment at lines `:403,452`, and think about it. Perhaps the only way to make things more simple would be to change the semantics of `sleep` and `wakeup`.

  By the way, the `Rendez` parameter of `sleep` and `wakeup` may be called so because it is used to rendezvous the processes calling `sleep` and `wakeup`; but it has nothing to do with `rendezvous(2)`.

- `proc.c:458,460`
  Without interrupts and locking `rlock`.

- `proc.c:466,473`
  Important action, see the comment.

- `proc.c:475,481`
  The condition changed while calling `sleep`; no need to sleep any more. Note how `r->p` is set to nil, so `wakeup` knows there is nobody sleeping there.

  Can you understand now why `tfn` (`proc.c:518,522`) is given as an argument to `sleep` at line `:567`? Beware that `tfn` is not `up->tfn`!

- `proc.c:487,488`
  No longer `Running`, `Wakeme` is the state for sleeping processes. Besides, the process is sleeping on `r`. Any `postnote` will notice the state and update `p->r` to be nil; so that a later `wakeup` notices that there is no process to wake up.

- `proc.c:490,506`
  Doing the work of `sched`, instead of calling it. Why?

- `proc.c:509,513`
  A note was posted, report the abortion of the sleep.

`wakeup()` *Wakes up a sleeping process.*

- `proc.c:576`
  `wakeup` will wake up the process sleeping on `r`, if still there.

- `proc.c:582,585`
  Important action, see the comment.

- `proc.c:589,590`
  The sleeper was gone. No locks by now!

- `proc.c:592,593`
  Now getting the lock

- `proc.c:595`
  The process could go away between line `:588` and this line. So check that `r->p` is still there and is the `p` we know. Both checks are necessary since a different process could sleep on `r` between both lines.

- `proc.c:596,601`
  The process is awaken and placed back in the ready queue. The return value is true if a process was awaken.

- `proc.c:604,607`
  One way or another, we are done.

`postnote` is discussed together with notes in the next section.

There is a routine used to put the process to sleep for a while, awaiting for a resource.

`resrcwait()` *Waits for some time due to a reason.*

- `pgrp.c:261,277`
  `resrcwait` uses `tsleep` to wait for a resource. It sets the "ps" state to let the user know what is the process waiting for, and sleeps for 300 ms. `return0` (a procedure returning zero) is used so that when sleep checks the condition function, it returns zero and `sleep` puts the process to sleep. `rescrcwait` seems to be used just to await for free slots in the process table.

4.8 Notes

Users make system calls to `notify` and `noted` to handle notes. See `notify(2)`. Notes are posted by writing to a note file or by the system; see `proc(3)`.

The desing of Plan 9 notes is nice is that it services several needs: the kernel can use it to notify of exceptional conditions to the user process, and the `proc(3)` files can be used by user code to notify anything even through the network. Other systems (e.g. UNIX) do not have a means to asynchronously notifying to processes over the network (e.g. you must use either sockets or signals for that, and signals do not work across the network!).

4.8.1 Posting notes

`sysnotify()` *Sets up the process handler for notes.*

- `sysproc.c:572,578`
  `sysnotify` registers the handler for notes in the field `notify` for the current process. The address is checked to be valid because the user could lie—if the address is zero, the handler is being canceled.

- `sysproc.c:580,586`
  `sysnoted` simply does nothing. Why? the noted system call has nothing to do, the work is done in `trap.c`.

  Let's start by posting a note to a process.

`syswrite`
  `procwrite()` *Handles writes for `proc(3)`.*

- `devproc.c:720`
  A write to `/proc/n/note` leads to a call to `procwrite` with Qid `Qnote`. Remember the section on files in the previous chapter?

- `devproc.c:721,724`
  It is an error to post a note for a kernel process. It is an error to post a note message longer than `ERRLEN` characters.

- `devproc.c:727`
  Here is where the note is posted. `postnote` does the work. If you grep for `postnote`, the kernel calls it in several other places, where it feels that the system must post a note to notify something.

- `devproc.c:677,679`
  There is another way for the user to post a note, send it to a group of processes. If the file written is `notepg`, `pgrpnote` posts the note... `notepg()` *Sends a note to a group of processes.*

- `pgrp.c:12,40`
  ...by scanning the whole process table searching for not-dead processes with the same `noteid`. In the end, `postnote` is called to post notes; kernel processes are not notified. The author scans the table without locking the processes, and when he thinks he got a process, a lock is acquired and the check repeated—now without races. This pattern is used in several other places as you will see. Any error in `postnote` is ignored.

```
syswrite
   ...
   postnote()
```
Posts a note to a process.

- `proc.c:611`
  The note is `n`, the process notified is `p`, not the current process. `postnote` must lock `debug` in the `Proc` affected, but will do so only if `dolock`—i.e. some caller of `postnote` does not hold the lock.

- `proc.c:623,624`
  `flag` is `NUser` if `postnote` is called by `devproc`—the user wrote the note file. `nnote` is the number of notes posted (but not yet notified) for the process. So, if the kernel posts the note and there is no handler or `p->notified`, the number of notes is set to zero. `notified` is true while the process is being notified. Rationale: if there is no handler, there is no point in keeping previous notes so set the number to zero; if the process is being notified, and the note comes from the kernel, forget about pending notes after then one being notified, because the kernel one is likely to be important. Only when there is a handler and a pending note, `nnote` is preserved so that the previously posted note is kept for the user before the one posted by the kernel.

- `proc.c:626,631`
  The `note` array holds the at most `NNOTE` notes posted to the process. If there is space, `nnote` is adjusted and the note copied (posted) to the slot in the array. Both `msg` (the note text) and `flag` (the note flag) are kept in the array. `postnote` returns true if the note was posted.

- `proc.c:632`
  There is a note for the process, let it know.

- `proc.c:636,645`
  Race against `sleep`/`wakeup`. Get the lock and look at `p->r`. If non-nil, the process is sleeping. Note the "paranoid" check to ensure that the race did not mess things up. Locking for these three routines is so complex that security must go first. The process is pulled out of the sleep and made `Ready` again. A later `wakeup` would notice that `r->p` is zero and do nothing.

- `proc.c:649,650`
  Unless the process is doing a `rendezvous`, the post is done. The process will notice the post and handle things itself.

- `proc.c:653,664`
  Besides sleeping, the other reason a process may be waiting is on a rendezvous. Both `sleep` and `rendezvous` may be interrupted due to a note. If the state is `Rendezvous` (note the double check once the lock is gained!) the value to be returned is set to the representation of `-1` in two's complement (see `rendezvous(2)`). The process is then removed from the `Rgrp` hash queue and set `Ready`. Compare this code with the code in `sysproc.c:708,715`. In `postnote`, there is no need to wait until the process stops running (if it was so).

Now that the note is posted, and the notified process is ready, it will run sooner or later.

4.8.2 Notifying notes

Imagine the just notified process starts running.

- `proc.c:475`
  If the process was in `sleep`, or enters `sleep`, `notepending` is seen, and the ongoing system call is aborted at line `:512`. The `notepending` flag is reset (now the process knows it has notes), but `nnote` is still non-zero as there are notes in the process' `note` array. Something similar happens to processes with notes posted while doing a rendezvous.

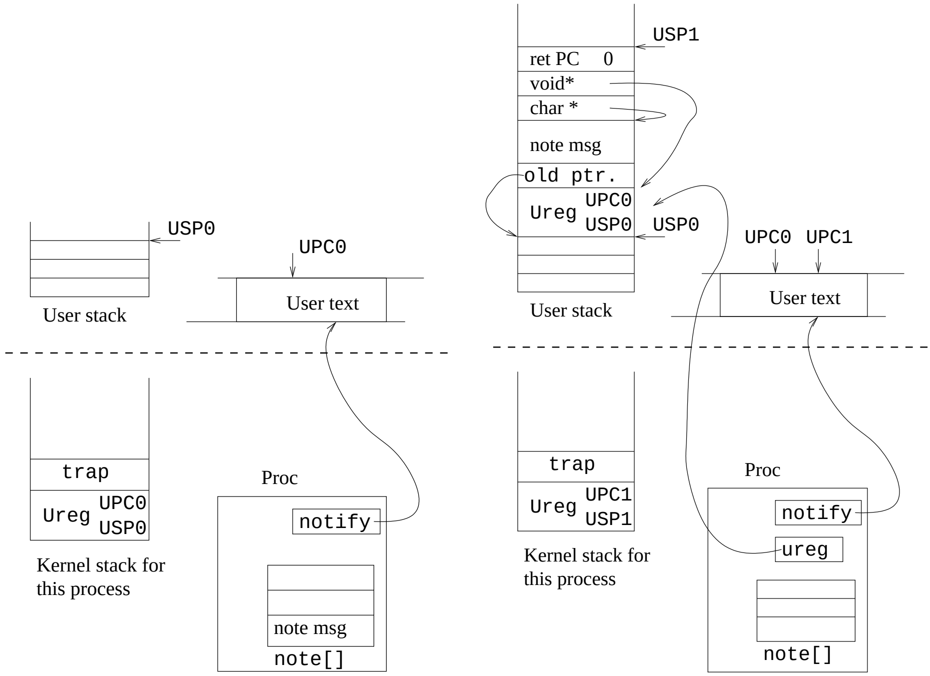
- `../pc/trap.c:532,538`
  In any case, when the notified process runs again, it will be inside the kernel, probably in `sched` or aborting a sleeping system call. One way or another, as procedures return, the process will reach `trap` (or the last lines of `syscall`). Ignore lines `:532,533` by now. Right now, the process is as depicted in figure 4.8(a).

  If not doing a fork, and the process has notes posted (`nnote` not zero), `notify` is called. The notified process was returning from an interrupt or a system call when `notify` gets called.

`trap`
  `notify()` *Notifies of a note for this process.*

- `trap.c:546`
  `notify` is the routine actually receiving the posted note. It will take appropriate actions depending on the note. You should note how the notified process handles the note itself. That is more simple than doing it in the notifying process because we are now running in the notified process context (e.g. user stack addresses can be used safely).

- `trap.c:552,555`
  Remember the check for `procctl` in `trap`? `notify` is taking care of "procctl" here—I defer the discussion of this until later in this chapter. If the process was posted a note, we pass these lines.



(a) The process right before being notified.

(b) The process after `notify` has setup the user stack for the handler. The next `iret` will make the handler run.

Figure 4.8: Notes are handled by the notified process. `notify` sets up the context for the handler. The previous state is recorded in the user stack.

- `trap.c:557,558`
  You know that `debug` is the lock to acquire when posting notes.

- `trap.c:559`
  The note is being handled, no longer pending.

- `trap.c:560`
  First, pick up the first note. If there more ones, they will be handled after the process has been notified of the current note (i.e. when the process enters the kernel again and starts to leave it in `trap`).

- `trap.c:561,566`
  If the note starts with "`sys:`", the kernel posted the note. Ensure that there is room to add to the note the user program counter and add it to the note. If the kernel posted the note, it is likely that the instruction pointed to by the user PC, caused a trap that caused the `postnote`. Therefore, the value for the PC is valuable to fix the bug.

- `trap.c:568,574`
  If the note was not posted by the user, and there is no handler (`notify` is null)

or the process is currently handling a note, the process is killed. `pexit` does the job. This is reasonable since the handler is probably faulting and it makes no sense to give it a second chance.

- `trap.c:575,579`
  The process is handling a note right now, do nothing; The check could be done at line `:537`, but notify would then have no chance to kill the process in case something caused a kernel posted note. Think of a process using an illegal instruction while running the handler for the note.

- `trap.c:581,584`
  Default action for user posted notes when there is no handler: die.

- `trap.c:585,586`
  Starting to setup the user stack to run the note handler (see figure 4.8(b)). We know there is a handler. Here the author makes room for a copy of the saved `Ureg` in the user stack. When the process is being notified, the note handler is supplied with a copy of the saved `Ureg`; the handler can make changes in the copied `Ureg` and the kernel will reflect those changes in the real saved `Ureg`. That way, a user can repair the cause for a note by changing the noted process context. Can you see that the kernel is using "user virtual addresses" directly? That is feasible because the kernel now runs within the context (i.e. address space) of the notified process. User virtual addresses can still be used from the kernel, they must be checked though to ensure that they really exist and have appropriate permissions.

- `trap.c:588,593`
  The user could lie regarding the address of the handler or its stack. Ensure that both the handler entry point and the place in the stack where handler arguments are copied are valid addresses. If they are not, the process dies. The space for the arguments must hold a copy of `Ureg`, plus room for an error message of at most `ERRLEN` characters, plus 4 machine words. See later.

- `trap.c:595,597`
  `up->ureg` is set pointing to the copy of the saved `Ureg` in the user stack. This is the copy for the user, and not the real `Ureg` (which is the parameter `ureg`). The real `Ureg` is copied into the user stack and a pointer to the just copied `Ureg` pushed on the user stack.

- `trap.c:598`
  What? did this before. Perhaps line `:595` should be deleted? Looks like the old `up->ureg` should be saved in the user stack before being updated to point to the current User's copy of `Ureg`. Note the comment.

- `trap.c:599,600`
  Make room in the stack for the just pushed `Ureg*` and the note message; copy the note message.

- `trap.c:601,604`
  Add room for three words: the return program counter, the pointer the the

copied `Ureg`, and the pointer to the note message in the user stack; and initialize them accordingly. Why is the return PC being set to zero?

- `trap.c:605,606`
  The real (note: `ureg` and not `up->ureg`) saved user stack pointer set to the new value for the handler. The real saved user PC set to the address of the handler (kept in `notify`). After trap returns leading to a return from interrupt, the reloaded process context would make it run the handler as if it had been called. The arguments are as they should be, but the return PC for the handler is zero! A return will cause a page fault (because address zero is not valid) and the process will surely die—because the system would post a note during the execution of the handler. But you know that a note handler must not return, you read `noted(2)`, right?

- `trap.c:607,610`
  One the note is notified, shift remaining notes (if any) to remove the first one. `lastnote` keeps a copy of the note just notified—used for debugging and while returning from the handler. Besides `notified` is set to record that the handler is running.

4.8.3 Terminating the handler

Assume now that the process note handler behaves correctly, and it calls `noted(2)` before returning from the handler.

`syscall`
  `sysnoted()` *Does nothing. The system call is just to enter the kernel.*

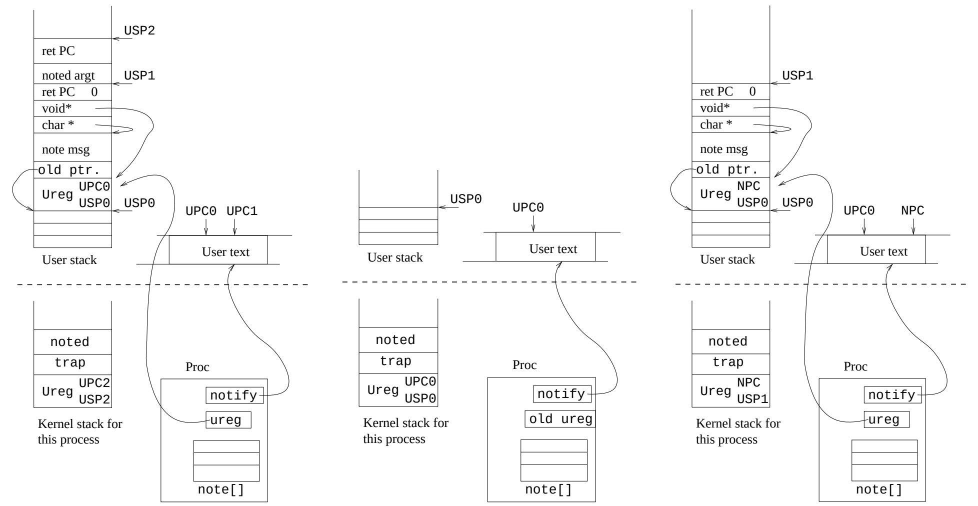
- `../port/sysproc.c:581,586`
  When `syscall` calls `sysnoted` (the system call for `noted(2)`) it does nothing (Although I think that the code in `noted` discussed below could be moved to `sysnoted`).

- `../pc/trap.c:532,533`
  The system call number is `NOTED`, and `noted` is called. The `Ureg` given corresponds to the context for the user within the user library `noted` function, right before the handler terminates. The second argument is a pointer to the arguments of `noted(2)`, which are just an integer (the return PC for `noted` is in the top of the user stack). Figure 4.9(a) shows how the stacks look like.

`syscall`
  `noted()` *Handles a posted note.*

- `trap.c:621`
  `noted` must restore the user context as it was before the note was notified.

- `trap.c:627,631`
  If `notified` not set, and the argument to `noted` is not `NRSTR`, abort. Read the `noted(2)` manual page if you have not done so. `notified` should be true, because `noted` is being used to return from a handler to the previous context, but looks like `noted(NRSTR)` can be used even when there seems to be no handler.



(a) The process right after calling `noted(2)`.

(b) The process after `noted(NCONT)` did its job. It will resume where it was before the note.

(c) The process after `noted(NSAVE)` did its job. It will start another handler.

Figure 4.9: Returning from a note handler and restoring the user context.

- `trap.c:632`
  The handler is no longer running. See how `notified` is used to report that the user context corresponds to a note handler?

- `trap.c:634,637`
  `nureg` and `oureg` set to the handler copy of the `Ureg` saved before the process was notified.

- `trap.c:638,642`
  The copied `Ureg*` and `Ureg` must be valid addresses. Kill the process otherwise.

- `trap.c:651,660`
  Can we trust that the copied `Ureg` is reasonable? The user could try to mess up with segment selectors in the copied `Ureg` and the kernel could crash or compromise security if the user was trusted. The same can happen to bits in the flag word that enable/disable interrupts and affect system issues; however, other bits in the flag word can be changed.

- `trap.c:662`
  This is it!, now that we trust the handler `Ureg`, copy it back to the currently saved `Ureg`. After `noted` returns, the return from interrupt will reload the (fixed) process context. Usually, the user PC and SP in the fixed `Ureg` would be those corresponding to the user context before the process was notified (as shown in figure 4.9(b)).

- `trap.c:664`
  The argument to `noted` specifies what to do next.

- `trap.c:665,674`
  Be it `NCONT` or `NRSTR`, the pointer to current copy of `Ureg` for note handlers is set to its old value. (But remember line `:595`! A bug there?).

- `trap.c:676,690`
  `NSAVE` arranges for the user stack to be almost preserved as it stands (see figure 4.9(c)). The user stack is set above the old `Ureg`, leaving place for three parameters and a fake return PC; a pointer to the `oureg` is set as the first parameter. But, parameters for who? If you ask this, did you read `noted(2)`?. The note handler calling `noted` has adjusted the `PC` in its copy of the `Ureg` so that a different routine is called when it returns; the `NSAVE` is set for `noted` to let it know that it should keep a handler stack frame for the routine. Using this "trick", the user can chain handlers for notes.

- `trap.c:691,695`
  None of the known flags, let the user know and continue as if `NDFLT` was said.

- `trap.c:696,702`
  The process did not say `NCONT` to let the process continue, and did not say `NSAVE` to chain another handler. The reason for the note is not likely to be repaired and the process must die.

  When `noted` returns, the saved `Ureg` is reloaded (including any fix from the note handler), and the process resumes operation.

By the way, most of the blocks with `qunlock`, `pprint`, and `pexit` used to handle errors could be folded into a common error handling block by using a `goto` like other kernel routines do; and perhaps the arithmetic done with the user stack pointer could be simplified a bit.

4.9 Rfork

Now it's time to see how are processes created and destroyed. The system call used to create a process is `rfork`. Read the `rfork(2)` manual page.

Plan 9 follows UNIX (as with many other things) in that processes are arranged into a hierarchy. The system creates an initial process, and remaining ones are created using `rfork` as descendant of the first one.

Using a hierarchy of processes is good in that provides a natural means to share resources, by setting them up in the parent before spawning any child. Unlike UNIX, Plan 9 is able to adjust the resources a process has, including its name space, so that any process can get a brand new instance of the resource, or a clone of the resource. But you already read this in the manual page, right?

Although the `proc(3)` device permits handling of processes using files (over the network), typical operations of process creation and program execution are performed by regular system calls on the local node. Moreover, processes can only share resources (namespace, descriptors, etc.) within a node. This is not a severe problem, because name spaces can be constructed to use the same resources (files) over the network. The approach chosen by the author is simple, yet effective.

`sysrfork()` Entry point for `rfork`. Creates new processes or adjusts the current process resources.

- `../port/sysproc.c:20`
  `sysrfork` is the entry point for the `rfork` system call. It is called from `syscall` in `../pc/trap.c`.

- `sysproc.c:31`
  The flag supplied to `rfork` is very important, because it controls what `rfork` will do. It is made of an OR of bits, stating that particular resources for the process should be (re)created, duplicated, or shared.

- `sysproc.c:32,38`
  The user cannot request that file descriptor group be both copied (`RFFDG`) and and cleaned (`RFCFDG`). The same for the name space and the environment. Flag names are not so hard to remember: they all start with `RF` (for `RFork`). Now, take the file descriptor group (`FDG`) as an example: to share it, say nothing; to duplicate it, say `RF` and `FDG`, i.e. `DFFDG`; to clean it, put a `C` before the flag name, i.e. `RF` and `C` and `FDG`, that is `RFCFDG`. Calls to `error` will jump to the label set by the last call to `waserror`—at `../pc/trap.c:496`.

- `sysproc.c:40`
  Important!, if `RFPROC` is said, the system must create a new process and use remaining flags to set up its resources. If `RFPROC` is not said, changes affect the current process. The set of flags for `rfork` is a kind of micro-language, used both to adjust resources in the current process and to control the initialization of resources for the new process. Lines `:41,78` are executed when `rfork` is adjusting resources for the current process.

- `sysproc.c:41,42`
  `RFMEM` requests data and bss segments to be shared between the parent and the child, but there is no child. `RFNOWAIT` request the child to be "independent" of the parent—more about this later. As there is no new process, these flags have no sense.

- `sysproc.c:43`
  the `fgrp` has to be either copied or cleared. It makes sense to copy the `fgrp` even when there is no new process. The `fgrp` may be shared among the current and other processes, the current process is probably going to adjust its `fgrp` and does not want to disturb the other processes. By calling `rfork` with `RFFDG` set, the process can "clone" the `fgrp` and get its own copy.

- `sysproc.c:44,48`
  `up->fgrp` set to either a duplicate of `up->fgrp`, or to a duplicate of nil—i.e. to a fresh new one. You already saw in the last chapter how `dupfgrp` works when creating a new group.

`dupfgrp()` *Duplicates an `Fgrp`.*

- `pgrp.c:185,208`
  When the `fgrp` is not being created, but being cloned from an existing one, these lines execute. Lines `:187,190` get the number of used entries in the cloned group—and round that number to a multiple of `DELTAFD` entries. Later, memory for the array is allocated, the reference count set to one and the array initialized from the cloned one. `incref(c)` adds an extra reference to each channel in the cloned group. The author ensures that the `fgrp` appears to grow in chunks of `DELTAFD` entries, no matter if that was really the case or not: he sticks to his design.

`closefgrp()` *Releases a reference to a* `Fgrp`*.*

- `sysproc.c:49`
  The process got a new `fgrp`, so the old one is no longer referenced by the process. `closefgrp` releases the reference to the previous `up->fgrp`; if the reference count gets down to zero (no other process using this `fgrp`), all file descriptors in the `fgrp` are closed and the `fgrp` deallocated.

- `sysproc.c:51,59`
  The name space adjusted. `pgrp` is actually the name space group. First, a new `pgrp` is created (an empty mount table for the process). If the `pgrp` is to be copied, `pgrpcpy` duplicates in `up->pgrp` the old name space. There is no `duppgrp` routine (although a simple wrapper for `pgrpcpy` could be created to make the code look like the one for `fgrp`). The `noattach` flag prevents `mount` and `attach` from being used on the name space. The old `pgrp` value is set for the new name space. Otherwise, a process could bypass the `noattach` flag by duplicating the name space. I think that a `duppgrp` routine could take care of this detail too. Finally, the reference to the old name space group is released.

- `sysproc.c:60,61`
  If `RFNOMNT` was set, forbid mounts and attachments on the name space by setting the flag.

- `sysproc.c:62,66`
  Start a new rendezvous group for the process. That is used to avoid clashes in the rendezvous tag namespace, and to prevent some processes to rendezvous with others. A new `rgrp` is created and the reference to the one dropped.

- `sysproc.c:67,74`
  Environment group adjusted. Again, I miss the existence of a `newegrp` and/or `dupegrp` routine—But that's just a naming issue mostly. The creation of a new `egrp` is done by allocating it and setting the reference counter to one. If the environment is to be copied, `envcpy` will recreate in `up->egrp` the variables found in the old environment.

- `sysproc.c:75,76`
  Create a new note group for the process. Similar to what was done for rendezvous, but more simple. Remember that the whole process list is scanned to determine who belongs to a process group when posting notes? To create a new note group it suffices to get a new `noteid` value.

- `sysproc.c:80`
  The previous lines were executed only when resources for the current process should be adjusted. Did not return at line `:77`, so the caller wants a new process. Allocate it at this line. If you remember from the previous chapter, `newproc` allocates a free `Proc` entry and initializes it with everything set to null but for the kernel stack, the process pid and the process noteid. The process state, the "ps" state, and the FPU state are set to initial values too.

- `sysproc.c:82,84`
  The newly created process has the same FPU saved state, and appears to be executing the same system call (number and arguments).

- `sysproc.c:85`
  No errors for the new process.

- `sysproc.c:86,88`
  The new process uses the same root directory and the same current directory. `rfork` usually duplicates the calling process unless told otherwise. Now there is another reference for `dot`.

  An `incref` on `slash` seems to be missing. I think that the author considered it unnecessary because all processes have the same value for `slash` (`boot` gets one pointing to the root device and `rfork` makes the child have the parent's `slash`), the `slash` channel will never be released. Nevertheless, I think it would be better to `incref`/`decref` it.

- `sysproc.c:90,96`
  The set of notes for the current process is duplicated for the new one. `dbgreg` points to the saved `Ureg` after traps, none by now. Also, there is no note handler running for the new process, set `notified` to 0.

- `sysproc.c:98,104`
  Going to work with segments, so gain the lock `seglock` and prepare to release it if there is an error. `waserror` is used to jump back to it on an error, release the lock, and re-raise the error to the `waserror` in `syscall` (`trap.c`). `dupseg()` *Clones or shares a segment.*

- `sysproc.c:105,107`
  For all segments, call `dupseg` to duplicate the `i`th segment in the `seg` array. The `n` lets `dupseg` know that the segment should be duplicated and not shared or cleared. I'll get back to `dupseg` on when talking about virtual memory on chapter 6. The only things you should know by now is that:

  - `dupseg` returns the segment given incrementing its reference counter for `TEXT`, `PHYSICAL`, and `SHARED` segments. You see how text segments (read only) and physical memory segments are shared between the parent and the child no matter what `rfork` is told.
  - `dupseg` creates a fresh new stack segment and returns it, when the segment given is a stack. The base address and size for the new stack are the same as in the passed segment.

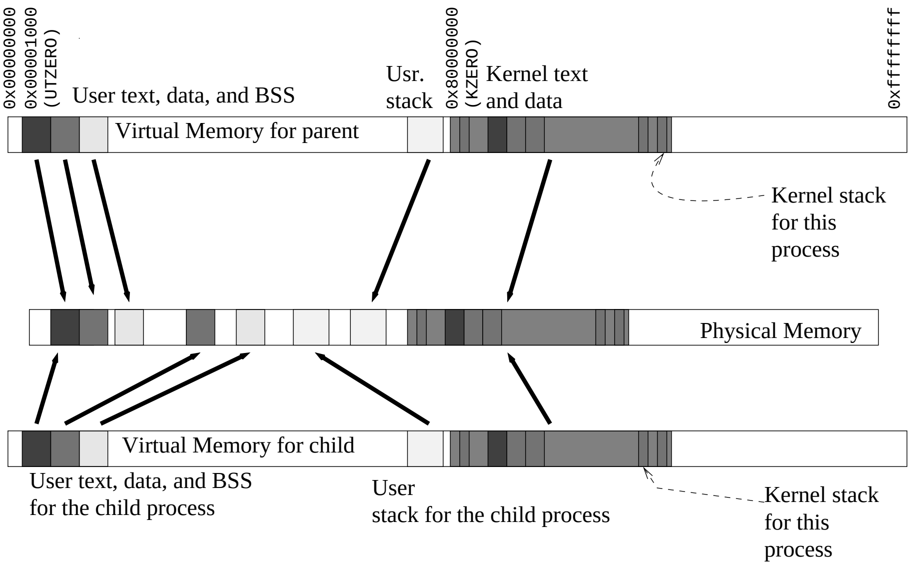


Figure 4.10: Virtual memory layout for a forked process. The layout of physical memory is not really like the one shown; besides, Plan 9 uses paging, and does not map whole segments.

- for `DATA` and `BSS` segments, `dupsep` will add a new reference and return the segment given (i.e. share it) or it will create a copy of the given segment, depending on the share flag (`n`).

So, after the calls to `dupseg`, the new process has its `seg` array setup either sharing the parent's segments or with a copy of parent's segments. Of course, text segments are always shared with the parent and the child always gets a fresh new stack. Apart from that, everything else in virtual memory looks like the parent's memory; see figure 4.10.

- `sysproc.c:108,109`
  Lock released and the last error label removed. A call to `error` to notify an error will jump now to the `waserror` at `syscall`: there is no cleanup to be done here now and the direct jump to `syscall` can be permitted upon errors.

- `sysproc.c:111,155`
  File descriptor group, name space, rendezvous group, and environment group are either duplicated from the parent for the new process, or created, or shared. It all is the same that was done by `rfork` to adjust resources for the current process when no `RFPROC` flag was given (The main difference is that resources are shared when they are neither cleared nor cloned). Perhaps some code could be shared and `rfork` made shorter; nevertheless the code is simple and easy to follow.

- `sysproc.c:156`
  `hang` is a flag stating that the process should stop when doing an `exec` to give

the user a chance to debug it. The child gets the same flag than the parent. As creating new processes is usually an `rfork` plus an `exec`, it makes sense to propagate the flag.

- `sysproc.c:157`
  "permissions" for the file representing the new process are the same they were in the parent.

- `sysproc.c:159,162`
  Read the comment! When you do an `rfork` requesting the creation of a new process, the parent is given the pid of the child, and the child appears to return from `rfork` just like the parent, but returns zero instead. `forkchild` sets things up in the child so that it would appear to be returning from `rfork` with a return value of zero. Note that `trap` (`../pc/trap.c:227,230`) did set `dbgreg` to point to the `Ureg` saved by the hardware when `rfork` was called.

`sysrfork`
  `forkchild()` *Handcrafts the child kernel stack.*

- `../pc/trap.c:772`
  `forkchild` has to be machine dependent because it assumes the stack layout for the current architecture.

- `trap.c:777,782`
  When the scheduler jumps to the new process, it will jump to the `sched` label in `p`. The author initializes the label so that the kernel code executing is not `sched` (which usually sets the label when the process is leaving the processor), but the first instruction of `forkret`. `forkret` will then return from the `rfork` system call in the child as if it had called `rfork`. The kernel stack pointer is set to the end of the kernel stack for the new process, but leaving room for a copy of an `Ureg` structure and two extra words. The two extra words are the return PC and the argument (the `Ureg*`) of the `syscall` routine. Yes, for the `syscall` routine and not for the `forkret` routine. More later.

- `trap.c:784`
  `cureg` points two words after the top of the kernel stack for the new process, that is where an `Ureg` is going to be copied.

- `trap.c:785`
  Important!, the `ureg` passed to `forkchild` is copied into the kernel stack for the new process. That `Ureg` was the one saved by the hardware when the user called `rfork`. Therefore, `forkret` has its own copy of that processor context.

- `trap.c:787`
  This is where `rfork` is forced to return zero at the child. The return value will be taken from the return-value register, which is `ax`. `ax` is set to zero in the `Ureg` copied for the child process. The whole picture can be seen in figure 4.11.

- `trap.c:791`
  `insyscall` is set and reset by `syscall` upon starting/terminating a system call. Reset it for the child since it is completing its "call to `syscall`".

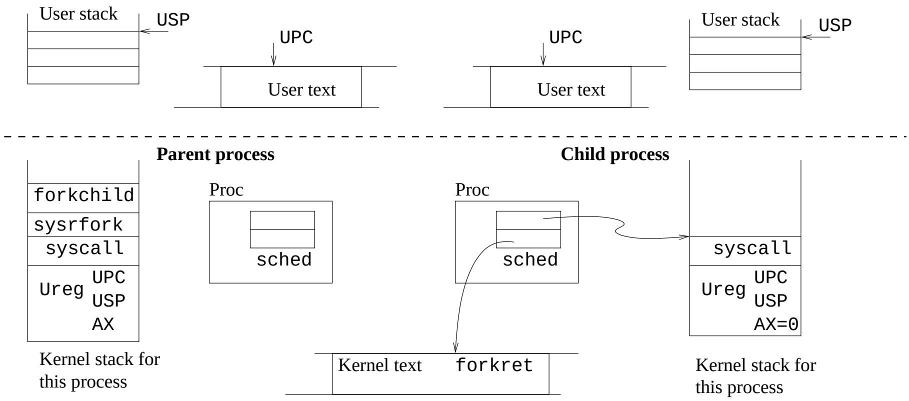


Figure 4.11: Kernel stacks and `Procs` after `forkchild`.

`sysrfork`

- `sysproc.c:164,165`
  The parent of the child is the current process.

- `sysproc.c:166,172`
  If `RFNOWAIT` was set, the parent will not call `wait(2)` for the child, so make the parent pid be the pid for the initial process. Every process likes to have a parent who cares for it! That process will `wait` for the child. More about `wait` in a following section. If it was not set, increment the number of children for the parent.

- `sysproc.c:173,174`
  No request to start a new note group, so keep the parent's. Remember that `newproc` did set `noteid` to be a new group.

- `sysproc.c:176,181`
  Initialize the state of the FPU, zero the time counters, and record at `time[TReal]` the starting time for the new process. By subtracting that value from `ticks` at processor 0, the system can know how much (real) time passed since the process was born. Names for the text file (binary file) and the user named duplicated.

- `sysproc.c:183,187`
  The comment says it all. The reason is that when a segment gets shared, permissions on the page table for the memory affected can change too. Therefore the MMU has to be flushed to drop the old permissions from cached page table entries. This will become clear in a following chapter.

- `sysproc.c:188,189`
  Priority (both base and actual) inherited from the parent.

- `sysproc.c:190`
  The child appears to have run at the same processor the parent was running at.

- `sysproc.c:191,193`
  If the parent is wired to a processor, the child gets wired to that processor too. `procwired` wires the process as you saw before.

- `sysproc.c:194,195`
  All set. The child gets linked into the ready queue and set `Ready`. When the scheduler is called, it could elect the child.

- `sysproc.c:196`
  When the current process gets back to the processor after `sched` runs other processes, the pid of the child is returned as the result of the system call.

  That was okay for the parent, but what does the child now?

`forkret` *Appears to return from* `syscall`*.*

- `../pc/l.s:539`
  When the scheduler picks up the child for running, it jumps to the `sched` label for the process and `forkret` starts running. `forkret` does exactly what is done after `syscall` returns from the call at `plan9l.s:43`. The only difference is that `syscall` was never called by the current process. The stack was set by `forkchild` as if `syscall` was called, so that `forkret` could believe in that.

  One thing to see here is that `forkret` is actually assuming that the process returns from `trap` and not from `syscall`; but the code in `forkret` and `plan9l.s:45,52` is exactly the same, which means that it would work anyway. Perhaps it would be better to move the `forkret` declaration from `l.s` to `plan9l.s:44`, since it is returning via `syscall` and not via `trap`.

- `l.s:540`
  throw away the fake `Ureg*` argument in the stack.

- `l.s:541,545`
  Reload the processor registers and segments from the `Ureg` saved in the child kernel stack by `forkchild`.

- `l.s:546`
  ignore the couple of words in the `Ureg` above the hardware saved processor context in the stack.

- `l.s:547`
  Here we go! The `iret` reloads the processor PC, SP and their segments so that the process continues back in user-level returning from the `rfork` system call. The `ax` register restored at line `:541` was set to zero by `forkchild`, therefore, `rfork` returns zero to the child.

- `../pc/trap.c:535,538`
  To complete the discussion of `rfork`, here is my guess about the reason for the `scallnr!=RFORK` in `trap`.

  Suppose that the child was setup by `forkchild` to start running in `trap` and not in `forkret`—probably by copying the kernel stack for the new process in this hypothetic previous version of the kernel source. If the system call was `rfork` (and it was called by `trap`), a new process would be created and both processes would return from the `rfork` system call back to `trap`.

If that would be the case, and the `RFORK` check was removed, both the parent and the child would check for pending `procctls` and pending`notes`. Perhaps the code used variables in the stack that could cause the child to be posted a note that was really for the parent.

Regarding the actual code, the only utility I can see for this check is to avoid posting a note to the child before giving it a chance to either install its own note handler or issue an `exec` system call and be forbidden for parent's faults. In any case, the child has its `notify` field as the parent has it.

4.10 Exec

Now that you know how a new process is created, let's see how it can execute a new program. It needs to both locate the program to be executed and execute it. The separation of concerns between `rfork` (creating a process) and `exec` (executing a program) allows a parent to perform adjustments on the child process before executing its program. This comes back from the days of UNIX.

An executable in Plan 9 is any file with the execute permission set. Unlike other systems, the file name has nothing to do with the fact that it could be executed. The file must be either a text file or an `a.out` file. A text file to be executed usually starts with "`#!`" and the path of the program to interpret the file; for example, `rc` scripts start with "`#!/bin/rc`". `a.out` files are generated by the Plan 9 assembler (see `a.out(6)`), and contain, among other things, the following items:

- An `Exec` header, with information about the image of the program (sizes for segments, etc.).
- The executable code for the text segment.
- The image of the data segment with initialized variables.

4.10.1 Locating the program

`sysexec()` *`exec` system call. Executes a new program.*

- `../port/sysproc.c:209`
  A process willing to destroy its memory in favor of executing a brand new program calls `exec`; this is the entry point for the system call.

- `sysproc.c:226,227`
  The first argument is a file name, where the executable for the new program is to be found. So, check that the address is valid, and get a pointer for it. Remember that the user virtual addresses are valid, therefore, the pointer supplied by the user is ok for kernel usage.

- `sysproc.c:228`
  Ignore this by now. To satisfy your curiosity, `indir` seems to mean "indirection".

- `sysproc.c:230,234`
  `tc` is the text channel, or the channel pointing to the text file for the new program. `namec` resolves the name in the current name space and returns an open channel checking for execute permission. The `waserror` prepares for closing the channel and re-raising the error if the following code raises an error.

- `sysproc.c:235,236`
  `namec` did set `up->elem` with the file name—without any previous path component. So now `elem` contains the name for the text file. Again, ignore the `indir` thing; just notice that it is zero now.

- `sysproc.c:238,240`
  You know this, right? Using the channel type to call the appropriate `read` routine to get the `Exec` header for the text file. At least two characters wanted. Yes, the `Exec` header is more than two characters, but keep on reading. If the error is raised, you get back to line `:231`, the channel is closed and the error re-raised.

- `sysproc.c:241,243`
  Extracting the magic number from the header, as well as the size of the text segment and the entry point. The `Exec` header is defined at `/sys/include/a.out.h:2,12`. `read` could get just two characters and all these fields could be trash. The numbers just extracted are stored in big-endian order in the `Exec` header, `l2be` is "little to big endian", however, that transformation on a big endian yields a little endian value; never mind, the fact is that `l2be` convert those values to a little endian representation—shouldn't be this a machine dependent operation?

- `sysproc.c:224,250`
  If the whole exec header was read and the magic number states that it is indeed an `a.out` file, it can be executed. The `break` would break the loop used to search for the text file, and execution would continue at line `:275` were `a.out` binaries are loaded. The error is raised in case the entry point is set before the start of the text segment for the user plus the size of the `Exec` header, or in case it is beyond the text segment (plus the size of the `Exec` header). The image in memory will contain the `Exec` header and then the text segment, hence the range—text images look very much like the file. Besides, the entry point should be within the user portion of the virtual address space (not with the `KZERO` bit set).

- `sysproc.c:252,254`
  Not an `a.out`. It is a file interpreted by another program.

- `sysproc.c:255`
  The exec header is copied into a character array.

- `sysproc.c:256,257`
  If the line does not start with "`#!`", it is not an script, so don't know what kind of binary it is. Ignore `indir` once more.

`shargs()` *Builds arguments for shell scripts.*

- `sysproc.c:258`
  `shargs` takes the line array, the number of characters kept at line, and builds in `progarg` an argument vector for the program. If you read lines `:441,469` you will see how that is done. Can you guess why the loop at `:447,449`? The number of arguments filled up is kept in `n` upon `shargs` return.

- `sysproc.c:261`
  `indir` is set when the file is to be interpreted by another program!

- `sysproc.c:265,266`
  `shargs` filled up `progarg` according to what follows `#!`. Now, consider an rc script named "`/tmp/f`" starting with "`#!/bin/rc -e -s`", when the file gets `exec`ed, `/bin/rc` should run with the command line `/bin/rc -e -s  /tmp/f`. `shargs` would have filled up `progarg` for the command line `/bin/rc -e -s`, but it knows nothing about the final missing argument. These two lines at `exec` are supplying as the final argument, the name of the file to be interpreted—note that the argument vector must be null-terminated.

- `sysproc.c:267,268`
  The first parameter in the argument array supplied as the second argument to `exec` is no longer valid, so remove it from the argument array.

- `sysproc.c:269,270`
  The file being executed is not the script, but its interpreter. The name of the interpreter is at `progarg[0]`. Besides, the interpreter should believe that it is named after the script name, not after the file containing the interpreter code. The first parameter for the program executed is set as the script file name, which was `elem`.

- `sysproc.c:271,272`
  Now let's get back to `indir`. You see how the channel for the script file is closed (and the error label popped because we already closed the channel). The loop will iterate once more with `file` set to the file being `exec`ed (the interpreter) and `indir` set to one (at line `:261`).

  Should the interpreter file on this new iteration be another "`#!`" file, the test at line `:256` would raise an error. The author does not want an interpreter to be an interpreted file! That can appear to be a restriction but it is not—interpreters are usually binary files, and if they are to be scripts, the can be easily wrapped with a silly binary file that calls the script. Should the author allow nested interpreters, a loop could arise because a malicious (or dumb) user could setup two files to interpret themselves recursively; e.g. file `/a` starts with `#!/b` and file `/b` starts with `#!/a`. It's more simple to forbid nested interpreters than it is to check for looks in the nested interpreter call. Besides, allowing nested interpreters would require more complex code in `exec`. Despite that, the author wrote `sysexec` in a way that makes it easier to allow it to handle nested interpreters.

  If the interpreter is a binary and not an interpreted file, the check for `indir` at line `:235` preserves `elem` as the script file name even though it is the interpreter the one being executed, and the `break` at line `:249` would lead to the code `exec`ing an `a.out` file.

4.10.2 Executing the program

`sysexec`

- `sysproc.c:275,276`
  Starting to execute an `a.out`, be it an interpreter or not. Now extract the lengths for the `data` and `bss` segments. The lengths for the `text` segment and the entry point had to be extracted before to check for illegal entry points.

- `sysproc.c:277`
  `t` is set to the end of the text segment. That is the first address of the segment (`UTZERO`), plus the sizes for the `Exec` header, and text segment proper. The `+BY2PG-1` and `&~(BY2PG-1)` is rounding the computed value to a page boundary. A page can be either text or data, but not both.

- `sysproc.c:278`
  `d` set to the end of the data segment, computed by adding the size of the data segment to the just computed (and rounded) end of the text segment.

- `sysproc.c:279,280`
  The end of the BSS (`b`) computed the same way.

- `sysproc.c:281,282`
  Don't trust the exec header. The end of text, data and bss segments should not invade the high part of the address space, used for the kernel. The `error` would jump to line `:231`, where the channel is closed, and the error re-raised.

- `sysproc.c:287`
  `nbytes` counts how many bytes are to be pushed in the user stack. You already know that the bottom of the stack is used for a profiling clock. This "first pass" counts the number of bytes to be pushed on the stack.

- `sysproc.c:288`
  No arguments pushed yet.

- `sysproc.c:289,296`
  If exec-ing with an indirection (i.e. an interpreter), the argument array is the `progarg` computed by `shargs`. Count the bytes for the null-terminated strings in `progarg`.

- `sysproc.c:297`
  `evenaddr` seems to fix the passed parameter to start at an even address. Some busses would raise an alignment error exception otherwise; but on the PC, `evenaddr` does nothing.

- `sysproc.c:298,307`
  Besides any argument counted in the case of an interpreter, the arguments given as the second parameter to `exec` have to be accounted for too—note that

for interpreters, the first parameter (the script name) was removed from the argument array; that is to avoid counting it twice. The calls to `validaddr` are ensuring that both `argp` and the strings kept there reside at valid user virtual addresses. The call at line `:299` checks the first word (the first page actually) for `argp`; when the page offset for the `argp` address is less than then size of the word, `argp` is jumping into the next page, so call `validaddr` once more to verify that the next page is still in place. Calling `validaddr` every pass in the loop would be a waste. The number of bytes to be pushed is incremented with the length for each argument, as reported by `vmemchr` (plus one for the final zero). `vmemchr` is like `memchr`, which returns the pointer to the first occurrence of a character (0) in a string (`a`); unlike `memchr`, `vmemchr` checks that the memory where the string resides is valid user virtual memory. By subtracting the start of the array (`a`), its length is computed.

- `sysproc.c:308`
  the size of the user stack is now known: One pointer per argument plus a null terminator for `argv`; plus the actual size of the arguments, rounded to a multiple of the word size.

- `sysproc.c:310,315`
  The comment says it all. On Intels it can waste a bit of memory but who cares. The author is still computing the size of the stack.

- `sysproc.c:316`
  Count the number of pages needed for the initial stack.

- `sysproc.c:321,322`
  Ensure the the user stack does not get too big. `TSTKSIZ` is the maximum allowed size for the "temporary" user stack being setup now.

- `sysproc.c:324,328`
  Going to operate on process's segments, `qlock` it. Note the use of a `QLock` (long waiting, maybe), and the use of `waserror` to release the lock in case of errors.

- `sysproc.c:329`
  A new stack created. `ESEG` is an extra segment slot used for `exec`. Right know `exec` could still fail, and you don't want to loose your user-level stack yet. This stack segment goes from `TSTKTOP-USTKSIZE` to `TSTKTOP`; noticed it is not *U*`STKTOP`? The author does not want to mess up the current user stack because `exec` can still fail, therefore, a temporary stack segment is created right below the user stack. `TSTKTOP` (`../pc/mem.h:54`) is precisely `USTKTOP-USTKSIZE`. Even though the stack is not at `TSTKTOP`, pointers pushed on it have values assuming that it starts at `USTKTOP`. This stack is going to move to its proper location, but later. By the way, I think that the comment that says "putting it in kernel virtual" is a bit confusing, since the stack is being built at the user portion of the virtual address space.

- `sysproc.c:331,350`
  setup the stack arguments for the new process. Arguments are copied appropriately including `progarg` when `indir` is one. You should understand the code.

Remember that the pointers pushed (i.e. `:346`) assume that the stack is mapped at `USTKTOP`, and not at `TSTKTOP`.

- `sysproc.c:352`
  `elem` was kept with either the binary file name (`indir` not set) or the script name (`indir` set); copy it as the name for the process text.

- `sysproc.c:354,363`
  Old segments "released". This is a point of no return. Only segments between `SSEG` and `BSEG` are released. That includes the current user stack (which is not shareable), text (which is being replaced by exec), data (which is being replaced by exec) and BSS (also replaced by exec). `putseg` decrements the reference counter for the segment and releases resources (memory, mostly) held by the segment when the counter gets down to zero.

- `sysproc.c:364,370`
  From the BSS on, only segments marked as "close on exec" are released. Remaining segments are kept. Shared segments created by the process would lie between `BSS` and `NSEG`; so they would be kept shared between the parent and the child. `fdclose()` *Closes file descriptors with a matching flag.*

- `sysproc.c:375,377`
  File descriptors marked as "close con exec" on the `fgrp` for the new process are released. `fdclose` closes all open file descriptors which happen to have set the flag passed it. In this case, all open file descriptors marked as `CCEXEC` would be closed.

- `sysproc.c:379,383`
  `tc` is the channel to the text file, `attachimage` returns an `Image` corresponding to that channel. The thing going on is caching. The `Image` structure, discussed later in the virtual memory chapter, is responsible for caching images of text files. If someone else is executing the program found at the file pointed to by `tc`, the memory used to keep the text loaded will be shared because the `Image` used would be the same. Don't worry too much about this, just think that the `Image` contains a segment (`img->s`) used as a cache for the program text. `ts` is kept pointing to the text segment and `seg[TSEG]` is set accordingly.

  The comment states that the image is "locked" when returned. That is because the author is going to update the segment held by the `Image`. The text segment may be shared by different processes and it would make no sense to acquire the `seglock` on one of them to operate on the segment. Instead, the `Image` must be locked to work on the text segment.

- `sysproc.c:384,387`
  You will know when virtual memory be discussed. Just to record that all the text should be "flushed" because it is now shared, and also to know where is the text in the image.

- `sysproc.c:389,398`
  A new data segment created and set in place. The `Image` for the data segment

(the place where memory comes from) corresponds to the binary file where the text was found, but starting after the text. You know that `a.out` files keep both text and data.

- `sysproc.c:399,400`
  The BSS segment created. The "zero-fill on demand" means that pages will be brought in for the segment as needed, they will be filled to all zeros when brought.

- `sysproc.c:402,412`
  `exec` passed the point of no return, so there is no problem to relocate `ESEG` into its place, `SSEG`, which is the proper location for the user stack. Now that the `seglock` is released there is no need to jump back to line `:325` on errors. The `base` address and top of the stack is set, and `relocateseg` adjusts the segment so it starts at `USTKTOP`, where it belongs. The movement is done by changing the virtual memory address translations.

- `sysproc.c:414,419`
  Read the comment, you already know about priorities. The "device character" for the root device is "/", so the kernel is checking that the file comes from the root device. If that is the case, the priority is adjusted accordingly; otherwise the process keeps the priority it had (probably inherited from the parent at `rfork`).

- `sysproc.c:420,421`
  Remove the error label first, so that if the close for the channel fails, `exec` would not close it again.

- `sysproc.c:428,433`
  No notes yet, and FPU state initialized.

- `sysproc.c:434,435`
  If `hang` was set, honor it by by setting `procctl` to `Proc_stopme`, which means that the process will be stopped for debugging before returning to user level.

- `sysproc.c:437`
  Finally, `execregs` initializes the user stack pointer and program counter.

`sysexec`
  `execregs()` *Initializes user registers so the program starts in its entry point.*

- `../pc/trap.c:706,719`
  `execregs` starts by setting up `sp` to the actual top of the user stack, with `ssize` bytes on it. Then it pushes the number of arguments to complete the main entry point arguments. As `syscall` did set `dbgreg` to point to the `ureg` saved by the hardware, the only thing to be done is to update on it the user stack pointer and the program counter. The return value for `sysexec` is the address of the profiling clock, which might be used by the user-level library code, but seems to be not relevant for the kernel. Remember that `exec` does not return when successful, so the return value can be only of interest to the assembler entry point for Plan 9 processes.

By the way, in case you didn't guess, the loop at lines `sysproc.c:447,449` is to ensure that the first line of the script fits within the size of an `Exec` header. Remember that `exec` read the header and then copied it to `line`? If the line is longer than the size of the header, `exec` would miss the trailing part of the line, so better fail. This is a tradeoff for simplicity, as `exec` could perfectly keep on reading until a whole line is read.

4.11 Dead processes

Processes can terminate existence in several ways. First a process can call `exits` to terminate itself (see `exits(2)`). A message can be passed to `exists`, which will be passed by the kernel to the parent process calling `wait(2)`. Thus, the concept of a process hierarchy also helps in controlling how processes went in their lives. The parent calls `wait` and receives reports about its dead children; every child tells the parent.

The message passed is more meaningful for humans than the UNIX error code, and what is actually more important, is portable to different architectures! The convention is that a null string means "ok".

Another way to (almost) terminate is by faulting, either voluntarily (see `abort(2)`) or involuntarily. Faulted processes are kept hanging around for debugging in a `Broken` state. That is better than saving a core file for several reasons: first, no more core files hanging around in the file system; second, a broken process is still "alive" and can be inspected for more than just data values, the `broke(1)` rc script can be used to locate and terminate broken processes.

Yet another way is by using the `proc(3)` device `ctl` file for the process. A write of `kill` to that file, terminates the process. This way works fine over the network, since the file can be used remotely.

4.11.1 Exiting and aborting

- `/sys/src/libc/386/main9.s:1,7`
  The entry point for the user process is usually `_main`. `_main` is an assembly stub that calls the C entry point, `main`, after doing some work for the profiling clock and the `main` arguments.

  Remember that the return value of `exec` was the profile clock? That value was "returned" to the new program.

- `main9.s:9,13`
  If `main` ever returns without calling `exits`, `exits` would still be called with the "main" string as the argument.

`sysexits()` *Terminates the process reporting a reason.*

- `/sys/src/9/port/sysproc.c:502`
  One way or another, `sysexits` is the entry point called by the process terminating.

- `sysproc.c:505`
  In case the user error string (the first argument) does not look fine, this is the string reported.

- `sysproc.c:509,522`
  If no status string was supplied, that is ok. If it was supplied, copy it to the kernel buffer `buf`. `validaddr` and `vmemchr` are used to be sure that addresses are valid. If addresses are not valid, an error is raised and `status` is set to `inval`.

- `sysproc.c:523,424`
  `pexit` kills the process; the `return` is to make the compiler happy—all system calls should return a value.

- `proc.c:732`
  The `1` as a second argument asked `pexit` to release the process memory; it is really being killed.

`sysexits`
  `pexit()` *Terminates the process.*

- `proc.c:745`
  By setting `alarm` to zero, any alarm is canceled. The `Proc` may still be linked into the alarm list. This is not a problem because if a new process reuses the `Proc` entry, and it does not use alarms before expiration of a previous alarm for this `Proc`, `alarmkproc` will find its `alarm` set to zero and ignore it. If the new process ever sets an alarm, it will be first removed from the alarm list.

- `proc.c:747,759`
  All resources cleared while the lock was held. Releasing them may take some time, so do not hold the lock for more than needed.

- `proc.c:761,770`
  Now resources are released by calling routines that decrement their reference counters; if a reference counter gets down to zero, the resource is released—perhaps causing other decrements in reference counters for structures used by the resource; e.g. the `fgrp` uses channels that are `cclosed` when the `fgrp` goes away.

- `proc.c:776`
  Kernel processes are always there, and the author does not do housekeeping for them; but user processes have parents and there is a relationship to be maintained.

- `proc.c:777,782`
  All processes have a parent. You will see what happens to a process when its parent is not there. Hint: read the panic message!

- `proc.c:784,788`
  A `waserror` but in a `while` loop. Remember that `waserror` returns false when first called to set the error label, and then it returns true when an error jumps

back to the `waserror`? The effect of the `while` is to call `waserror` again when an error happens—i.e. to restore the error label popped by `error`. That means that the process keeps on trying to `smalloc` a `Waitq` structure, no matter what errors happen. But, `smalloc` provides guaranteed allocation. What error could be raised by `smalloc`?

`smalloc` calls `tsleep` to wait for free memory if the pool is exhausted, `tsleep` calls `sleep`, and `sleep` raises an `Eintr` if the process is interrupted. So, no matter how many interrupts the process gets, `exits` will not be aborted returning to the process with an `Eintr` error; `exits` does not return, ever!

- `proc.c:790`
  `readnum` prints the number given into the buffer passed. It returns the number of characters used to print the number, or zero if it did not fit. The buffer is `wq->w.pid`, and the number id `up->pid`. So, the pid field in the message for the parent kept (in the `Waitq` allocated) is being filled up with the ascii representation of the pid.

- `proc.c:791,798`
  These calls to `readnum` are filling up the the wait message for the parent with times as said in `wait(2)`. `TK2MS` converts ticks to milliseconds and the time entry at `Treal` is used to know for how long the process had lived. Saw how the message can be understood at any architecture? Guess why?

- `proc.c:799,805`
  If a non-null (and not empty!) error string was supplied by the process, it is copied into the error message in the wait message—note how the message is prefixed with the name for the text file and the process pid. That is very important when the parent process is a shell, like rc, to let any human user know who did die. It is also important if the parent cares about who died.

- `proc.c:807,830`
  If the parent's pid (`p->pid`) does not match `parentpid`, the parent is not "associated to the child" (see `rfork`) and does not care about the wait message from this child; if the parent is broken it does not care either; and if more than 128 wait messages are queued for the parent, the author thinks that the parent does not care either. Daemon processes that fork a child per request, but "forget" to call `wait` would be able to leave an indeterminate number of wait records behind them but the 128-check enforces a 128 limit. This is yet another detail where you can see how the author tries to protect the system against buggy processes.

  To pass the wait message for the caring parent, just queue it in the parent's `waitq` (queue of wait records). If the parent cares, the number of child processes and wait records is adjusted. The `wakeup`, awakes the parent in case he is sleeping waiting for a wait record. By the way, remember that `nchild` was incremented in `rfork` only if `RFNOWAIT` was not set?

- `proc.c:833,834`
  User processes bookkeeping is complete. This code is executed for both kernel

and user processes. If the memory should not be released, the caller wants the process to hang around in a broken state.

```
...
  pexit
  addbroken()
```
Keep the process in a `Broken` *state.*

- `proc.c:670,696`
  `addbroken` moves the process to an array of broken processes, and changes the state to `Broken`. By calling the scheduler, `addbroken` will not return unless the process is set again ready; that happens when the process is really terminated (e.g. by a write of `kill` to its `ctl` file). Should this happen, `addbroken` returns and the remaining code at `pexit` would terminate the process. It is nice how the code to terminate the process is shared in this way for both processes exiting and processes aborting. When `NBROKEN` broken processes exist, the first who broke is terminated to make room for the new broken process. One thing to note is that too many broken processes are a waste because they would probably never be debugged. Perhaps for CPU server kernels it would be better to keep a `broken` structure per user using the CPU server, but the author thinks this suffices. Another thing to note is that the broken process is terminated just by placing it in the ready queue—when it runs, `pexit` will terminate the process. Just simple.

```
...
  pexit
```

- `proc.c:836,844`
  Segments released. The process' mind is going. Only when the last reference to each segment is gone, it is released. Do you think that these lines would destroy the stack segment? And the text segment?

  Although you are not expected to answer this before the chapter on virtual memory, what happens if there is an ongoing page fault on one of the segments released? How can the page fault handler ensure that the segment will not go away under its feet?

- `proc.c:846,849`
  By setting pid to zero, no child will leave a wait record because of the test at line `:815`. The `wakeup` is not for this parent process (which is dying and not in `pwait`), it would awake any process waiting in `devproc.c:561` for a child to die.

- `proc.c:851,854`
  Now that nobody is linking more wait records, release all wait records queued—the parent could terminate without calling `wait`.

- `proc.c:856,862`
  Awake any debugger waiting for us (e.g. waiting for a note to be posted for us) and dissociate from the debugger.

- `proc.c:864,866`
  After a couple of locks are taken, the state is set to `Moribund` and `sched` is called. `sched` will never return because the process is really destroyed and will not get back to the ready queue. If a bug makes `sched` return, panic. Why does the author acquire these locks here?

- `proc.c:64,79`
  `sched` calls `gotolabel` for `m->sched`, which leads to code in `schedinit`. This time, this branch is taken and the process state is set to `Dead`. After releasing MMU data structures for the current process (using the prototype page table for the current processor afterwards), the process is linked into the free process list. Releasing MMU data structures and linking the process in the free queue, requires both `palloc` and `procalloc` locks to be held. However, right now in `schedinit`, which one is the current process? There is no process. What if `lock` couldn't acquire the lock at the first attempt? What if it even called `sched`? That is why the locks are acquired while there the dying process is still alive enough for requesting a `tas` lock.

  The kernel stack is kept bound to the `Proc` (and reused by the next process using that `Proc`). After the locks are released, any other processor could pickup the `Proc` and its kernel stack could be reused. This is no problem since the current stack is the "scheduler stack" kept near `Mach`. Using a scheduler stack allows the author to step back out of the dying process while killing it.

- `proc.c:80,83`
  The current process is gone, `sched` called and it will call `runproc` to run another (existing) process.

In case you didn't notice, processes aborting, generate a fault that (as you will see) end up calling `pexit` with an indication not to release the process memory.

By the way, it would be nice not to release the process resources (`fgrp` et al.) for broken processes (at least when explicitly requesting so), so that the process could be debugged even looking at the set of open file descriptors; and perhaps it would be feasible to fix things up a bit and let the process get ready again. One simple way to do so would be to move the process into the `Broken` state before line `:747`, and to add control operations to fix up the process state and set it back to ready.

I think it's time now for you to look at figure 4.12 and see how a process changes its state. Probably you did draw a scheduling diagram while you learned how processes are born, get ready, etc. Compare yours with the one in the figure. For the sake of simplicity, I have not shown the `Scheding` state, which is used while the processes is changing its scheduling state in several places (e.g. from `Ready` to `Running` and from `Dead` to `Ready`).

4.11.2 Waiting for children

`syswait()` *Waits for dead children.*

- `sysproc.c:528`
  `syswait` is the entry point for `wait(2)`. After checking that the `Waitmsg` pointed to by the first argument resides at valid user addresses, `pwait` does the work.

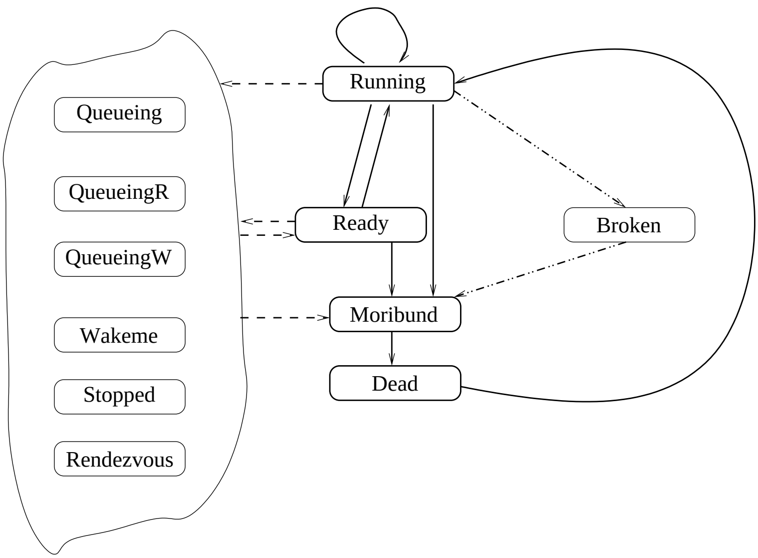


Figure 4.12: (Simplified) process state transition diagram. Do not take it verbatim: the `Scheding` state is missing, `Broken` processes appear to go right to `Moribund`, without passing through `Ready`.

`syswait`
`pwait()` *Wait for dead children.*

- `proc.c:883`
  `pwait` receives the `Waitmsg` to be filled up.

- `proc.c:888,889`
  If the wait queue is being manipulated (a child dying right now at a different processor?) just give up. Why? seems to be for avoiding deadlocks between `devproc` and `proc`. In any case, the parent is likely keep on calling `wait` until he gets the desired wait record.

- `proc.c:891,894`
  Now the lock held.

- `proc.c:896,901`
  If there is no child (dead or alive), abort.

- `proc.c:903`
  sleep until `haswaitq`; `haswaitq` returns true if the wait queue is not nil. Therefore, if there are children in the wait queue, the process will not even sleep. If all children are alive, the process sleeps until it gets awaken by a dying process, or by a note.

- `proc.c:905,909`
  Got a dead child, remove a wait queue entry.

- `proc.c:911,912`
  Nothing else to do, release the lock.

- `proc.c:914,918`
  Return extracted information by copying it to the parameter passed (if it was not nil), and returning the dead child pid.

4.12 The proc device

Although process creation is done only with system calls, processes are represented as files for most other purposes. Read the `proc(3)` manual page to learn what files are serviced by the `proc` driver. `Proc` can be mounted over the network to operate on remote processes as if they were local; not the UNIX's `/proc`, definitely. Files under `/proc/n/` correspond to views of *running* processes; e.g. two successive `cats` for `/proc/n/status` would return different file contents, because the file contents are the status of the process, and the status changes over time. Reads under `/proc/n` are used to inspect processes, and writes under `/proc/n` are used to change various things on running processes.

In the next chapter, you will learn more about file systems in Plan 9, and will be able to understand better `devproc.c`, that is where the `proc` device implements its file system. However, I think you can understand most of `devproc` now. I am going to discuss the code related to inventing a file hierarchy from the set of processes; but skipping code related just to the file system, which will be clear after the chapter on file systems.

4.12.1 Overview

- `devproc.c:10,28`
  Several Qid types defined.

- `devproc.c:54,67`
  Here you see how actual Qids are built from the Qid types defined above and the process numbers. Why does the author do this?

- `devproc.c:31,49`
  This data structure is a template for the directory serviced for each process. You can see the file names, the Qid for the file, the file length and the file mode.

- `devproc.c:757,776`
  This is the `Dev` structure linked into `devtab`. The '`p`' is the letter name for the driver, and most fields are pointers to routines used when a channel type corresponds to `devproc`. Routines with names starting with `dev` are default implementations for channel operations in `dev.c`. Many of the routines supplied by `proc`, relay on generic implementations that use the `procgen` routine to iterate through a proc directory. Let's see some of the routines now.

`open...`

  `procopen()` *Opens a* `proc` *file.*

- `devproc.c:161`
  `procopen` is the routine used when a file under `/proc` is being opened. The file to be open is represented by the `c` channel. In Plan 9, opening a file means to check permissions and prepare the file for I/O. For example, after you walk to `/proc/3/notepg`, you have to open it before writing it.

- `devproc.c:168,169`
  The `CHDIR` bit is set in Qid, therefore, a directory is being opened; rely on the generic routine for that.

- `devproc.c:171,176`
  The process slot is kept in the Qid; give the slot to `proctab` and obtain the `Proc` for the `c` channel. Remember that the file being opened is not a real file, but some aspect of a process. `procopen` is locating the process and locking it so that the process could be inspected without race conditions.

  By keeping both the slot and the kind of file in the Qid, the author knows quickly what kind of processing (and on which process) should be done given the Qid for the file.

- `devproc.c:177,179`
  The process could have died since the channel was obtained (by a walk) and the open was requested. If the process died (even if its `Proc` was reused by a different process), the PID in `Proc` will not match the PID in the Qid.

- `devproc.c:181`
  `openmode` checks `omode` for invalid bits.

- `devproc.c:183`
  Each file under `/proc/n` has a type encoded in the Qid (see lines :10,28).

- `devproc.c:184,191`
  The text file for the process is the one being opened (`/proc/n/text`). Only opening for reading is allowed, since the text is being used for executing the process. `proctext` is the routine doing the job. `proctext()` *Gets the channel for the process text.*

- `devproc.c:785,797`
  Check that the process text is still there and get a reference to the `Image` for the process text segment.

- `devproc.c:805,807`
  The image contains a channel to be used for accessing its file.

- `devproc.c:809,812`
  Increment the reference counter for the channel (someone opened it), and check that channel is still opened for reading. When the process is still there, the code is not assuming that the image is there; and when the image is there, the code does not assume that the channel is set up for reading. Why? Hint: processes are living things.

- `devproc.c:820`
  Now got a channel for reading the file for the process text segment image. It has been `incref`'ed, and will be the channel resulting from `procopen`.

- `devproc.c:193,200`
  For these files (`/proc/n/proc`,etc.) nothing has to be done but to check that the open is for reading—processing continues after the `switch`.

- `devproc.c:202,210`
  Nothing done; can be opened for read or for write.

- `devproc.c:212,216`
  For `/proc/n/ns`, mode has to be read and temporary storage for walking the mount table is allocated.

- `devproc.c:218,226`
  For `/proc/n/notepg`, only writes are allowed, and not for the first process group (boot). The id of the process group and the `noteid` for the process are kept as a Qid in the channel.

- `devproc.c:228,231`
  Defense against bugs; no other Qid types known.

- `devproc.c:233,246`
  After checking that the process is still there, the generic `devopen` routine is called. `devopen` uses `procgen` to iterate through the `proc/n/` directory searching for a file matching the channel supplied by the user (i.e. the file being opened, like, `/proc/n/ns`). Once found, `devopen` checks permissions and either raises an error or returns the channel.

`wstat...`

`procwstat()` *Updates attributes of a `proc` file.*

- `devproc.c:250,256`
  `procwstat` is used to modify file attributes, including permissions. No `wstat` is permitted on directories.

- `devproc.c:268,269`
  Only the user who started the process, and eve can change attributes in `proc` files.

- `devproc.c:271`
  `convM2D` converts the machine independent representation of file attributes (given by the caller) into a `Dir` structure, more amenable for processing. The file could be `wstat`'ed from a different machine with a different architecture.

- `devproc.c:272,279`
  If the user in the `Dir` structure (to be written) is not the owner of the process, a `chown` is being done. Only eve is allowed to do such thing.

- `devproc.c:280`
  Honor a `chmod` in the file.

`close...`
  `procclose()` *No more I/O on a `proc` file.*

- `devproc.c:337,342`
  The temporary storage allocated in `procopen` is released when the file is closed.

4.12.2 Reading under `/proc`

`read...`
  `procread()` *Reads from a `proc` file.*

- `devproc.c:363,364`
  `procread` services reads under `/proc` files. It corresponds to a call `read(f,va,n)` with the file offset set to `off`.

- `devproc.c:378,379`
  Use the generic routine for reading directory entries.

- `devproc.c:381,383`
  The process could have died since the open was done.

- `devproc.c:386,414`
  `/proc/n/mem` represents the process memory. Reading is achieved by doing a `memmove` to copy the memory being read into the buffer transferred to the caller of read. The `mem` file represents virtual memory: `offset` 0 is virtual address 0. Not all addresses are valid.

  – `devproc.c:387,389`
    Addresses before `KZERO` or within the user stack are read with `procctlmemio`, which checks that addresses are valid and lie within process' segments. Perhaps some time ago the user stack was within the kernel portion of the address space; right now, the first part of the "or" at `:387,388` is true whenever the second part is true.

  – `devproc.c:391,401`
    Addresses between `KZERO` and `end` correspond to kernel addresses and are read by a direct `memmove` without further checks.

  – `devproc.c:402,413`
    Remaining addresses correspond to memory found at the two memory banks in `conf`—also read with `memmove` by the grace of the direct map between physical and virtual memory. Remember that addresses in `conf` were updated to be kernel virtual addresses—although early when booting they were physical addresses instead.

  If you trace the kernel execution after the `open` of `/proc/n/mem`, you will see how permissions are checked using the mode kept in the `Proc` structure for the process. The author is permissive in allowing any user with permissions to read kernel memory (even though he protects the memory used to keep user keys, all memory allocated from `xalloc` can be read). However, this permissiveness is good to make it easier to debug and inspect the kernel state.

- `devproc.c:415,427`
  Profiling is not discussed now, but see `proc(3)`.

- `devproc.c:428,451`
  Copy to the user buffer the text for the first note posted for the process, and decrement the number of notes. You can see how posted notes can be read/canceled by reading this file.

- `devproc.c:452,458`
  The `Proc` structure is read. Useful to debug the kernel: No need to put more `prints` nor to attach a debugger just to see a value in `Proc`, just read it.

- `devproc.c:460,483`
  Useful!, `dbgreg` pointed to the saved `Ureg` while the process was switched out. A read here returns the user context. The code below `regread:` simply copies the memory read from the `Ureg` pointed by `rptr`. The `kregs` file corresponds to the kernel context. When the process in on its way to be switched out, there is an `Ureg` saved when last entering the kernel which holds the user register set; but then the process is really being switched out, there is a label set by the scheduler before jumping to other process' kernel label: `setkernur` sets in the "kernel Ureg" the PC and SP saved in the process scheduling label. There is no kernel Ureg, although users are told so. Remaining registers in the "kernel Ureg" are reported as zero.

- `devproc.c:485,518`
  The `status` file is invented to contain the name for the text file (`:495`), the owner of the process (`:496`) and the process state (as kept in `psstate`). If `pssate` is not set, the process state name is obtained by translating the process scheduling state `state` to a printable representation. Besides, various times and the size for the process are reported as said in the `proc(3)` manual page. To compute the size, the lengths (`top` minus `base`) for the various segments are accounted for. `NAMELEN` and `NUMSIZE` are used to pad the various pieces of status at fixed positions in the "file". That can simplify a lot the code to read a particular field of the `status` file.

- `devproc.c:520,539`
  `segment` contains a printable representation of the segments for the process. The segment array is iterated to obtain segment names, types, and boundaries.

- `devproc.c:541,575`
  The contents of the `wait` file are the next wait message from a died process. The code uses the `waitq` as you saw before for `wait(2)`. The `read` on `/proc/n/wait` would block until a child dies. If you see line `:554`, the `read` would fail if there is no children and the `read` of `/proc/n/wait` is being done by the process `/proc/n/`. That is, the parent can use read to block waiting for a dead child; other (unrelated) process can read this file to cause a "wait" for the child. Perhaps the `wait` system call could be removed in favor of reading `/proc/n/wait`.

- `devproc.c:577,611`
  Not to be discussed now, but the code synthesizes the text corresponding to

commands to reproduce the name space for the process. That text can be fed to a shell to recreate the name space even at a different machine.

- `devproc.c:613,616`
  `noteid` is simple. `procfds` is generating a text representation of open file descriptors.

`read...`
  `procread`
  `procfds()` *Reads a `proc` file descriptors file.*

- `devproc.c:286,335`
  The `Fgrp` for the process is iterated and for each open file descriptor, the channel is inspected to obtain the open mode, device type, Qid, file offset, etc.

4.12.3 Writing under `/proc`

`write...`
  `procwrite()` *Writes a `proc` file.*

- `devproc.c:661`
  `procwrite` is analogous to `procread`, but does a write instead. The write is for file referenced by `c`, and corresponds to a `write(f,va,n)` when file offset is `off`.

- `devproc.c:669,670`
  No writes on directories.

- `devproc.c:677,680`
  A write to `notepg` would post a note to the process group. You saw how `pgrpnote` did that. Remember that `procopen` set `pgrpid` in `c` to contain the `noteid`? When the user opened the file, the `notepg` file represented a concrete note group. Should the process change its note group in the mean time, the file still points to the old note group. Besides, the process is not checked for death before `pgrpnote` is called. So, imagine you want to kill all processes in a process group by posting a note to the group. Imagine that you open the `notepg` file, and then the process starts dying voluntarily; by using the saved `noteid`, the file would still cause a note post to remaining processes in the note group.

- `devproc.c:691,697`
  A write to `mem` is used to modify process memory. A debugger can use this file to update variables in the debugged process. The process should be stopped though—because it would be unpredictable what could happen if memory could be updated while the process is running. `procctlmemio` is used to operate on process memory, as happened in `procread`. If you look at the last parameter it was `1` in `procread` and it is `0` in `procwrite`; it is deciding what to do: read or write.

- `devproc.c:698,706`
  A write to `regs` updates the saved registers for the process. A debugger can

use this to update the process PC, SP, etc. (`dbgreg` points to the saved `Ureg` for the process). `setregisters()` *Updates the process `Ureg`.*

- `../pc/trap.c:734,747`
  `setregisters` copies the supplied registers into the `Ureg` for the process. Both flags and code and stack segments are ensured to be valid ones. Otherwise the user could cause a system crash or break system security.

- `../port/devproc.c:708,714`
  The same for FPU registers. In this case, the user can update all of the FPU context and no machine dependent routine is needed to ensure that a valid state remains. If the user is writing a wrong state, he would just harm himself.

- `devproc.c:716,718`
  A write to `ctl` can be used to perform control operations on the process.

```
write...
  procwrite
  procctlreq()
```
Writes a `procctl` request.

- `devproc.c:873,881`
  `procctlreq` does the job after copying the request string to a kernel buffer.

- `devproc.c:883,884`
  A write of "stop" to `ctl` leads to a call to `procstopwait` with the last parameter set to `Proc_stopme`. `procstopwait()` *Wais for a process to stop.*

- `devproc.c:828,835`
  `procstopwait` attaches the current process (`up`) as the debugger for the process whose `ctl` file is being written. To setup a debugger for process `p`, the `pdbg` field of `p` is set pointing to the debugger process, and `procctl` in `p` is set to be the process control operation. The author wrote things so that the control operation is known to `p`, and it will honor it if needed. If the `pdbg` field of the process' `Proc` was set, there was already a debugger and the call fails. If the process was already stopped, the write of `ctl` results in a non-operation.

  The write of `ctl` can be done through the network, and in that case, the debugger would be running at a different machine. So, who is the debugger process? The write request would have been sent through the network, but there is a (file system server) process in the node of `p` doing the actual write to the `ctl` file. That process would be setup as the debugger in the `Proc` structure. For the kernel, it does not matter if that is the real debugger or a remote delegate for the debugger.

- `devproc.c:838`
  The (debugger) process `psstate` is set to `Stopwait`. `sleep` will make the process wait. The `p` process `state` can still be `Running` or `Ready`, so the scheduler can elect it for running. When the to-be-debugged process runs again because the scheduler elects it, it will reach soon either the end of `trap` or the end of `syscall` in `trap.c`.

```
trap
    notify
    procctl()
```
Checks for process control operations.

- `../pc/trap.c:310,313`
  If it is `trap` the first one to notice, it would call `notify` when noticing that `procctl` is set.

- `trap.c:535,538`
  `syscall` would do the same.

- `trap.c:552,553`
  `notify` checks for notes, but it calls `proctl` too–when it sees that a control request is pending.

- `../port/proc.c:1096,1098`
  If the control operation is to terminate the process because it consumed too much physical memory, do so. The process is killing itself upon request.

- `proc.c:1100,1102`
  If the process is being killed, do so.

- `proc.c:1104,1107`
  This is for tracing processes, you can ignore it now; although you can see how it does the same of `Proc_stopme` when the process has pending notes.

- `proc.c:1109,1126`
  The process stops itself voluntarily. The "ps" state is updated to reflect that the process already stopped. The scheduling state (`p->state`) is set to `Stopped`. The call to the scheduler switches to a different process and the debugged one will not run again until it is set `Ready` (by the debugger). The local `state` is used to resume the process in the state it was before being stopped, because it could be a different thing each time the process is stopped. Before discussing the `wakeup` call, note that interrupts were disabled since `notify` was called. That makes sense since it messes up the user stack and besides it can stop the process via `procctl`.

- `proc.c:1116,1119`
  If there was a debugger waiting for the process to stop, wake it up. The debugger expects the write to `/proc/n/ctl` not to return before the process is stopped.

- `devproc.c:844`
  So the debugger sleeps until the process is stopped. If the `wakeup` runs before the `sleep`, the `procstopped` function will notice and the debugger will not even sleep. Otherwise the debugger sleeps until the process stops itself and notifies the debugger.

- `proc.c:1118`
  One more thing, `pdbg` is set to nil when the operation is done. `pdbg` is used to let `p` know who is its debugger (so it could awake it, etc.), but as soon as `p` does not need to know who is its debugger, the "connection" is reset. `pdbg` acts as a

lock in that if a debugger is already (waiting for) stopping the process, no other debugger would be allowed to do so. Once the process is stopped, a different debugger process can operate on the process.

- `devproc.c:847,848`
  The debugged process could die due to a note post or a control operation.

```
write...
  procwrite
  procctlreq
```

- `devproc.c:885,900`
  Back to `procctlreq`, a write to `ctl` with a "kill" string would kill the process. Should the process be broken (did fault), `unbreak` sets it `ready` again.
  `unbreak()` *Terminates a broken process.*

- `proc.c:699,713`
  It does so by scanning the broken process array and setting as `Ready` the one passed as a parameter—the array is updated to reflect the deallocated entry.

  Although it may look silly to scan the array instead of using `p`, remember that at most `NBROKEN` processes are kept broken. In general, processes must be either running, linked into a ready queue, or linked into the data structure that prevents the process from being ready. In this case, `broken` does the job.

- `devproc.c:891,895`
  Should the process be stopped, it is killed by the system. The `Proc_exitme` control operation will be handled later by `procctl`. The process is set back into `Ready` state so it could run and kill itself.

- `devproc.c:896,899`
  In any other state, the process will either be `Ready`, or get back to `Ready` if it was sleeping or doing a rendezvous. So just post the note and setup the control operation.

- `devproc.c:901,906`
  `hang` requests that the process stops when doing an `exec`. Just update the flag accordingly. It is honored by `exec`, which uses the `Proc_stopme` control operation to stop the process doing the `exec`.

- `devproc.c:908,909`
  A write of "waitstop" uses `procstopwait` again to stop the process. Unlike the previous usage, `ctl` is now zero, which means that no control operation is posted (`:833,834`). So, the writer of `ctl` would sleep until the process is stopped, but it does not stop the process: it waits until the process stops. For example, a debugger process may write "hang" to `ctl` and then `waitstop`, to wait until the debugged process does an `exec`.

- `devproc.c:911,917`
  A write of "startstop" resumes an stopped process (sets it `ready`) and then waits until the process stops. Although `procstopwait` would set `procctl` to

`Proc_traceme`, it is set by hand before allowing the process to run; otherwise the process could be free running before honoring the `Proc_traceme` operation.

- `proc.c:1104,1107`
  The `Proc_traceme` operation is handled by the started process like a stop one, but only when the started process has a note posted. For instance, a debugger may set a breakpoint and let the process run until it reaches the breakpoint or faults. Whenever that happens, `procctl` would not return at line `:1106` because there are posted notes, and the process will stop after awaking the debugger process.

- `devproc.c:919,923`
  Just set ready an stopped process.

- `devproc.c:925,936`
  `procctlfgrp` closes all open file descriptors. I don't know what this control operation is really for, but it could be useful if other control operations allowed file redirection to be done by means of `proc(3)`.

- `devproc.c:938,949`
  A write of "pri N" to `ctl` would set a new base priority for the process. Only eve is allowed to raise priority this way—it is her machine, isn't it? Can you see the difference between the "root" user on UNIX and "eve" in Plan 9?

- `devproc.c:951,956`
  To wire a process to a processor number. You already saw how `procwired` does it.

- `devproc.c:958,968`
  A write of "profile" to `ctl` is clearing the profile information, which hangs from the text segment `profile` field.

4.12.4 A system call? A file operation? Or what?

Let's look briefly at how the user C library implements some services using the system calls and system services that you now know. I hope you will be reading more of that library as you learn how the kernel works.

`abort()` *Aborts execution.*

- `/sys/src/libc/9sys/abort.c`
  Just crosses a null pointer. The process gets a page fault and enters the `Broken` state.

`fork()` *Creates a new process.*

- `fork.c`
  Just calls `rfork` asking for a new process, with a copy of the file descriptor and rendezvous groups.

`postnote()` *Posts a note.*

- `postnote.c`
  Writes to `/proc/n/note` or to `/proc/n/notepg`.

`getenv()` *Get the value of an environment variable.*

- `getenv.c`
  Just read the file "`/env/`*name*".

`getpid()` *Gets the process it.*

- `getpid.c`
  Just read the file "`#c/pid`"—see `cons(3)`; more on the next chapter.

Can you see how even most of the user utility functions are actually using file operations?

Chapter 5

Files

File systems are central to Plan 9. Remember that the key point is that everything is a file and files can be accessed over the network. In section 3.11, “Files and Channels”, you already learned a bit about files and channels. You should reread that section if you forgot it and then continue with this chapter.

In this chapter you are going to read the implementation of the various system calls related to files, including `bind`, `chdir`, `close`, `seek`, `dup`, `open`, `read`, `create`, `fd2path`, `remove`, and `wstat`. Besides, as an example of a file system, you are going to revisit kernel devices (e.g. `pipe`), looking at the generic routines provided in `dev.c`. Finally, the device translating calls to file procedures into RPCs, `mnt`, is also discussed in this chapter.

During this chapter, you will be reading these files:

- Files at `/sys/src/9/port`:

  `sysfile.c`
  : File system calls.

  `chan.c`
  : Channels.

  `cleanname.c`
  : Name cleanup.

  `portdat.h`
  : Portable data structures.

  `pgrp.c`
  : Name spaces.

  `dev.c`
  : Generic device routines.

  `devpipe.c`
  : Pipe device driver.

  `devmnt.c`
  : Mount driver (remote files).

`cache.c`
Caching remote files.

`qio.c`
Queue based I/O.

. . . and several other ones used as examples.

5.1 Files for users

Users operate on files using the system calls provided at `sysfile.c`. As you already know, such system calls are serviced by using channel operations. Let's describe now how such system calls work, without looking too much into the channels. I hope you get a better image of what is going on after seeing how your system calls are translated into channel operations. See `open(2)`, `dup(2)`, and `close(2)` manual pages.

`sysopen()` *Entry point for the* `open` *system call. Prepares a file for I/O.*

- `/sys/src/9/port/sysfile.c:221`
  Users operate on files by obtaining file descriptors using the `open` (also `create`) system call. `sysopen` is the entry point for `open`.

`openmode()` *Checks the open mode for a file.*

- `sysfile.c:226`
  `openmode` is called with the second argument. It does a cleanup of the `omode` parameter for open and returns a clean `omode`. In this case, the cleaned mode (the return value) is not being used. The author does this call to let `openmode` raise an error in case the open mode is not valid.

- `sysfile.c:227,231`
  Cleanup the channel for the file being opened on errors.

- `sysfile.c:232,233`
  `namec` opens the file and returns a channel for the file name given. It resolves the name in the current name space; `namec` is discussed below.

- `sysfile.c:234,236`
  A new file descriptor allocated to the new channel.

- `sysfile.c:237,238`
  The descriptor points to the channel used. `open` is done.

`sysopen`
`newfd()` *Installs a new file descriptor for the given channel.*

- `sysfile.c:46,65`
  The `Fgrp` contains an array of pointers to channels. File descriptors are simply indexes into this array. For example, if file descriptor `3` is open, `fd[3]` would point to the channel for the file. After locking `fgrp`, the array of pointers to channels is searched for a nil entry. The first entry unused (`:54`) is selected.

Lines `:56,59` allocate new entries in case all entries in the array are being used. `maxfd` records the end of the used part of the array, and the new allocated file descriptor is set to point to `c` at line `:62`.

`sysopen`
  `newfd`
    `growfd()` *Resizes the file descriptor set.*

- `sysfile.c:13,43`
  When the descriptor array is exhausted, `growfd` resizes it, `DELTAFD` new entries at a time. The routine does nothing if the array is already big enough to hold the descriptor desired (`fd`); it can be used just to check that `f` is big enough (`nfd` is the number of entries, used or not, in the array). The array will never contain more than 100 entries going from 0 to 99 . A good reason to limit the number of file descriptors to 100 is that nobody could allocate a big amount of file descriptors to exhaust kernel memory. Another good reason is that this limit can convince users to close unused file descriptors, which would also release resources on the file servers involved. Lines `:27,31` are double checking that the number of allocated descriptors is kept under a reasonable value, for the same reason. Note also the use of `malloc`, and not `smalloc`. If no more memory is available, the `open` will fail, and the process will not sleep waiting for memory.

`sysdup()` *`dup`(2) entry point. Duplicates a file descriptor.*

- `sysfile.c:180`
  `sysdup` is used to duplicate a file descriptor. After `dup`, two file descriptors point to the same channel.

- `sysfile.c:189,190`
  `fdtochan` returns a reference to the channel given the file descriptor. `c` holds the channel for the file descriptor being duplicated, and `fd` holds the file descriptor number where the user wants to place the duplicate.

- `sysfile.c:191,205`
  The user specified where to place the duplicate. `growfd` ensures that the entry for the descriptor exists in the array. `maxfd` has to be updated in case the new `fd` is the biggest one. Finally, the channel is linked into the descriptor entry. The dance with `oc` is to close the previous channel in case the “duplicate descriptor” was already opened. The channel `c` has now another reference (at `fd`).

  Can you find out where is the `incref` for the channel?

`...`
  `fdtochan()` *Gets the `Chan` for a file descriptor.*

- `sysfile.c:68`
  `fdtochan` is used wherever the kernel wants the channel given a file descriptor specified by the user.

- `sysfile.c:77,80`
  Has the file descriptor a channel? `nfd` is checked before `fd[fd]` to be sure that the entry exists.

- `sysfile.c:81,82`
  If the caller of `fdtochan` said `iref`, a new reference is added. This is where the reference was added for the duplicated channel in `sysdup`.

  The reference is added while `f` is locked, so that the channel does not disappear even if someone else is releasing the `Fgrp`.

- `sysfile.c:85,89`
  Ignore this by now.

- `sysfile.c:91,104`
  If the caller gave a valid `mode` to specify that the channel is going to be used according to `mode`, check permissions. If `mode` is `-1`, or if the channel mode allows everything, no check is done (the kernel can pass `-1` to `fdtochan` to request the channel no matter what is going to be used for). Otherwise, the checks ensure that a read only channel is not used to truncate a file and that whatever be mode (read or write), the channel has the bits on.

`syscreate()` *`create(2)` entry point. Creates a file and opens it.*

- `sysfile.c:735,753`
  `create` creates a new file and returns an open file descriptor for it. It is similar to `open`, but note how `Acreate` (and not `Aopen`) is given to `namec`; and how the last argument of `namec` specifies the permissions for the new file. Besides resolving the path given by the user, `namec` would create the file and return a new channel for it. Probably it would be good to use a single system call for both `open` and `create`—even though users could still have to different routines to prevent accidents.

`sysremove()` *`remove(2)` entry point. Removes a file.*

- `sysfile.c:755,776`
  `remove` removes a file (be it a file or a directory). It uses `namec` to get a channel for the file, and then it calls the device specific `remove` function, which removes the file.

  Removing a directory does not require that the directory be empty, it depends on what the file server implementing the directory wants to do (e.g. see `upas/fs` in `mail(1)`).

`sysclose()` *`close(2)` entry point. Closes a file descriptor.*

- `sysfile.c:271,277`
  Users close their file descriptors using `close(2)`. `sysclose` is the entry point for that. `fdtochan` is called, but the channel returned is not used. If the file descriptor is not valid, `fdtochan` will raise an error and `sysclose` would be done. Otherwise, `fdclose` is called to close the open file descriptor.

`sysclose`
  `fdclose()` *Closes a file descriptor with the matching flag.*

- `sysfile.c:242`
  `fdclose` closes `fd`.

- `sysfile.c:248,254`
  First, the channel (`c`) for the file descriptor is obtained. After `fdtochan` and before line `:248` another process could have closed the file descriptor, so `c` has to be checked to be still there. The call to `fdtochan` not only checked that the file descriptor was open, it also checked that the entry in the array was allocated, so the author can index on it safely—the array can grow but it does not shrink.

- `sysfile.c:255,260`
  The flag given to `fdclose` is checked against the channel flags. If the `flag` is not set in the channel, the routine returns—without closing the descriptor!

  That is useful to close only those descriptors that have a flag set. Saw how the author writes routines that can be used in more than one way? Did you read “The Practice of Programming”?

- `sysfile.c:261,267`
  The first line closes the descriptor, the last line drops the reference to the channel (which is actually “closed” if this was the only reference for it). Besides, if the descriptor closed is the biggest one, `maxfd` is updated to mark the end of the used part in the file descriptor array.

But for `namec` and `cclose` you already know how file descriptors are added and removed from the process `Fgrp`. You also know how are new `Fgrps` allocated, duplicated, and deleted when processes are created and deleted. So, what remains for you to understand what the user sees of the file system (names and descriptors) is to discuss name spaces.

5.2 Name spaces

In Plan 9, every process has a name space. Well, every process group has a name space. Usually, processes sharing a “session” share their name space. For instance, every `rio` window starts usually with a new name space, which is a copy of the name space for the process that started the window.

The name space is simply a mapping from names to files (actually, from names to channels to files) that can be adjusted to alter the set of existing mappings. Name spaces are implemented in `/sys/src/9/port/pgrp.c` (because each process group has its own name space) with tight cooperation from the implementation of channels in `chan.c`. To understand name spaces, it is crucial to remember that every `Proc` has a `Pgrp` (name space), as well as a `slash` channel (root directory) and a `dot` channel (current directory). To understand the implementation of name spaces, I think it is better to see how are names resolved; then you will understand better the code used to customize the name spaces.

As you will see through this section, name spaces can be customized by using `mount(2)` and `bind(2)` to add new entries. Figure 5.1 shows an example of this. You should read the paper “The Use of Name Spaces in Plan 9” [12] from volume 2 of the manual if you are confused.

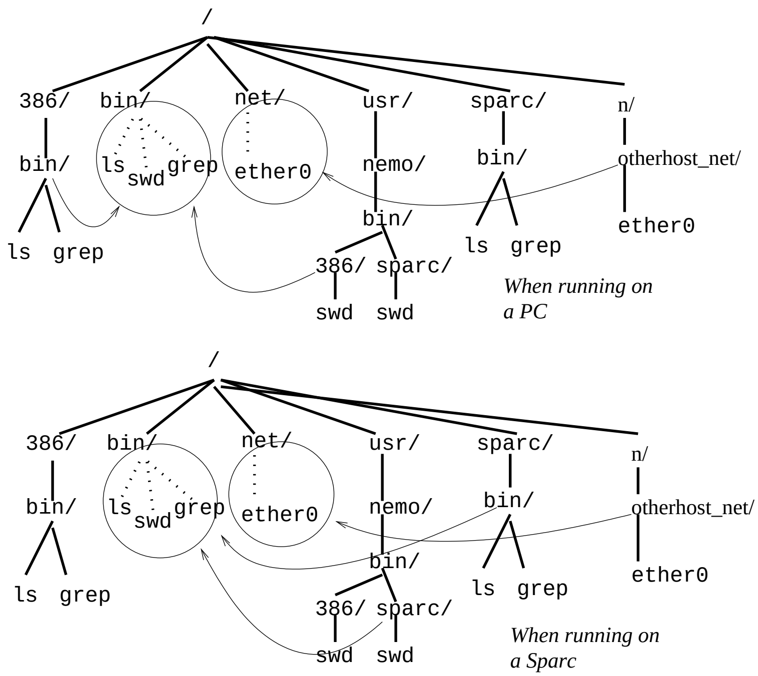


Figure 5.1: Name spaces for a couple of processes. Note how they differ. Each name space has in `/bin` appropriate binaries for the architecture used. Besides, the file used for the ethernet device seems to come from a different machine.

5.2.1 Path resolution

`namec()` *Get the channel for a given file name.*

- `chan.c:630`
  `namec` translates a name into a channel, using the current name space. It receives an access mode that shows what will be done with the file (e.g. create it, open it, etc.) as well as an open mode that specifies whether the caller is going to open the file for reading, writing, etc. `Perm` is used to initialize permissions for files being created.

- `chan.c:641,642`
  `namec` is called with paths that come from user code; don't trust them and check that they are at least a non-empty string. If the caller of `namec` "consumed" all the path, there is no such file.

- `chan.c:644,651`
  The (virtual) address is not bigger than `KZERO`, which means that it is an user-supplied path (not a kernel supplied one). So, verify that the memory from `name` to the end of the string is valid; note how `BY2PG` is used to do only one check per page (perhaps this could be embedded into `vmemchr`?).

- `chan.c:653,658`
  In `cname`, the author keeps the name for the file pointed to by channel; release it on errors. You will understand soon the reason for keeping names on channels. Can you guess it now? Hint: consider how symbolic links mix with `cd` on UNIX.

- `chan.c:660,663`
  Names are resolved differently depending on the first character of the path supplied. The `switch` is just selecting the starting point for resolving the name.

  Names starting with `/` are resolved within the `pgrp`, but starting at `slash`; names starting with `#` are resolved within the kernel driver name space (they are kind of absolute names for kernel devices); everything else is resolved starting at `dot`. Once the starting point is selected, a name can be resolved by iteration, resolving one path component at a time. Lines `:660,726` are setting up things so that lines `:728,732` could iterate to resolve the entire path. But how are things set up?

  Callers of `namec` usually want to know what is the name (without any previous path; just the file name) for the file just resolved; e.g. when executing a new program, `up->text` is set with the file name (`ls`) for the file (`/bin/ls`) being executed. The author sets `elem` pointing to the `elem` field of the current `Proc`. As each path component is resolved, `elem` is updated to contain the path component. So, the caller of `namec` can later use `up->elem` to recover the file name.

  The author sets `mntok` to reflect whether a mount operation is allowed in the file name being resolved or not; by default, it is allowed. More on that later.

  `isdot` is set whenever the path being resolved corresponds to the current directory. By default, assume it does not.

- `chan.c:664,665`
  Resolving an absolute path (starts with "`/`"). `newcname` creates a channel name from a string; so `cname` is set to keep the given path.

  `newcname()` *Creates a channel name.*

- `chan.c:108,122`
  A `Cname` is used to share names among channels with the same name. It is reference counted. The actual memory allocated is `CNAMESLOP` (20) bytes more than needed to hold the path. That seems to be to permit changes in the name that increase the path slightly. The `ncname` counter is used to keep track of how many channel names there are; If the author sees that the number of names is close to the number of channels, there is no point on sharing channel names. The path length is kept within `Cname`. Although it could be recomputed by calling `strlen`, the author prefers to call `strlen` just once, and reuse the computed length whenever it is needed.

- `chan.c:666`
  So, what follows the "`/`"?

  `skipslash()` *Advances a name past the prefix slash (if any).*

- `chan.c:862,872`
  `skipslash` returns a pointer past the `/`. The path could be `//xxx`, and `skipslash` would skip all the adjacent slashes. Users seldom write `//`, but programs do; just define a variable `v1` to be `/a/b/c/`, and then add a relative path `v2` of the form `x/y`: you get `/a/b/c//x/y`. The system should cope with that. Lines `:867,870` remove any "`.`" component, so that file servers do not see unnecessary "`.`" names. The system is replacing paths like "`/./a`","`./a`", and "`/a/.`", with "`/a`","`a`", and "`/a`" respectively. By understanding "." here, this code does not need to be duplicated at every file server in the system, and what is better, the meaning of "." can be kept consistent across file servers.

  I lied a bit, file servers can still see "." as a path. Keep on reading.

- `chan.c:667`
  Iteration to resolve an absolute path should start at `slash`, so get a clone of the `slash` channel for the current process. A clone is needed because the iteration used to resolve the path will "move" (actually, `walk`) the channel to point to each file/directory along the path name. If the author used `up->slash` to `walk`, the root directory for the process would change.

  After this line, `cname` is the entire absolute path, `name` points to the first component name (after the slash, once any dot has been removed), and `c` is a channel pointing to the root directory for the process. Besides, `mntok` and `isdot` are set appropriately. Although `elem` has not been set, `nextelem` will take care of that later.

- `chan.c:669`
  The name is referring to a kernel device path (e.g. "`#SsdC0`" to specify the disk `sdC0` from the `sd` device). Kernel device paths allow you to use devices even

though you may have an empty name space (Remember the implementation of the `getpid` function in the C library?)

- `chan.c:670,679`
  As with absolute paths, the channel name is the supplied name. Mounts were allowed for absolute paths, but not for kernel device paths. The author wants kernel paths to remain the same, so `mntok` is set to zero. Besides, `elem` is set to contain both the initial "`#`" and whatever follows until the next slash. The `n<2` check would copy the slash in "`#/`", which is the name for the `root` device's root directory. So, `elem` is set to contain the file name.

- `chan.c:680,697`
  When you use a kernel device path, you are actually "attaching" to (mounting) the file tree serviced by the kernel device to your name space. Once it is attached, you can resolve path components within the device's file tree.

  Lines `:692,695` are extracting the first "rune" (e.g. character for Spanish, but something different for a Japanese) and looking if it is an "M" (`utfrune` is an `strchr` for runes). If it is an "M", the path is "`#M`", which corresponds to a mount driver path (see `mnt(3)`). The mount driver is the one issuing RPCs for remote files when you perform a file operation on them. If users could attach to mount driver paths, they could bypass file permissions. Imagine that a server checks credentials from a client when the server file system is attached to the client name space. Once the client has been allowed access, another process could try to "borrow" some files serviced by the mount driver on behalf of the previous process. By denying attaches to mount driver path names, the only way to get files from the mount driver is by attaching to (and authenticating with) a remote file server.

  Lines `:696,697` just check that the `Pgrp` does not have the `noattach` attribute set, which would prevent attachments. This flag can be used to prevent a process from acquiring more files than found on its name space. If you don't trust a program too much, you can build a name space where the program cannot hurt anybody else, and set the `noattach` flag. These lines forbid attachments when `noattach` is set and the device is any of "`#|`", "`#d`", "`#e`", "`#c`", or "`#p`"; see the comment to learn which devices they are. The comment says that it is okay to allow attachments on these devices (i.e. if `r` is contained in "`|decp`", `noattach` should be ignored). However, that would need a *not* ('`!`') before `utfrune`. I think that the comment is right, and the `if` condition is missing a "not".

- `chan.c:698,700`
  `r` holds the first character after the '`#`'. That character identifies the kernel device. `devno` returns an index into `devtab` for the given device character. The `1`, is to tell `devno` that it is the user the one specifying a device name; it is okay if the user is mistaken: the device may not exist. If a `0` was given, `devno` would `panic` if the device is missing—that would be a kernel bug.

- `chan.c:702`
  The initial channel to resolve a kernel device path is the channel obtained by "attaching" to the device file tree. That channel points to the root file for the

device. `c` is setup properly now. The channel does not need to be cloned because it is a brand new channel.

- `chan.c:703`
  Once the initial name is processed (the device name), advance to the next one; as the author did after processing the "/" for absolute paths.

- `chan.c:705,707`
  Must be a relative path. So, the name for the channel is the name for the current directory followed by the relative path supplied by the user. `up->dot->name->s` is the C string in the `Cname` for the `dot` directory of the current process. Once `cname` has a `Cname` built from the current directory name, `addelem` adds, as a suffix, the relative path.

  `addelem()` *Adds a suffix to a name.*

- `chan.c:137,148`
  If the name is shared between several channels, make a new copy so that adding a suffix to the name will not change the name of other channels using the shared copy. For instance, when a channel is cloned, the name is shared among the clones; if a clone changes its name (e.g. because of a `walk`), other clones should be kept untouched.

- `chan.c:150,158`
  More space is allocated to hold the suffix (`s`) and the small extra space.

- `chan.c:159,160`
  The new name is *prefix*+/+`suffix`, unless *prefix* was really "`prefix/`" or *suffix* was "`/suffix`".

- `chan.c:708,711`
  The starting point to resolve the path is `dot`, get a clone of the `dot` channel in `c`—the clone is needed for the same reasons it was needed for absolute paths. The call to `skipslash` would simplify things like `./x` and `.//x` down to `x`. If `name` is the empty string after simplifying the path, `name` referred to the current directory, so the author sets `isdot`.

  When `isdot` is set, there are no further `elems` in the name; this case is handled as a special case by the code following.

- `chan.c:714,717`
  Starting to walk on the channel, close it on failures.

- `chan.c:719`
  `nextelem` obtains the next "component" name in the remaining path name, placing it in `elem` and advancing `name` past the element; the advanced `name` is returned, hence the assignment.

  `nextelem()` *Get the next element from the name.*

- `chan.c:903,926`
  `nextelem` is called after `skipslash`, so there is a bug if the first character is

a slash. If there are no more slashes, the whole name is the next component name. The loop copies the component name to `elem` (`up->elem`), one rune at a time. `isfrog` contains characters that cannot appear as a component name. Looks like Rob Pike decided to try allowing blanks in component names. After the element name has been obtained, `skipslash` does more "dot" and "slash" cleanup and advances to the next component name.

For paths like "`/`" and "`.`", `nextelem` would set `elem` to be the empty string and it would advance `name` up to the end.

- `chan.c:721,726`
  `mount` is discussed later.

- `chan.c:728,732`
  The heart of path name lookup. For each component, `walk` updates the channel to refer to the next directory/file in the path. `nextelem` advances the name for the next component (and cleans it up), and `walk` updates the channel (receives a pointer to it).

  You now see that because the channel moves down the path, `slash` and `dot` had to be cloned.

  When the loop finishes, the name has been resolved, but for the last component! In `/x/y/z`, `c` would then correspond to `/x/y`, and `elem` would be `z`; `name` would be an empty string after `nextelem` extracted `z` to `elem`.

  It is important to stop before the last component because it could be that it is a file to be created, and it would make no sense to walk to it. When the path is "`.`" or "`/`", the whole path was resolved when looking up the initial directory for the iteration. In this special case, the loop does not execute because `name` was already exhausted.

- `chan.c:734`
  How to resolve the last component depends on what is the channel for. That is why `amode` is used.

- `chan.c:735,742`
  `Aaccess` means that the file is checked (for existence, attributes, etc.) The author `walks` to the final component. For the "`.`" special case, no walk is done, since the whole path was already traversed (ignore `domount`). For the "`/`" special case, `isdot` is not set and `walk` is called; however, `elem` is the empty string and `walk` would notice and return without doing anything.

- `chan.c:744,753`
  The name refers to a directory the user is trying to get into. The author `walks` to it, and checks that it is actually a directory. For the special case, nothing is done but to check the directory flag.

- `chan.c:755,761`
  The file is being opened. Directories can be opened too. Forgetting about `domount`, in the normal case, just `walk` to the file being opened; in the special case, no walk is really done (not called, or it does nothing).

The author used an `else` instead of the `break` at line `:738` because more things have to be done by continuing after the `Open` label.

- `chan.c:762,778`
  If you remember `sysopen`, a channel was obtained for the given path, but where was the file opened? Here is where the file is opened. `c` points to the file to be opened. So, `c->type` is used to obtain from `devtab` a pointer to the `open` routine—how to prepare a file for I/O depends on who is implementing the file. The "close on exec" bit is removed from the open mode supplied to `namec`; the file server does not care about it. `open` may return a different channel than the one for the file being opened. That is useful, and allows the driver to "generate files" when other files are opened, like when you open a `clone` file. In that case, `c` already contains its own `Cname` and the `cname` created before is unnecessary; the author sets `newname` to reflect that.

  Bits to flag the channel as close on exec, and remove on close, are kept in the channel flags (remember that channels with `CCEXEC` are closed on exec?).

  Guess what is `saveregisters` doing? Hints: the compiler saves registers when doing a procedure call (so that the called procedure could use registers at will); if an error is raised and you get back to line `:714`, which `c` are you closing?

- `chan.c:780,788`
  Mount is not discussed yet, but note how in the special case nothing is done and, in any case, `c` would be pointing to the file where the `Amount` is to be performed.

- `chan.c:790,792`
  The file is being created; remember that in the normal case,`c` points to the directory where the file is being created at. If the file name was "`.`", forbid creation. (See below for the "`/`" special case).

- `chan.c:794,800`
  Get a clone for `c`; read the comment. `Clwalk` does both a clone and a walk for the channel; it is not used from the kernel (see `clwalk(5)`). By getting a clone before walking, there are no race conditions regarding `clwalk`—otherwise one of the `walk/clwalk` could fail because both of them arrived to the file server.

`nameok()` *Name contains valid characters.*

- `chan.c:805`
  `nameok` checks that only valid characters are in the file name (using the `isfrog` array). The `0` means that "/" is not ok. `nameok` is also used to verify that a whole path is valid, in which case "/" would be considered to be a valid character.

- `chan.c:806,812`
  If the walk succeeds, the (cloned) channel points to the file to be created; and the file already exists! `c` points to the directory containing the file and is no longer needed. By setting `OTRUNC` and going to `Open`, the file is opened with truncation to zero bytes—achieving the same effect of create. For the "`/`" path,

walk would succeed and "/" would be opened with truncation, which would typically lead to an (`Eperm`) error.

- `chan.c:813,814`
  The `walk` failed (did not raise an error, but failed because of its return code). That means that the file does not exist. The cloned channel is closed, and `c` will be used to create the file.

- `chan.c:819,820`
  Forget this by now. It is just ensuring `c` is the appropriate directory for creating a file; assume it is and `createdir` is not executed.

- `chan.c:822,832`
  When `syscreate` is called, the caller expects that the file would be opened and empty after `syscreate` returns. The algorithm would be simply "if the file does not exist, create it; else truncate it". However, things can change during the algorithm. Just in case file creation fails, assume that it could be because another process created the file (after the failure of the previous `walk` and before the `create` operation was executed). That can happen more easily here than on a centralized system because the latency of file system operations while going through the network is not to be underestimated—see figure 5.2.

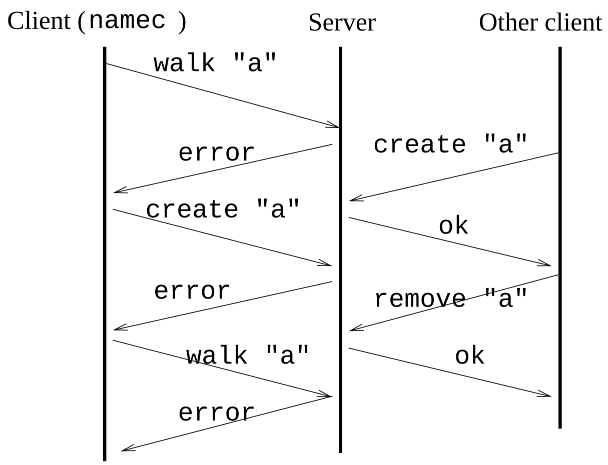


Figure 5.2: Open/Create races. Not so probable...

Should the `create` fail, try once more to `walk` to the file. If the file was indeed created, the `walk` in the error handling code will succeed and `namec` behaves as if the previous `walk` succeeded. If this second `walk` fails, it makes no sense to try over and over to "create it if `walk` fails" and "walk to it if `create` fails". A real fix could be done by folding the open and create operations into a single one, so that the file server could hold a lock for the file while deciding whether to create the file or truncate it.

- `chan.c:833`
  Here is where the actual create is done. `c` is the directory where the file is being created and `elem` holds the file name—special cases were dealt with before. The final argument `perm` is used now to establish initial permissions for the file.

- `chan.c:834,839`
  Close on exec and remove on close noted in the channel.

- `chan.c:847,852`
  `newname` was set to true before starting to resolve the path. It is only set to false after `Open`, when the channel returned by `open` was not the one supplied. Only when `open` returns an already constructed channel, the name in the channel is "old". Otherwise, the name for the channel has been built from the user supplied path, and perhaps from the current directory name. Thus, most of the times, the channel name is a new one built by `namec`. Should the channel name be new, `cleancname` does some cleanup on it, and it is set as the new name for the channel—after releasing the previous name in the channel, if any. Should the channel name be an "old" one, the new `cname` just built is not used and has to be released.

  By the way, how does `cleancname` work?

  `cleancname()` *Cleans a channel name.*

- `chan.c:604,623`
  `cleancname` calls `cleanname` for both device paths and other paths. However, for kernel device paths, only the portion of the part after the slash (e.g. "`/a/b`" in "`#S/a/b`") is handed over to `cleanname`. As `cleanname` may leave a final slash for directories, it has to be removed (`:619`) for all kernel device names but for the root driver, whose name is `#/`.

  `cleanname()` *Cleans a file name.*

- `/sys/src/libc/port/cleanname.c:9,52`
  The code looks more complex than it is. Although you should try to implement the algorithm yourself if you think the code could be easily written in a more compact way. It does several things: removes duplicated slashes, removes any "`.`" (but for the case when the path is just "`.`"), and simplifies "`b/..`" by removing both. The "`..`" in "`/..`" is removed because the parent of the root directory is the root directory by convention. The path may contain "`..`" when no simplification can be done (as in "`../x`").

  Try to understand the code yourself after reading carefully the comment. If you get lost, exercise the algorithm with several paths.

  By the way, the author put much effort into name cleanup at several places, as you saw. Although that can be worth just to keep cleaner names, there are good reasons for this effort, as you will see later.

5.2.2 Adjusting the name space

Name spaces are adjusted (on a per-process basis) by using `bind(2)` and `mount(2)` system calls. Read their manual pages now.

Both system calls are essentially the same thing: They modify the name space so that a path will lead to a different file tree. The difference between them come from where is the "different tree" located. For `bind`, it is a portion of the file tree the process sees, whereas for `mount` it is a file tree serviced by a different process. In what follows, unless I explicitly tell otherwise, I use the term "mount" to refer both to "mount" and to "bind". Besides, I use the term "mounted file" to refer to either a file or a directory which is either bound or mounted; I use the term "mount point" to refer to either a file or a directory where either a file or a directory has been bound or mounted. Got confused? read `bind(2)` and reread this paragraph.

The name space structure is mostly kept under the `Pgrp` structure for the process.

When a directory is being mounted (or bound) onto an existing directory, it is feasible to keep both the previous and new contents in place. That is, by mounting a new directory onto an existing one, you can "add" the contents of the new one to the existing one. That is done often with `/bin`, where new "binaries" are added by mounting other directories like `/386/bin`, and `/usr/$user/bin` onto `/bin`. When a file is looked up later in `/bin`, it can be found at any of the mounted directories. A directory where several other directories are mounted is called a *union* in Plan 9.

When a new directory is mounted onto a union, the user requesting the operation can specify where to place the new directory within the union. That is important because to lookup a file on a union, each directory mounted in the union is searched in order until the file is found. Flags like `MAFTER`/`MBEFORE` request that the new directory be added after/before the previous ones in the union. For example, depending on the flag, `/bin/cat` may be either `/386/bin/cat` or `/usr/$user/bin/cat`. Another useful flag is `MREPL`, which dictates that the previous directories be omitted from the union so that the new one replaces previous contents.

Now, consider a file being created on an union. On which one of the mounted directories is the file created? In the example, is it created in `/bin`, in `/386/bin`, or in `/usr/$user/bin`? The author added an `MCREATE` flag that can be given to any mounted directory. Any new file is created in the first directory mounted at the union that has the `MCREATE` flag set.

You now have an overall picture of how mounts work. But before reading how `mount` and `bind` work, it is better to see what is the final effect of a `mount`/`bind`. To do so, let's look at what does `namec` to resolve a path taking into account mount points.

Walking mount points

`namec()` *Get the channel for a given file name.*

- `chan.c:630`
  `namec` resolved a name into a channel by walking a file hierarchy. Now you know that, at some point, a new file tree may be bound to the file tree, and `namec` should walk through the mounted tree.

- `chan.c:725,726`
  Before this point, the initial channel for the iteration has been set to be `c` and `elem` contains the name for the first component to resolve. Now, unless the path is for a kernel device (`!mntok`), the path corresponds to the current directory,

or the path is not being resolved to mount the root directory (`elem` is empty), `domount` is called. The channel to start the iteration is the one returned by `domount`, which may be `c` or may be not.

`namec`
`domount()` *Process mount points for a channel.*

- `chan.c:392`
  `domount` is just "doing" the effect of a mount on a file. It receives a channel for a file, `c`, and takes care of what channel should be used instead when something has been mounted on `c`. The processing for a mount point is done at a channel, and not at a file. That is reasonable if you think that mount is a client operation, and not a file server operation. The client kernel sees the files through channels, therefore the processing to honor a mount point can be done by looking into the channel of interest.

- `chan.c:398,399`
  First the name space (the process group `Pgrp`) for the current process is locked for read. This is important since `domount` is called often for dealing with possible mounts at a channel. Since processes walking through the file hierarchy are not modifying it, they can "read" the information for the channels traversed at the same time; a read lock prevents any change while there are readers, but allows multiple readers at the same time.

- `chan.c:400,403`
  A channel has an `mh` field, that points to a "mount head" or `Mhead`. An `Mhead` (`portdat.h:232,239`) contains information about what is mounted upon the channel: `from` is a reference to the channel used as a mount point, `mount` is a reference to whatever is mounted there. The information about what is mounted in the channel is actually found in the list starting at `mount`; `from` is used just to recover the mount point given the `Mhead`.

  At these lines (`:400,403`), any previous reference to an `Mhead` for the channel is released (`putmhead` only drops the reference to the `Mhead`). More later.

- `chan.c:405`
  `domount` iterates over a set of `MHeads`.

- `portdat.h:394,405`
  Apart from the `noattach` flag, and a read/write lock, you see an array of `MNTHASH` `Mhead` hash entries. This is the real mount table; see figure 5.3. Each "process group" has a namespace with mount entries dictating how is the namespace modified for the `Pgrp`. The `MOUNTH` macro applies a simple hash function to a channel and returns the hash bucket for it on the mount hash table. The hash function uses the `qid.path` to hash the channel.

  The loop iterates on the hash slot for the channel, which has a list of `Mheads` for channels with the same hash value. The list is built using the `hash` field of `Mhead`.

  The mount information is kept at `Pgrp`, in the hash table, and not at the `mh` field of `Chan`; `mh` is just a "cache" for the mount information—so that the table does not need to be scanned all the times.

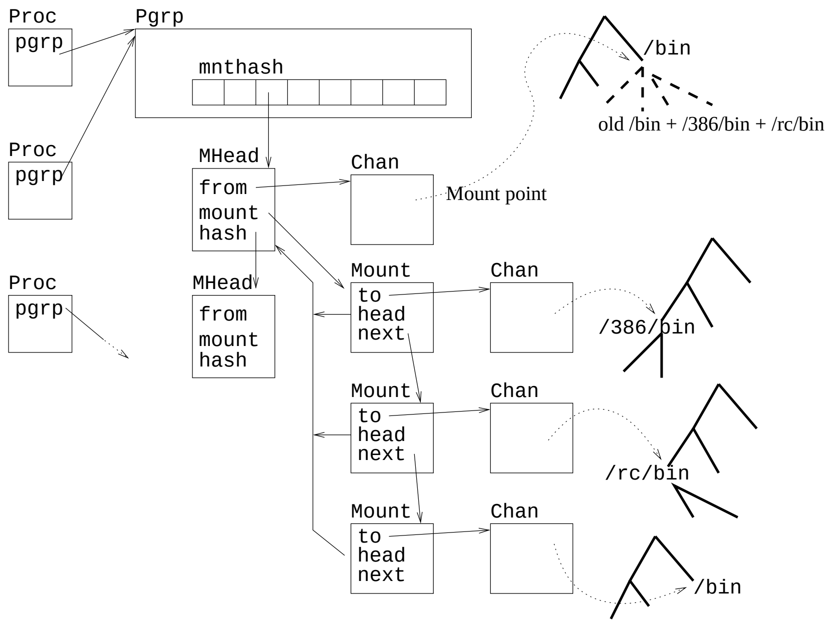


Figure 5.3: Mount structures. Here you can see how two directories have been mounted over `/bin`, with the `MBEFORE` flag.

- `chan.c:406,407`
  Different channels may have `Mheads` (because they were used as mount points), and different channels may have the same mount-hash (`MOUNTH`) value. Therefore, not all channels in the hash list point to `c`'s mount point—if any. After acquiring a read lock on the `Mhead`, `c` is compared with the `from` field of the `Mhead`. Now, the check is done using `eqchan`, and not `==`. Why?

  `eqchan()` *Are both channels referring to the same file?*

- `chan.c:212,223`
  `eqchan` is needed because two different `Chans` can refer to the same file. Regarding the kernel, if two channels point to the same file, they can be considered to be the same—for mount purposes and other things.

  Remember that two channels are a reference to the same file if their Qids, device type, and device number are the same. In our case, `pathonly` is true, which means that `vers` in the Qid is ignored. Remember, the author is locating `Mheads` for the `c` channel, he doesn't care if the mount point is modified or not since the time when `m->from` was set as a mount point (`Mhead`) in the mount table. No matter if `c` is physically the same channel kept in the table, if its file is used as a mount point, it is recognized in the table.

- `chan.c:407`
  If the channel in the `Mhead` does not point to the file `c` points to, it is a different mount point; ignore it. If the channel does not correspond to a mount point, it

will not be in the hash list, and the routine returns the original `c` channel. In this case, `namec` would continue using the initial `c` channel.

- `chan.c:408,413`
  The channel is the same, so `c` is a mount point. `cclone` gets a clone of `m->mount->to`, where `m` was the `Mhead` for `c`. Let's see what this means.

- `portdat.h:220,230`
  `Mheads` have a `mount` field pointing to `Mount` structures. The `Mount` (list) represents file(s) mounted at the channel whose `Mhead` has the `Mount` linked at. Several fields are self-describing: `next` points to following `Mount`'ed trees at the same mount mount; `head` points to the `Mhead`, so that both the mount point (`head->from`) and the list of mounted trees (`head->mount`) are accessible; and `to` points to the channel for the mounted tree. See figure 5.3 if you got confused.

  So, if you mount `/usr/nemo/bin` onto `/bin`, there would be channel for `/bin`, with an `Mhead` entry in your `Pgrp`. That `Mhead` would have `from` pointing to the channel for `/bin`, and `mount` pointing to a list of `Mount` structures. That list would have a `Mount` structure with `to` being a channel to `/usr/nemo/bin`.

- `chan.c:413`
  `nc` is now a clone of the channel for the file mounted at the file pointed by `c`. It points to the same file `to` points to. A clone channel is needed because the returned channel is going to be walked upon return from `domount`; the author does not want `m->mount->to` to walk, it should be kept pointing to the mounted tree. If you read `nc` as "new `c`", it is clear what the code is doing.

- `chan.c:414,415`
  As it was done with `c`, if `nc` has an `Mhead` cached at `mh`, release it.

- `chan.c:416,418`
  So, what was `mh` in `Chan`? It is set to the mount header for a channel that is used as a mounted file tree. The usual picture is to get a channel `c`, then do a `domount` for it, and later use its `mh` field to operate on the mounted tree.

  It is important to see that since `nc->mh` points to the `MHead` in the list, successive mounted files can be quickly found by following `next` pointers in the `Mount` list. You should remember that `nc` represents not just the first mounted file; it also provides access to any other mounted file in the same mount point (by means of `mh`).

  The `xmh` field is set as `mh` when `domount` crosses a mount point, therefore it is the "last mount point crossed"; But it seems to be unused. Either a previous version of the kernel used `xmh`, or it is there for debugging purposes. The last line adds a reference—because of `mh`.

- `chan.c:419,421`
  Channels keep the file name used to create them, as you know. The name for `nc` is not the name of the mounted tree, but the name of the mount-point—which was the name given by the user to `namec`. The `Cname` is now shared.

- `chan.c:422,426`
  `c` is released, its name will stay because of the added reference. From now on, the caller of `domount` should use the channel to the mounted tree instead of the channel to the mount point.

`namec`

- `chan.c:725,732`
  Back to `namec`, `c` is now the channel to the mounted file—or the original channel if nothing was mounted on its name. At lines `:728,732`, `walk` would call `domount` for every path component resolved if `mntok` is set to true. That means that during path traversal, what you read about doing a `domount` for the initial path component, is also done for each remaining path components (because each one could also be a mount point).

  In the first two lines, if the path is "`.`" , `isdot` is set and `domount` does not execute. That means that for the current directory, `c` is the original channel and not the one for any file mounted on it. Besides, if the path is "`/`" (`elem` is empty) and it is being accessed for mounting, `domount` does not execute either and `c` is also the original channel. Why? keep on reading…

- `chan.c:736,739, :756,757`
  The `domount` for "`.`" is done if the access is for `Aopen` or for `Aaccess` (e.g., if you open "`.`", you would be opening the mounted file, in case there is one) However, should the access be for `Atodir` or `Amount`, the original channel is used (i.e. the mount point is not traversed). By default, the author does not execute `domount` at line `:726` and then executes it later for the two cases where it should be done. Why shouldn't it be done for `Atodir` and `Amount`?

  – `chan.c:745,753`
    All `Atodir` (`cd`) accesses got `domount` executed at line `:726`, but for an `Atodir` into "`.`". Thus, for `Atodir` accesses, "`.`" resolves always to the mount point for "`.`", and not to the mounted file—if any.

    Suppose you "`cd`" into a directory. "`.`" is resolved and `up->dot` is set to the mount point for that directory. Suppose you later use (not for "`cd`") a relative path, either "`./something`" or "`.`", `domount` will be called (for a clone of `up->dot`) to resolve "`.`"—it is called either by `walk` or by explicit calls at lines `:736,739` and `:756,757`. This means that if you mount something at "`.`", any future relative path will traverse the mount entry just added for "`.`", even though `up->dot` was set (resolved) before the mount was done. That is the meaning of the comment at lines `:745,748`; if there were a `domount` here for `isdot`, you would resolve the mount when `cd`'ing into "`.`", any ulterior mount would not be processed by your relative path resolution, because `up->dot` already crossed the mount point.

  – `chan.c:780,788`
    Regarding the other case when `domount` is not called for "`.`", it also affects "`/`" paths. When `namec` resolves "`/`" or "`.`" to mount something on it, `domount` is not called. When the path has more elements, `domount` is not called for the final path element (`walk` called `domount` for all but for the

last element). So, in few words, `domount` is never called for the last path element when it is being resolved to mount something on it. Why?

That is because the new `Mount` is going to be added to an `Mhead` for the original channel. When you resolve a path to mount something on it, the channel from `namec` is the original channel for the path, and not the channel for anything previously mounted on it. For example, if you do three `mounts` on `/bin`, the three `Mounts` would be linked at an `Mhead` setup for the first mount. That `Mhead` would be the header of a list of three `Mount` entries. To say it other way, you mount to the mount point, not to something mounted on it.

What remains to learn how are mount points traversed is to see what does "`walk`" when walking through a mount point. Of course you already know how `domount` works, but that is only part of the story (given a file, go to the thing mounted on it); the other part of the story is what to do when you have a directory that is a mount point, and you have to lookup names on it, or to create names on it.

Directory walking

`walk()` *Walks to a name on a channel.*

- `chan.c:493`
  `walk` is called to resolve a file name (no paths!) on a channel. The channel points to a directory (mounted or not). After `walk`, the channel points to the file named `name` within the directory pointed to by the channel given to `walk`. When the directory is mounted, there many be more than one directory mounted. `domnt` tells `walk` whether the given directory is considered to be valid as a mount point; if it is not, no mount related job has to be done.

- `chan.c:499`
  `ac` is the ancestor channel (the parent you are walking through). `cp` is a pointer to a `Chan` pointer; that is to update the passed `Chan`. After a `walk`, the caller might be using a different channel than the one given to `walk`.

- `chan.c:501,502`
  I told you.

- `chan.c:504,508`
  redundant "`..`" names were removed, but what about the "`..`" path? In this case, `undomount` returns the channel for the "`..`" file; more later.

- `chan.c:510`
  `Walk` is going to move the channel to a subdirectory or to a file contained in the directory represented by the channel. If the passed channel had a `CCREATE` flag stating that it was mounted allowing file creation, clean it up so it does not pollute the inner file.

- `chan.c:511,517`
  First, the device specific `walk` routine is called. That is to tell the file server that the file being used by this client (the kernel), and identified by the channel,

should now point to `name` instead. If the device `walk` succeeds, the ancestor file contained a component named `name`;

`Domount` is called on the channel after the walk. The channel points to the file named `name`, but if something was mounted on that file, `domount` will find a mount hash entry for the channel Qid, and traverse it. Next time `walk` be called on the channel it will walk within the mounted tree, not within the mount point.

An interesting implication of calling the device `walk` before looking at mount points, is that to use `bind` on a file, the file must exist, not just its name. However, that is reasonable given that all you have to bind something on a file, is the channel to the file. The ancestor of the file has nothing to do with it. What I mean, is that `bind` is not like `link` on UNIX.

`updatecname()` *Updates the name of a channel after a* `walk`*.*

- `chan.c:477,490`
  `updatecname` adds `name` to the `cname` in the channel, so that `cname` still corresponds to the name of the channel file. Now `ac` is a channel for the file named `name`. For "`..`", `updatecname` uses `undomount` as `walk` did before; more about that later.

- `chan.c:519,520`
  The device `walk` failed; i.e. there was no such file below the file pointed to by `ac`. Now what?

  The device `walk` just tried the name in the served file tree; But it could be that the ancestor channel is a directory with other directories mounted on it (i.e. you are walking through a union). In this case, the channel has a non-nil `mh` pointing to the `Mhead` for the mount point. Besides, it is clear that in this case the `walk` could fail yet the file may be there because of a mount.

  Remember that the channel `mh` was set by `domount`, in case the channel had something mounted on it. Well, not exactly, `domount` received a channel with something mounted on its file name, and `domount` returned a substitute channel for the mounted file, with `mh` set; but you get the picture. What follows is done only for channels that correspond to mount points.

- `chan.c:522,530`
  Going to exercise mounted files in turn. Hold a read lock on the `Mhead`.

- `chan.c:531`
  Remember, `mh->mount` is the first `Mount` entry for the channel, and they are linked through `next`. The author is iterating over the mounted files.

- `chan.c:532,537`
  After getting a clone for the channel pointing to the mounted tree, try the walk on it. Should it work, we are done: got the file!. Otherwise, try with the next mounted thing. Why is a clone needed? Hint: consider what would happen to the mount entry if its `to` channel was used for the `walk`.

- `chan.c:542,543`
  No luck, no such file at any unioned directory.

- `chan.c:545,548`
  Because `c` is already resolved, drop its `mh` reference, if any. `cclone` asked the device for a clone channel, it may come with an `mh` reference, but it should have none; because it is already resolved and it does not represent the mount point—it represents just the file mounted on it. Should `mh` remain set, further calls to `walk` could try to iterate over the mount entries found through it.

- `chan.c:551,556`
  The name for the channel updated. The name coming out of the mounted tree is mostly ignored, and the channel name is updated to contain the name for the channel given to `walk` plus the element name: the channel name is the name used by the user to walk from the root to the channel. Two channels may point to the same file, yet have different names because of `bind`. Start to understand why channels have `cnames`?

- `chan.c:557,561`
  The channel given is updated with the new channel, and it is checked as a possible mount point by `domount`. Although the channel points to a file in a mounted file tree, its name can be a mount point too. `domount` would replace the channel with that of the mounted file, if any.

To completely understand `walk`, you still must read how `undomount` works. The one who understands the meaning of "`..`" when it must cross a mount point (hence its name), is `Undomount`.

Consider the case when `/usr/nemo/bin/rc` is bound to `/bin/rc`. Now, imagine there is a directory `/usr/nemo/bin/386`. If you `walk` to `/bin/rc` you end up in the same directory as if you `walk` to `/usr/nemo/bin/rc`. Consider the case when you do the last `walk`. Your file is `/usr/nemo/bin/rc`. Imagine that you `walk` to `../386`, your file is `/usr/nemo/bin/386`. Now consider that you `walk` to `/bin/rc`, if you later `walk` to `../386`, you should end up in `/bin/386`, not in `/usr/nemo/bin/386`; even tough "`../386`" was applied to a channel really at `/usr/nemo/bin/rc`!

When doing a "`walk("..")`", the new file should be the parent of the current file, but the parent regarding the path used to get to the file. To pick up yet another example, if you use the path `/bin/rc`, you do not care that it was really `/usr/nemo/bin/rc`, for you, "`..`" means `/bin`, not `/usr/nemo/bin/rc`.

By recording in channels the name used to create the channel, "`..`" can be implemented that way. Both `walk` and `updatecname` call `undomount`.

To make it more clear, consider in our example a `walk` to "`..`" on a channel pointing to the file `/usr/nemo/bin/rc` whose `cname` is "`/bin/rc`":

1. `walk` calls `undomount` (`chan.c:506`), which returns a channel pointing to `/bin/rc`, given the channel pointing to `/usr/nemo/bin/rc`. The channel name is still `/bin/rc`, and the channel still points to "`rc`"; but now you are at the `/bin/rc` tree, not any more in the `/usr/nemo/bin/rc` file tree.

2. Later (`chan.c:511`), `walk` calls the device specific walk routine, which would do a `walk("..")` in the channel for `/bin/rc`, making the channel point to the file `/bin`. You are mostly done, but for the channel name, which is still `/bin/rc`.

3. At line `chan.c:513`, `updatecname` would build a name `/bin/rc/..` and simplify it, leading to `/bin`. You are done!

   `updatecname` calls `undomount` too. When `/bin` is also a union, `updatecname` would return the mount point for the union, and later `chan.c:515`, `domount` would return the channel for the first mounted directory at that mount point. That channel has the `mh` field set so that lookups in unions could work for it too.

   If `walk` fails when calling the device specific walk for "`..`", it would continue below line `:517`; if the channel is a union, the same `device_walk` + `updatecname` + `domount` sequence is played at each unioned directory.

...

`undomount()` *Goes from the mounted file to the mount point.*

- `chan.c:436`
  Back to `undomount`, it does part of the "walk" for "`..`" on `c` . In our example, the channel for `/bin/rc` after the bind of `/usr/nemo/bin/rc`, was really a channel to `/usr/nemo/bin/rc`; only that its `Cname` was `/bin/rc`. `Undomount` steps back from the mounted file, to the mount point.

- `chan.c:443,448`
  The meaning of "`..`" depends on the mount table.

- `chan.c:450,453`
  `he` is the end of the mount table. The loops are iterating over the entire mount table: all hash buckets, all mount points, all mounted files. A very expensive operation! (hence the effort to remove unnecessary "`..`" elements in path names). Hopefully, there will not be so many mount entries and this would not have a noticeable impact on system performance.

- `chan.c:454`
  The channel `c` corresponds to a mounted file (it would be an entry for `/usr/nemo /bin/rc` mounted somewhere).

- `chan.c:455,462`
  But, in our example, you don't know whether it is the entry for the mount at `/bin/rc`, or it is an entry for a mount at a different mount point. `t->head` is the `Mhead` for the mount entry, `t->head->from` is the channel for the mount point; its `name->s` is the C string for the mount point channel name. Should the string be the same that the one in `c`'s `Cname`, the mount point is the one we are looking for (e.g. it would be `/bin/rc`). Otherwise the loop continues searching.

- `chan.c:463,467`
  Strings did match, so to step back the union, get a clone of the mount point and stop the search.

  One final note about this. The author breaks just the inner loop and I don't see the point on searching remaining mount points once one has been found—I mean that the two outer loops would continue. I think this is a bug which affects just efficiency, although the author may have a good reason for doing so.

Creating files on mounted places

You may remember that

`namec`

- `chan.c:819,820`
  while resolving a name for creation, `namec` tries a `walk` to the file, and reaches these lines if the `walk` failed. The file named `elem` is going to be created under the file pointed to by `c`. However, if `mh` is not nil in `c`, it is a mounted file. Besides, it is one of (maybe) many files mounted at the directory where the `elem` file has to be created. On which one of these mounted directories must the file be created: on the first in the union that has the `MCREATE` bit set. Channels to mounted files have the `CCREATE` bit set if their mount had the `MCREATE` flag. So, if this channel (the first in the union) has the `CCREATE` flag, it is the channel to the directory where the file should be created.

`namec`
`createdir()` *Chooses where to create a file in a (union) directory.*

- `chan.c:567,593`
  Otherwise, `createdir` locks the `Mhead` and iterates over the set of mount entries. The first one with the `CCREATE` bit set is cloned, and the clone returned. `namec` would then use the channel for the first directory mounted with `MCREATE` to create the file on it.

  The `mh` field of the cloned channel is cleaned up, and reset to be the `mh` for the channel `c` (i.e. for the first mount entry). If the creation fails and another `walk` is tried once more, the `walk` would use the `mh` field to lookup names in the union.

  When no mount entry has the create bit set, an error is raised and file creation is aborted.

Mounting and binding

`sysbind()` *Entry point for the `bind` system call. Binds a name to another name.*

`sysmount()` *Entry point for the `mount` system call. Mounts a file tree.*

- `sysfile.c:687,697`
  Both `bind` and `mount` are implemented by calling `bindmount`. The last parameter is an indication that a mount is being done.

`sysmount`
`bindmount()` *Mounts or binds a file to another.*

- `sysfile.c:624,629`
  The third parameter `flag` controls how the operation behaves. It is a bitmap of bits defined in `lib.h:85,91`. The two low bits are used to specify the order for the new directory. The user cannot request both that the directory be mounted before and after the previous ones. Besides, only bits in `MMASK` are valid flags. `MCACHE` requests that file data should be cached, and is noted in `bogus`. `Bogus` is being initialized to contain the information about the mount.

- `sysfile.c:631,662`
  For `mount`, a file descriptor is supplied as a first argument to `bindmount`. Must convert it into a channel. For `bind`, it is a path, which must be converted to a channel too.

- `sysfile.c:632,633`
  `mounts` are forbidden if the `noattach` bit is set. It is okay to `bind` because no new files are brought into the name space, but no new file trees can be attached.

- `sysfile.c:635,640`
  `fdtochan` takes a file descriptor and returns a channel for it. The last parameter specifies to add a new reference to the channel. Should an error occur, `cclose` would drop the reference. The channel corresponds to the file descriptor being mounted, and is noted in `bogus`. On the other end of the channel, there should be someone speaking 9P, to service file requests.

- `sysfile.c:642,650`
  For `mount`, a request is going to be issued to the file server to attach its file tree to our name space. As a server can service several trees, the fourth argument of `mount` specifies (as a string) which file tree to mount. The author is ensuring that the string is in valid virtual memory, and noting the string into `bogus.spec`. `nameok` is used to check that the `spec` has a valid path (i.e. valid characters); it is okay if `spec` contains slashes, hence the `1`.

- `sysfile.c:652,656`
  The file tree being mounted is serviced by a remote process which speaks 9P. The client is going to be the kernel mount driver, which translates procedure calls into 9P RPCs as the mounted tree is used. The `attach` procedure of the `mnt` driver is used to get a channel to the remote process: First, knowing that the name for `mnt` is `#M`, `devtab` is searched by `devno`. `devno` returns a valid channel type (an index into `devtab`) for the mount driver; Second, the `attach` procedure for the driver is called supplying the `bogus` structure just filled up. If you look at it, `bogus` contains a channel to the server, the `spec` for the file tree requested, and an indication of whether files are being cached or not; everything `attach` needs to get in touch with the file server and attach to it. This is discussed later.

  If `attach` completes without error, `c0` is a channel to the (root of the) file tree in the server, which (after completion of `attach`) recognizes us as a valid client and is willing to talk 9P with us.

- `sysfile.c:658,662`
  The first argument for `bindmount` was the path given to `bind`; therefore, to obtain a channel it suffices to use `namec` to resolve the path into a channel. `c0` is now a channel to the file tree being mounted.

- `sysfile.c:669,674`
  The second argument was the path for the directory where the file tree is being mounted at—or for the file where the new file is being bound at. `namec` is used to get a channel `c1` for the mount (or bind) point. Note how `Amount` access mode is used in `namec`; `c1` would be the very first mount point for `arg[1]`.

- `sysfile.c:676,685`
  The first line is where the mount is being done; what remains is to close both channels (mount point/mounted file) because the mount table already holds what it needs, and to close the descriptor supplied to `mount` in case it was a `sysmount`. The descriptor is closed because the kernel is going to exchange 9P messages on it with the file server; the user would only interfere and cause problems. But how does `cmount` work?

```
sysmount
   bindmount
      cmount()
```
Adds a mount entry.

- `chan.c:226`
  `cmount` is called with channels for the mount point (`old`) and the mounted file (`new`), the flags and any spec are passed too.

- `chan.c:233,234`
  It is okay to bind a file to a file, and a directory to a directory; but not for any other case. Don't trust users!

- `chan.c:236,239`
  If mounting with `MBEFORE` or `MAFTER`, ensure channels are for directories; otherwise, it has no sense—for files, you only can replace entire file contents.

- `chan.c:241,242`
  Going to write the `Pgrp`, stop any further lookup/change to the name space.

- `chan.c:244,249`
  Search the name space for any `Mhead` for this mount point (note `eqchan` again!). If an `Mhead` is found, `m` points to it; otherwise, `m` would be nil.

- `chan.c:251,269`
  The comment says it all. The `from` field of the `Mhead` is set to the mount point—and the reference noted by `incref`. As you now know, the point is not that this particular channel is set in the `from`; the point is that a channel with its `cname`, its `type`, its `dev`, and its `qid` is sitting at the `from`.

  Lines `:267,268` may add a mount entry for the mount point itself to the `Mhead`. Guess why?

  Exactly, if the mount is not an `MREPL`, previous contents at the mount point are still visible; therefore, the channel to the mount point is added as if it was one of the directories mounted on it. When later the mount entry list be searched, names originally at the mount point will be searched too. So, the `Mhead` has now either the original mount point channel, or it is empty.

`newmount()` *Adds a new mount entry.*

- `pgrp.c:231,245`
  `newmount` simply allocates a `Mount` entry, and initializes it. The author sets `to` i to the channel for the mounted file; and sets the pointer to its `Mhead`.

`mountfree()` *Releases mount entries.*

- `pgrp.c:247,259`
  The counterpart of `newmount` is `mountfree`, which releases references to the channels for the mounted files as well as the mount entries.

- `chan.c:270,275`
  After the mount entry is locked, there is no need to keep locked the whole name space.

- `chan.c:277`
  An entry for the new mounted file is created.

- `chan.c:278,292`
  As the mounted file could itself be a union, any mount entry for the mounted file must be copied to the list of entries for the mount point. The author links a copy of each such entry starting at the `next` pointer for the new mount point being set (`nm`).

  If the mounted file is to replace the mount point, its mount entries are set with the `MAFTER` flag. I don't think this flag is used for anything. The point is that the `Mhead` for the mount point has either its previous mount entries, or an entry for the old contents if appropriate; besides, the new `Mount` has all entries for the mounted file linked on it through the `next` pointers.

- `chan.c:294,297`
  If this mount was an `MREPL`, cleanup all previous mount entries for the mount point. `mountfree` releases *all* the mount entries.

- `chan.c:299,300`
  Cache the `MCREATE` flag on the channel, so `createdir` only needs to look at the channel.

- `chan.c:302,306`
  If mounting after, skip any previous mount entry at the `Mhead`, and link the new mount point (and all trailing mount entries for directories mounted at the mounted file!) at the end.

- `chan.c:307,312`
  Link the list of new entries before the previous ones. If mount was `MREPL` and not `MBEFORE`, the list was already cleaned up, so a "link before" works too. In the case of `MREPL` it could be more efficient not to iterate the list being added to the `Mhead` (because that is just to set the last `next` pointer to nil), but nobody cared to optimize that. Would the user notice the optimization?

Mounted files can be unmounted by calling `unmount`.

`sysunmount()` *Removes a mount entry.*

- `sysfile.c:700`
  This is the entry point for `unmount`

- `sysfile.c:706,707`
  The name for the mount point is resolved (note the `Amount` access) and `cmount` is now a channel to it.

- `sysfile.c:709,717`
  If the first argument is not nil, it specifies the mounted file—in this case the user wants to unmount that specific mounted file, and not any other one. The address is checked and `namec` used to get a channel for the mounted file (note the `Aopen` access mode). Should the first argument be nil, `cmounted` remains set to nil.

- `sysfile.c:726`
  `cunmount` undoes the mount. Note, not `undomount`!

`sysunmount`
  `cunmount()` *Undoes a mount.*

- `chan.c:329,334`
  The `Mhead` is located for the mount point—yes, perhaps a `getmount` routine could be created to do the lookup and share these lines among routines looking up mount entries.

- `chan.c:336,339`
  The user could call `unmount` on a file with nothing mounted on it.

- `chan.c:341,352`
  If shouldn't care about unmounting a particular mounted file, `mountfree` is called on the whole list of `Mount` entries. All entries are released and the head released too. All mounts are undone. The `unlock` is done in any case after locking the particular `MHead` of interest.

- `chan.c:354,376`
  The user cares of unmounting a particular mounted file. So, iterate the set of mount entries for the mount point. Iteration stops when an entry is found such that its `to` channel is `eqchan` to `mounted`. (Line `:358` checks if the channel used to talk 9P to the server of the mounted directory is `eqchan` with the mounted directory; this can happen with `exportfs`, as you will see later when you read about the mount driver). If the entry is found, it is removed from the list and released. If the list gets empty, the `Mhead` is released too. If no entry is found, an error is raised—that "mounted file" was not mounted at the union.

Creating and destroying name spaces

Other routines that operate on name spaces are the ones creating an empty name space, copying an existing name space and deleting a namespace. That happens as a consequence of `rfork` and process death.

`sysrfork`
  `newpgrp()` *Creates a new name space.*

- `pgrp.c:42,51`
  An empty `Pgrp` is created with everything set to nil. As a result, the mount table hash is empty and `noattach` cleared. You saw in a previous chapter how `pgrpid` was assigned later.

sysrfork
  `pgrpcpy()` *Copies a name space.*

- `pgrp.c:124,130`
  `pgrpcpy` is used to copy a name space into another. Only the source is locked because the target is still being built.

- `pgrp.c:132,159`
  Hash entries are replicated so that `to` holds a copy of the mount entries in `from`. Note the `increfs` when structures are shared (Channels are!, because the namespace is copied, but channels to mounted file systems and mount points are not!). The loop is simple to understand if you notice that for each pass (`:135`) an `Mhead f` is being copied into a new one, `mh`, (`l` is used to build the list); and at each inner pass (`:144`), `Mount m` is being copied into a new one, `n`.

  The field `copy` of `m` is set to `n` (the place it is being copied to). `pgrpcpy` relies on `pgrpinsert` to link mount entries being added through the `Mount.order` field. Both `copy` and `order` seem to be in `Mount` just for this occasion—to save the author the burden of allocating a whole bunch of data structures during a brief amount of time just to copy the set of mount entries.

  `pgrpinsert()` *Inserts a mount into a list ordered by `mountids`.*

- `pgrp.c:100,118`
  What does `pgrpinsert` do? It receives the pointer to the `Mount` being copied and a pointer for the start of the "order" list. If the list is empty, it is set to the node just copied. If the list is not empty, `f` advances until its `mountid` is bigger than the one for the node copied. So, `pgrpinsert` is building a sorted list of `Mount` entries. The list is sorted in ascending order of `mountids`.

- `pgrp.c:163,167`
  That was the list for, `mountids` for the copied entries are assigned in the same order they were assigned for the original `Pgrp`. Why?

  The `mountids` can tell the order in which the user issued `mount` requests that resulted in the set of `Mount` entries. That order does not correspond to the order of entries in the `Mhead`, because of the `MBEFORE`/`MAFTER` flags. The `proc` driver services an `ns` file that returns (when read) commands to replicate the namespace. It relies on the order of mountids to reproduce the commands in the appropriate order.

  Execute in your Plan 9 box a couple of `mounts`. Then, using the same window, go and then execute `cat /proc/$pid/ns`. More clear now?

`closepgrp()` *Releases a name space.*

- `pgrp.c:71,97`
  `closepgrp` is called to release a reference to the `pgrp`. When it comes down to zero, all entries are released. Two locks are needed: `devproc` uses the `debug` lock, although routines using the namespace use the `ns` lock.

5.3 File I/O

Once a file descriptor is open, the user can use `read(2)` and `write(2)` on it. Although the actual implementation is done by the server servicing the file, functionality such like the file offset is kept at the client side and is provided by channels (You already know what a file offset is, if you don't, read the "File I/O" section of `intro(2)`).

That is a fine way the author has to side-step the problem of sharing file offsets on distributed file systems: by not doing it! Take this as an example of the principle that, before adding a new feature to your system (e.g. sharing file offsets among several nodes) you should ask yourself what is the benefit, and what is the cost; then decide.

The most useful feature of shared file offsets, i.e. allowing several related processes to consume/produce the same file, is still here:

- Several processes (within the same node) can share a file descriptor and that means they would share the file offset too.

- The "append only" bit of file permissions can be used to allow several processes to produce contents for a file no matter the node they are running at. This is in effect "sharing" the file offset, which is always kept at the end of file for write purposes.

5.3.1 Read

`sysread()` *Entry point for the `read` system call. Reads from a file.*

- `sysfile.c:375`
  `sysread` receives a descriptor and a buffer together with its length. It is expected to fill up the buffer with bytes coming from that descriptor. The bytes can come from an actual file, or from a file synthesized by a file server, nobody knows.

- `sysfile.c:381,383`
  Buffer addresses are verified and `c` is set to the channel for the descriptor. The channel should have the `OREAD` bit set. Remember that when you open a file, its mode is cached in the channel. Although permissions are checked by the file server, when the device specific `open` routine is called, once the file server granted access, the mode is kept in the channel. Future access checks can be done by the client without disturbing the server (although the server is likely to check that the file id has the `OREAD` bit set too).

  The last `1` to `fdtochan` requests an `incref` for the channel. Should the descriptor be closed by a different process, the channel would stay alive because of the added reference. Forget by now the `1` asking `checkmnt` to `fdtochan`.

- `sysfile.c:390,396`
  If the channel Qid has the `CHDIR` bit (it is a directory), a read should return a integral number of directory entries (See the `read(2)` manual page). The buffer length (`n`) is adjusted to be a multiple of the directory entry size `DIRLEN`. Should the channel offset be not a multiple of `DIRLEN` (shouldn't happen) or the buffer too small to keep even a single entry, `Etoosmall` is raised.

- `sysfile.c:398,405`
  If `c` is a union, `read` should get entries from all the mounted directories.

  It is easy to know if this is a union; remember that the channel installed for the file descriptor was obtained with `namec` and `Aopen` access mode. That means that any mount point was traversed and `mh` initialized in the channel to point to the `Mhead`.

  Thus, if there is an `mh` in the channel, `unionread` reads entries from each directory mounted. Otherwise, the device specific `read` is called. The device should honor the convention that a read from a directory should return an integral number of directory entries. Should the device fail to honor that convention, the channel offset would be set to a non-multiple of `DIRLEN` at line `:404`, and the next `read` will fail. The offset for the file is kept by the channel; you already knew this. For directories, the file offset counts bytes, and not directory entries! as it should be.

  The channel is locked just while changing its offset. Apart from that, it can be used safely without locking because it is not going to be deleted, nor its device type is going to change. The real `read` is done by the device, which could stand miles away, and there is no guarantee that while this read is in progress no other read could be made. Nevertheless, the server is likely to serialize read/write requests for the file.

`sysread`
  `unionread()` *Reads from a union.*

- `sysfile.c:280`
  `unionread` does the work for reading the union while holding a lock on it.

- `sysfile.c:291,292`
  Where to start reading? If you read a union from the beginning to the end, you should get all directory entries in all directories mounted, as you find them in the list of `Mounts`. The problem is that users tend to declare a buffer and read repeatedly from a file; each read should start past the previous read (remember offset, right?).

  The author could use the channel offset and skip as many `Mount` entries and directory entries as needed to fill up `offset` bytes. However, that would be a waste because reading a union would be $O(n^2)$ regarding the number of entries. You don't know how many entries there are at each mount entry. Therefore, you cannot compute how many entries to skip unless you read them. Can you think of more alternatives besides re-reading them and doing what is said in the next paragraph? What are the benefits? What are the penalties?

  Instead, an `uri` (union read index) field in the channel is used to record how many mount entries were exhausted by previous reads. That saves lots of reads that are likely to be serviced from the network and not from a local cache. The loop at lines `:291,292` is skipping over already exhausted mount entries.

- `sysfile.c:294`
  Keep on reading while there are mount entries to read from.

- `sysfile.c:299,300`
  As far as I know, `to` is only released by `mountfree`, and that is done with the lock on the `Mhead` held. Therefore I would say that in no case `to` should be nil, but the check does not hurt anyway.

- `sysfile.c:301`
  Going to read from `to`, so clone it. Otherwise, the channel would be modified (e.g. its offset would change). Perhaps in this case it would be more simple to save the previous offset, explicitly set it up, use the channel and restore the offset. But that would require that all channels for mounted directories be kept open all the time; that would mean "use more resources" for the file servers. Besides, it is a good thing to keep channels for `Mount` entries untouched; the author can rely on that to make the code more simple.

  Don't you see any other problem here? You cannot walk on an open file, so the clone is necessary anyway.

- `sysfile.c:305,308`
  This is not the usual error handling block. Should an error occur, the channel to the mounted directory is closed and the routine continues at the next entry. That means that if a file server goes down, only people really using it would be affected. If a directory serviced by the crashed file server is mounted, its entries will be ignored due to errors, but remaining entries in the union would still work. This is the kind of thing that makes a distributed system more reliable: be prepared to tolerate remote failures bothering the user as few as possible.

- `sysfile.c:310`
  The device specific open for the mounted directory is called. The clone channel is used.

- `sysfile.c:311,314`
  The author is indeed adjusting `offset` by hand between successive `unioreads`. The offset is saved in the channel for the mounted file; But note how the `offset` is used only within the same mounted directory.

- `sysfile.c:318,321`
  If could read something, return that. If more entries remain to be read, the user would call `read(2)` again. `offset` is adjusted before returning so that next time the read would start where it was left at.

- `sysfile.c:323,329`
  If there are no more in the current mounted directory, `nr` would be zero and this code execute. When an error happens, line `:307` would jump here too. Just increment `uri` to remember that the current mount entry should be skipped next time. If there are no more entries, break the loop and return 0 (eof). Otherwise, reset the `offset` and iterate again, so that the next mounted directory starts to be read at offset zero.

5.3.2 Write

`syswrite()` *Entry point for the `write` system call. Writes in a file.*

- `sysfile.c:444`
  `syswrite` has the same interface of `sysread`, but it writes rather than reads.

- `sysfile.c:453,459`
  Should an error happen, restore the offset in the channel. The code following advances offset, yet it could fail. If an error occurs, the write failed but the offset should be kept untouched. Perhaps this would be more clear if the offset were simply saved to a variable and restored from there, but the author does not do so. Guess why?

- `sysfile.c:461,462`
  The only way to write to a directory is by creating or deleting files (also by changing file attributes).

- `sysfile.c:464,467`
  Advance the offset by the number of bytes to be written. `oo` keeps the old offset, and could perhaps be used to restore the initial channel offset on errors, perhaps not.

- `sysfile.c:469`
  Here is the actual write. The old offset is supplied and the device specific write would write at that position.

- `sysfile.c:471,475`
  The device could write less than `n` bytes. This is usually due to an error (e.g. "disk full"). If the device wrote `m` bytes, the offset has to account for those `m` bytes. However, it was `n` that was added to it, by subtracting `n-m`, it gets `m` units more than it had before `syswrite`. Should the device write all `n` bytes, offset is kept `n` bytes beyond its value before `syswrite`.

Could you guess the reason for the dance around `offset`? `sysread` did it in a more straightforward way, by simply adding `n` bytes to the offset after the read was done.

I think that the reason is that the author does not want the channel to be locked during the whole `syswrite`. Suppose the channel offset is `o`. For `sysread`, the worst thing that can happen if two different processes are reading the same file (through the same channel) is as follows:

- The first `sysread` calls `read` in the device and reads at offset `o`.

- The second `sysread` calls `read`, in the device, which also reads at offset `o`!

- Both `sysread` increment the channel offset.

Although this could be considered to be a bug (If I am not missing anything here), the net effect is not harmful, as the file is kept in a consistent state. Actually, I would say that the bug is at the application, which didn't synchronize on its access to the file.

Now consider `syswrite`. `writes` are different in that if they are mixed the file could be left in an inconsistent state. Suppose again that offset is `o`, and two `syswrites` are being done through the channel. If `syswrite` worked like `sysread` (by adding `n` to offset after calling the device `write`), the two writes could overwrite the same portion of the file; e.g. the two calls to the device `write` are performed, then the offset incremented twice (holding the lock for the offset).

By incrementing the offset in the channel before calling `write`, any following write through the same channel would ask the device to write past the `n` bytes theoretically being written. Of course that means that the offset must be restored (in case of errors) by doing arithmetic and not by restoring the initial value. The reason is that if the first `syswrite` fails, but in the mean time the second `syswrite` could really write, offset should account for the written bytes by both the first and the second `syswrite`. If you didn't understand, try to exercise concurrent `syswrites` and see how are the file and the channel left. I think that the approach used in `syswrite` should be used in `sysread` too, but the author may disagree.

Finally, note that when two different channels are used for the same file (applications could run at different nodes), it is the application responsibility to synchronize. The `CHEXCL` bit could help here. As it is said in `create(2)`, if a file is created with that permission bit set, only one client may have the file open at a time. That functionality is implemented by the file server, which serializes file usage when notices the `CHEXCL` mode bit.

5.3.3 Seeking

The `seek(2)` system call can be used to move the offset in an open file to a desired position.

`sysseek()` *Entry point for the `seek` system call. Moves the file offset.*

- `sysfile.c:535,541`
  After checking the argument, `sseek` does the work.

`sysseek`
    `sseek()` *Seeks on a channel.*

- `sysfile.c:495`
  The first argument is the file descriptor to seek on. Get the channel. But note the `arg[1]` (not `arg[0]`!) for the first argument.

- `sysfile.c:500,501`
  Should `seek` be allowed on directories, `sysread` could found offsets not aligned to `DIRLEN`. So the author forbids that.

- `sysfile.c:503,504`
  No seeks on pipes. Other file servers could perfectly ignore seeks as well as they could ignore file offsets (i.e. they could read always from the very first byte). But that's the file server choice.

- `sysfile.c:506,509`
  An `vlong` (second argument to `seek`) uses two longs. Now, the arguments for system calls were assumed to be machine words (longs). These lines use the `u` member of the `o` (offset) union to fill up the two words of the `vlong`. Arguments 2 and 3 are actually the second argument (two words) of `seek`. The first argument was kept at `arg[1]` and not at `arg[0]`. That is because `arg[0]`

is kept unused to make the vlong stand aligned to an vlong size boundary regarding the argument array. That is a good thing if the code is pretended to be portable, as some machines (not the PC) are very picky regarding alignment issues. The user-level library stub to issue the system call for seek must honor the same convention of wasting arg[0] and pushing the vlong in the order in which it is being extracted here. If the stub (machine dependent) honors this convention, this code can be kept portable. Now you know also that arg[4] is actually the third argument for seek, right?

By the way, look /sys/src/libc/9syscall/mkfile:50,61. Understand the if now?

- sysfile.c:509,528
  Depending on type, the n kept at o should be either used as the new offset, added to the current offset, or used as the new offset but counting backward from the end of the file. When arithmetic is being done with the channel offset, a lock is held. Does it makes sense to hold a lock just to assign the offset?

  If the offset can be written atomically, it doesn't matter; but on Intels, in this case, two words are written as the new channel offset. There is a very thin critical region here, and the offset could be mangled in the very improbable case that a sseek writes the offset, while another sseek does too. Just too improbable, but perhaps that should be fixed.

  Stat is used to get DIRLEN bytes with status information from the file, and convM2D is used to convert that into a Dir structure, from where recover the length of the file. stat is discussed later, but convM2D is not. The DIRLEN bytes are in a "standard format" and have to be converted to a native format before used (byte ordering et al.). The reason for this all is that Plan 9 is a distributed system.

- sysfile.c:529,531
  The final offset value given to the user (line :506 is not needed but it is good to keep it there for safety). uri is reset, because unionread could use that to read entries? That couldn't happen. If this is a directory, seek is forbidden. However, it is a good practice to set uri to a reasonable value, in case a bug (no check for directory?) is introduced in the code by future changes.

- sysfile.c:543,560
  You may have noticed that there is a sysoseek routine below sysseek. If you look at system call numbers for seek and oseek (libc/9syscall/sys.h), you would notice that seek is placed the last one, and oseek is placed near read. That means that the author had once just the seek system call, and it was number 16 (OSEEK now). But seek was changed in a so incompatible way, that the author preferred to keep both the previous and the new version for seek in place. Old Plan 9 binaries would have the library stub named seek that does a system call with number 16, and that would call sysoseek. New Plan 9 binaries would be built with the stub which calls seek instead. This is a common technique to reduce the impact of changes. What sysoseek does is to rearrange the argument array so that a[0] contains the address of the vlong, and the

`vlong` is filled up with `arg[1]`, which means that the change was probably that seek received a `long`, and now it receives an `vlong`. In other words, `sysoseek` is adapting the old interface to the new interface, but the implementation of `seek` stands the same. Try to learn from this all how to minimize the impact of changes.

5.3.4 Metadata I/O

Files have attributes, and you already know some. They have permissions, a name, owner, etc. The system calls reading/updating attributes are `stat(2)`, `fstat(2)`, `wstat(2)`, and `fwstat(2)`. The former two ones read attributes, and the later two ones update attribute values. Services like `chmod(2)` are implemented by doing a `wstat(2)` on the affected file. File attributes are read and written in a machine independent format with `DIRLEN` bytes per file attribute set. Read `stat(5)` if you are curious about file attributes.

Let's see the system calls that read attributes before, and then the ones that write them.

`sysfstat()` *Entry point for the* `fstat` *system call. Reads file attributes (given a file descriptor).*

`sysstat()` *Entry point for the* `stat` *system call. Reads file attributes (given a file name).*

- `sysfile.c:563,578`
  `sysfstat` takes an open file descriptor, and tries to read new values for attributes. All the routine does is to get a channel and call device specific `stat` to read the attributes. It is the file server the one filling up the buffer with information (probably) coming for `Dir` structures.

- `sysfile.c:581,597`
  `sysstat` is the same, but takes a file name instead of a file descriptor. `namec` does the job of getting a channel, and then the device specific `stat` routine reads any attribute. Perhaps both system calls could be folded into one, but it's not a big deal.

`syswstat()` *Entry point for the* `wstat` *system call. Writes file attributes*

- `sysfile.c:778,813`
  The "write attributes" version of the routines simply call the device `wstat` instead of the device `stat`. The only thing the author verifies regarding the new attributes, is that the new name for the file (first bytes in the buffer with attributes supplied) looks fine and has valid characters. That is the only attribute that would hurt 9P. Should the name be ok, the call to `wstat` can be done, and remaining attributes should be checked by the file server.

This is another place where you can see how the design of Plan 9 works well for a distributed system. File attributes are kept (together with the file) within the file server providing the files. The system does not impose any particular way of implementing file attributes. All Plan 9 cares about is that the device should either be in-kernel, or service 9P requests. Besides, by forwarding all calls related to file metadata to the file server process, Plan 9 does not introduce new problems related to metadata sharing over a distributed system. Yet another point is that the file server is free to trust you and accept `wstat` requests; it would do so if you authenticated the connection to the file server. Saw how all pieces fit together?

5.4 Other system calls

5.4.1 Current directory

The `getwd(2)` function is not a system call. It is a library function that opens "`.`" and calls `fd2path(2)` on it.

`sysfd2path()` *Entry point for the* `fd2path` *system call. Returns the name for a file descriptor.*

- `sysfile.c:120,135`
  `sysfd2path` receives a file descriptor and a buffer together with the buffer length. It verifies that the buffer has valid addresses (`arg[1]` is the pointer to the buffer and `arg[2]` is its length). Then it uses `fdtochan` to get the channel for the open descriptor. The work was really done when the channel (`Cname`) was built. The only thing `fd2path` has to do is to extract the name (if there is any) and print it in the user buffer. The channel name is the name used by the user to get to the file. If the user used a relative path to open the descriptor given to `fd2path`, the name of `up->dot` was used to build `c`'s name, in `namec`.

Another useful system call is `chdir(2)`

`syschdir()` *Entry point for the* `chdir` *system call. Changes the current directory.*

- `sysfile.c:599,610`
  It simply verifies the name given, and resolves it to a channel using `namec` (Note the `Atodir` access, that was explained before). Then `dot` for the current process is set to that channel, after releasing the reference to the previous `dot`.

5.4.2 Pipes

There is a system call to build a pipe. A pipe is a buffered channel used to let two processes communicate. Surprisingly, the pipe system call is not a real system call; I mean that although it is a system call, it uses files serviced by a `pipe` device to provide its service. The system call is provided as a convenience, although the user is perfectly capable of using the pipe device himself without relying on the system call. Using the device has the good thing that the user is conscious that pipes are provided using files serviced by `pipe`, and they could be even mounted through the network.

`syspipe()` *Entry point for the* `pipe` *system call. Creates a full-duplex pipe.*

- `sysfile.c:138,146`
  `syspipe` is called to create a pipe (see figure 5.4). The pipe interconnects two file descriptors so that at one you read what was written at the other.

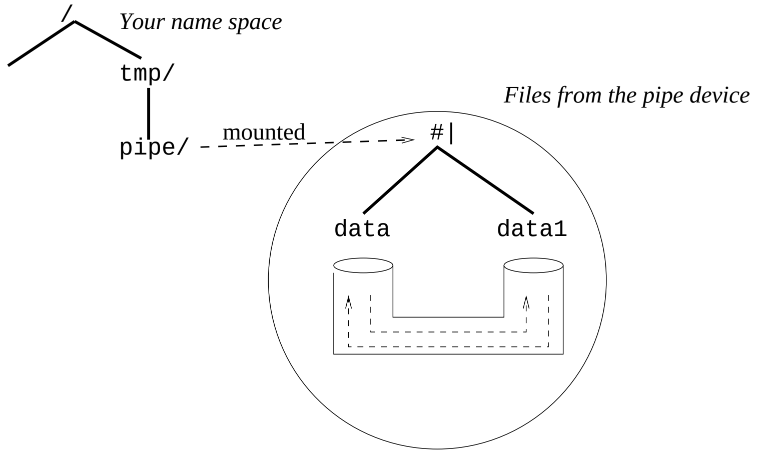


Figure 5.4: Pipes are bidirectional channels accessed through files provided by the `pipe` device.

`arg[0]` is an array with space to put the descriptors numbers in. `evenaddr` is used because integers are going to be written into `arg`; some machines issue alignment exceptions when you write an integer into a location not aligned to an even address.

- `sysfile.c:147`
  `d` holds the `Dev` entry for the pipe device, which is named `#|`.

- `sysfile.c:148`
  `c[0]` is a channel to the root of the pipe device file tree. The name works despite what the user did with his mount table. Here is where the pipe was actually setup. If you read the `pipe(3)` manual page, it says that an attach of the pipe device causes a new pipe to be created. The pipe endpoints are files named `data` and `data1` below the root supplied by the pipe device. Different attachments to the pipe device cause different pipes to be created; files `data` and `data1` would be different for each attach.

  As a side note, Plan 9 does not have the UNIX `mkfifo` system call, which creates a pipe with a name in the file system. In Plan 9, all pipes have names, even those created with `pipe(2)`. Execute "`bind '#|' /tmp/dir`", and then try to read/write files in `/tmp/dir`.

- `sysfile.c:150,151`
  By default, descriptors given to the user are invalid ones. On error, the user will know.

- `sysfile.c:152,161`
  Preparing to cleanup on errors. Descriptors are closed only if they were open. `fd` holds the the descriptor numbers, which are indexes for the `Fgrp` descriptor array `f->fd`.

- `sysfile.c:162,166`
  Both `c[0]` and `c[1]` are channels to the root of the pipe device. So, walk them to `data` and `data1` to get channels to both ends of the pipe. (why did not the author choose names "data0" and "data1" for both endpoints?)

- `sysfile.c:167,168`
  Important. In Plan 9, the walk merely changed the channel to point to a different file, but you have to open the file before doing I/O. The mode is `ORDWR` because Plan 9 pipes are bidirectional.

- `sysfile.c:169,170`
  The caller of `pipe(2)` is unaware of pipe files, the author plumbs the channels to a couple of file descriptors and `pipe` is done.

5.5 Device operations

In the code you read before, you noticed that the actual file system work was done by device specific operations. In fact, you already knew this since chapter 3, "Starting Up". Let's read now the code still unread regarding devices and device operations.

- `portdat.h:175,195`
  The `Dev` structure (which you already saw), contains the information for a known device type. Instances of `Dev` are configured into `devtab` when compiling the kernel. The first two fields contain the "device character" and the device name. You know that kernel devices have names like "`#C`", the "`C`" is at `dc`; it identifies the device type. To locate the entry in `devtab` for a particular device, you only have to iterate through it searching for an entry with the wanted `dc`. The name is mostly for debugging.

  Regarding device operations, they correspond to 9P messages (read `intro(5)`). When a user process (or the kernel) performs a file operation, that operation translates into 9P requests (transactions). For example, you know that to open a file you need to `clone` a channel, `walk` on it, and `open` it. These procedures (`clone`,`walk`,`open`,etc.) correspond actually to `Tclone`, `Twalk`, and `Topen` 9P requests. When the file server is within the kernel, the kernel looks up the device in `devtab` and calls its implementation for the request (the `Dev.open`, `Dev.walk`, etc.); when the file server is remote, the kernel driver for the file is the mount driver, which issues 9P messages (`Topen`, `Twalk`, etc.). Do you get the picture?

  9P uses the term "transaction" for requests. So, what is each 9P transaction for? One fine way of learning it is to look at the implementation of each transaction for a particular device. I'm going to use the `pipe` device. While you read the `devpipe.c` source, you will learn what is each transaction for; and you will see how the file `dev.c` provides default implementations for most 9P operations. Such default routines are handy when a device has nothing to do (but for replying to the request) to service a particular 9P transaction. All devices are file servers, and all file servers are likely to share much code; that shared code is located into generic utility routines in `dev.c`.

5.5.1 The pipe device

Initialization

- `devpipe.c:370,389`
  This is the "entry point" for the pipe device. The `Dev` structure is linked into `devtab` so that the kernel can locate it. The kernel device name is "`#|`", among routines linked here, there are routines corresponding to 9P transactions.

  `devreset()` *Generic procedure to reset a device.*

- `dev.c:62,65`
  The "reset" procedure is not a 9P operation, but a routine provided to "reset" the device to an initial state so that its `init` procedure could be called. You saw how it worked for ethernet devices while learning how the system boots. For pipes, nothing has to be done to reset the device; and that is very common. The `dev.c` file contains a `devreset` routine to use as a generic `reset` procedure. It does nothing. Instead of declaring an empty routine for each device without resetting needs, this one is used.

  Remember that `chandevreset` calls all reset procedures for configured devices at boot time.

`pipeinit()` *Initializes the pipe device.*

- `devpipe.c:41,50`
  The "init" procedure is not a 9P operation. It is used to initialize the device and prepare it for operation. After `init` is called, other procedures can be called. For pipes, the configured `pipeqsize` parameter determines the size of the queue used by each pipe. Should it be unspecified in the configuration file, 256K are set for multiprocessor machines, and 32K for monoprocessors. For multiprocessors, processes at both ends of the pipe can execute at different processors. It is not a problem if a process is allowed to run until it puts 256K in the pipe, even if the reader has not read a single byte. By giving more room to the pipe, the writer can write more without blocking, and the reader can read more without blocking—even if the other process is not attending the pipe. On monoprocessors, the author thinks that it is better not to let the process run for so long before being blocked; after all, the process at the other end of the pipe has to use the only processor in the system.

  `devinit()` *Generic procedure to initialize a device.*

- `dev.c:62,70`
  The default implementation (used if the device does not care about init) is one that does nothing.

  You know that `chandevinit` calls all the init procedures for configured devices (after `chandevreset`) at boot time.

Attaching to the server

`pipeattach()` *Attaches to the pipe device.*

- `devpipe.c:55,56`
  The first 9P request issued to a file server is an `attach`. It attaches a client to the file server. Any authentication is done here. (see `attach(5)`). Usually, `attach` would just attach to the server's file tree. However, for `devpipe`, attach also creates a new pipe and attaches the client to its corresponding file tree. To create multiple pipes, you attach multiple times to `#|`. An string `spec` is supplied to `attach`. That is useful in case a server services multiple file trees, to select one of them.

- `devpipe.c:61`
  `devattach` is a generic attach procedure that contains common stuff for attach procedures. Both the name of the device and `spec` are given to it.

`pipeattach`
  `devattach()` *Generic code to attach to a device.*

- `dev.c:72,78`
  `devattach` creates a new channel, `c`, which would point to the root of the device file tree.

`pipeattach`
  `devattach`
  `newchan()` *Setup a new channel.*

- `chan.c:67,103`
  `newchan` tries to get a channel structure from a free list found in `chanalloc` (a channel allocator). Should the free list be empty, `smalloc` is used to allocate a `Chan`, and it is linked into the `chanalloc` list. Channels in use are linked into `chanalloc.list` through the `link` field of `Chan`; channels not in use (deallocated) are linked into `chanalloc.free` through the `next` field of `Chan`. The `fid` field is given an unique value, different from other channels in the system. `chanalloc.fid` contains a number which is incremented every time a channel is created. So, every `Chan` in the kernel has its own `fid`. The FID, represents a file for the client in 9P. When requests are issued from a client to a 9P file server, every file in use by the client has its own fid. You will learn more about FIDS when discussing remote files. By now, note how this kernel ensures that all its channels have different fids; So, file servers attending this kernel (including the servers within the kernel) see different fids for different files used by this kernel. Remaining fields of `Chan` are reset, but for the reference held by whoever is allocating the channel.

`pipeattach`
  `devattach`

- `dev.c:79`
  The channel allocated is to be given to the client (the caller of `attach` for `devpipe`). In the same way the client identifies files by FIDs, the server identifies files by QIDs. A QID is an unique number within the server, identifying a file on it. The `qid` field of `Chan` holds the QID for the file pointed to by the channel. In this case, it is a directory and the convention is that directory QIDs have the `CHDIR` bit set. The `path` of the Qid is just `CHDIR` and its `vers` is zero.

- `dev.c:80`
  The `type` in the channel is used to index back to `devtab` and obtain 9P procedures for this channel. The index is the one for `tc` (`|`) as found by `devno`.

- `dev.c:81,82`
  The name for the channel is an absolute path for the file serviced. By now, it is the name for the device root (`#|` in this case).

- `dev.c:83`
  The channel returned. All this code in `devattach` is the typical work that all `attach` routines must do. `dev` is giving the author a means to share that code.

`pipeattach`

- `devpipe.c:62,64`
  The channel is setup, but `devpipe` has to do its job. First, allocate a `Pipe` structure. (`exhausted` raises an error with the appropriate message.)

- `devpipe.c:67,77`
  `Pipes` have two queues because they are bidirectional, unlike UNIX pipes. `qopen` creates a new queue, using the configured size. Queues are discussed later.

- `devpipe.c:79,83`
  `path` relates to the path field in the QID. It identifies a file within the server. In this case, a pipe has two files (`data` and `data1`). To give each file an unique `path` value, the author increments `pipealloc.path` every time a pipe is created.

  `NETQID()` *Builds a `Qid.path` from two values.*

  The macro `NETQID` takes two values and builds a `Qid.path` field. The reason for that is that it is handy to use `Qid.path` as two fields, one specifying the file and another specifying the file type. In this case, `Qdir` is used as the type for the pipe root directory, `Qdata0` and `Qdata1` are used for the data files for the pipe. By looking at this tiny field within `Qid.path`, `devpipe` knows which kind of file is being used. Regarding the other little field in `Qid.path`, specifying which file is the QID for, the value is set to $2 \times$ `path`; even values refer to one data file for the pipe, odd values to the other. Nevertheless, in this case the `CHDIR` bit is kept set (this is the root directory for this pipe).

- `devpipe.c:84,85`
  The `aux` field of `Chan` is provided to let the drivers place there their state for the channel. In this case, the pipe structure is linked there. As there are no multiple "pipe devices", the device number is set to zero (another alternative would have been to use `dev` to multiplex the device among multiple pipes).

- `devpipe.c:86`
  So, after calling `attach` for the device, the client (the kernel) has a channel to the root of the specified file tree in the server.

Navigating

Once the client has a channel to the root directory, the client can move it through the file hierarchy by issuing `walk` requests. More precisely, the client would `clone` the channel (to keep the channel to the root intact) and then `walk` on the clone.

By using `walk` to move a file descriptor (a channel, with a fid) to point to a different file in the server (to change its qid), navigation of paths can be done in a more simple way that it is done by protocols such like NFS [16] or RFS [14]. For instance, there is no need to read directory entries, nor to check that an entry is there in the client, nor to open/close intermediate directories. Just one `walk` per path component suffices.

`pipeclone()` *Clones a channel for the pipe device.*

- `devpipe.c:89,90`
  `pipeclone` is the `clone` procedure for the pipe device. The kernel calls `clone` when it wants a copy of a channel it has. The procedure is supplied the original channel (`c`), and the new (clone wannabe) channel (`nc`). It is expected to return the cloned channel unless there are errors.

- `devpipe.c:94,95`
  Recover the state for this channel (the `Pipe` structure), and call the generic `devclone` procedure provided by `dev.c`, which contains common stuff done almost by every `clone` procedure.

`pipeclone`
`devclone()` *Generic clone procedure for devices.*

- `dev.c:86,110`
  To get a clone, first ensure that the channel is not open. Should it be open, somebody could be doing I/O to the file and it is not polite to mess up with the file in the mean time. As the client in this case is the kernel, if the channel was open, it would be a kernel bug; hence the panic.

  Later, if the caller of `clone` did not bother to allocate a `Chan`, `devclone` does the work. All fields are copied; but see how a new reference is added for any `Mhead` structure pointed to by `mh`. Besides, `Chan` links are kept untouched and the clone has no name. The reason for not cloning the name is that it is likely to be computed differently by the caller.

`pipeclone`

- `devpipe.c:96,109`
  The pipe is locked, and a new reference accounted. (Another client is using this pipe). By the way, the if would never be executed because of the panic in `devclone`.

`pipewalk()` *Walks on a pipe device file.*

- `devpipe.c:147,151`
  After cloning the channel to the root, the kernel would probably walk on it to make it point to a different file. As this is almost the same processing for all

devices, `devwalk` does all the job. The only thing `devwalk` has to know is how to obtain directory entries for the file being `walked`. You will understand this right now, while reading `devwalk`.

`pipewalk`
  `devwalk()` *Generic walk procedure for devices.*

- `dev.c:113`
  `devwalk` is a generic `walk` routine. It performs a walk to `name` on the `c` channel. However, it does not know what is the file hierarchy serviced by the device. Therefore, it needs some help from the device to learn what names it is servicing. The help comes in the form of an array of `Dirtab` entries, and a `Devgen` procedure.

- `portdat.h:197,203`
  A `Dirtab` entry contains the name and the Qid for a file. That is mostly what is needed to scan directory entries. Besides, file length and permissions are found here too. The directory table supplied to `devwalk` is a fake one, file attributes are not kept there; they are kept wherever the file server wants. Of course, it is convenient to use an array of `Dirtab` entries for directories, but it is not a must.

- `portdat.h:43`
  A `Devgen` procedure is an iterator for directory entries. It receives an array of `Dirtab` entries, and an index for a file (third parameter). It is expected to both update the channel to point to the i-th file in the directory table, and to fill up a `Dir` structure with attributes for the file.

- `dev.c:118`
  Back to `devwalk`, it first ensures that `c` represents a directory. `isdir` raises an error when the QID has not the `CHDIR` bit set.

- `dev.c.:119,120`
  Should the name be "`.`", the walk is already done: `c` already points to "`.`". The procedure returns true to indicate success.

- `dev.c:121,125`
  Should the name be "`..`", it is `gen` the one who knows how to walk to it. The channel and the directory table are given to `gen`. That is why `devwalk` received the table, to pass it back to `gen`, to let it iterate through the table if needed. In this case, the "file index" supplied to `gen` is `DEVDOTDOT`, which is `-1`. The convention is that `gen` should walk to "`..`" when `DEVDOTDOT` is given as an index. How to do that, only the device (`pipe` in this case) knows.

  A `Dir` structure is supplied to `gen`. `Devgen` routines should fill up the `Dir` with attributes for the file determined by the file index. Once `gen` did its job, the QID is extracted from the just filled `Dir`, and updated in the channel. From now on, `c` points to the file found by `gen`.

`pipewalk`
  `devwalk`
    `pipegen()` *Directory entry iterator for the pipe device.*

- `devpipe.c:112,122`
  In the case of `pipe`, the `Devgen` routine is `pipegen` (see line `:150`). Should the index be `DEVDOTDOT`, the file is the root of the pipe file tree. Why? Because if file was one of the data files, the parent is the root; but if file was the root, its parent is also the root by convention. To fill up a `Dir` structure with file attributes (which is a common job for devices), there is a generic `devdir` routine.

  `devdir()` *Fills up a `Dir` structure.*

- `dev.c:25,43`
  `devdir` simply receives as parameters the qid, name, length, owner, and permissions for the file, and puts all that information into the `Dir` structure. If the channel has the `CHDIR` bit set, the mode field of the `Dir` structure is kept with `CHDIR` set. That is the convention for directories. Access time and modification time are set accordingly with the current time. Every time `devdir` fills up a `Dir` structure, times are updated. The group for the file is set to eve by default. Of course, should the device supplying the file disagree regarding modification time or group id for the file, it can update the `Dir` entry before using it.

- `dev.c:126,127`
  Back to `devwalk`, the name is neither "`.`" nor "`..`". The procedure iterates through the given `tab` to find the file. Starting with directory index zero, it calls `gen` with successive indexes until `gen` returns `-1`.

- `dev.c:128,130`
  The convention is that `gen` should return `-1` when the index given is not valid. In this case, that means that the index is past the entries in the directory: there are no more entries.

- `dev.c:131,132`
  `gen` returned zero. That means that the entry does not exit, however, the index is valid and no major error occurred. So, continue iteration.

- `dev.c:133,138`
  A valid entry was found and `gen` filled up `dir` with attributes for the file. If the name in `dir` is the `name` to walk to, the procedure uses the QID for the file (`dir.qid`) as said by `gen` to make the channel point to that file. A return of true from `devwalk` means that the walk could be done. Iteration is continued (linear search) until the entry is found.

- `dev.c:141`
  Just safety. Be sure that if you reach this line, a false is returned to say that `devwalk` couldn't walk to the file.

- `devpipe.c:124`
  In the case `devwalk` was called for a file not being "`.`", nor "`..`", you saw how it called `gen` (`pipegen` in this case) to iterate. In this case, this line is reached, and `tab` corresponds to `pipedir` (see line `:150`).

- `devpipe.c:34,39`
  `pipedir` has `Dirtab` entries for a typical pipe root directory. It contains two

entries (for two files) named "data" and "data1". Their Qids are `Qdata0` and `Qdata1`, but note that these are actually "file types" to build the real Qids. Their mode is `0600`, which makes sense for pipe data files, and their length is set to zero—just to give them a length.

`NETID()` *Extracts the id from a Qid.*

- `devpipe.c:124,126`
  `NETID` takes the Qid's path, and extracts the file id from the Qid path. Remember that the Qid for a pipe file keeps both the "type" and the "id" for the file. Pipe services multiple file trees (one per pipe), although for its clients, it is servicing just one. The way pipe has to distinguish among different pipes is to use the Qid path. `Qid.path` must be unique for the three files used for each pipe. Let's revisit how are Qid paths built:

  – The root directory for the n-th pipe has $(2 \times \texttt{pipealloc.path}; \texttt{Qdir})$ as `Qid.path`. (cf. line `:83`).

  – The 0th file in the pipe directory has $(2 \times \texttt{pipealloc.path}; \texttt{Qdata0})$ as `Qid.path`. (cf. line `:124`).

  – The 1st file in the pipe directory has $(1 + 2 \times \texttt{pipealloc.path}; \texttt{Qdata1})$ as `Qid.path`. (cf. lines `:124,126`).

  Just `Qdir` and `Qdata` would suffice to distinguish among different files; because the "id" is different for `data0` and `data1`. Nevertheless, the author thought it was convenient to have two types for the data files.

- `devpipe.c:127,128`
  No table given, or index out of range. Return failure.

- `devpipe.c:129`
  `tab` points now to `tab[i]`, the entry for the i-th file.

- `devpipe.c:130`
  The `Pipe` state for the file is recovered from the channel.

- `devpipe.c:131,141`
  The index given by `devwalk`, selected an entry in `pipetab`. For each entry (data0/data1) set the file length reported as the length of the queue associated with the file. `qlen` returns the queue length. The `default` should never execute because the index was within range and `pipetab` has two entries. But just in case something changes, the routine does its best by looking at the length field for the file in the table.

- `devpipe.c:142,143`
  Fill up the `Dir` for the file. The length was computed, the name came from the `pipetab` entry for the file, as well as permissions came from there too. Regarding the Qid, `vers` is set to zero (the author does not care), and `path` is set by placing in it both the `Qid.path` from the table (actually the file type!) and the file id.

Opening pipes

The client using `devpipe`, after attaching to it would clone the channel to the root and walk on it. Once the channel is positioned into the file of interest, the device `open` routine is called. This pattern of usage is common in 9P. `open(2)` would issue multiple requests on its own (`attach`?, `clone`, `walk`, `open`).

`pipeopen()` *Open procedure for the pipe device.*

- `devpipe.c:181`
  `pipeopen` is the open routine for `devpipe`. It receives the channel being opened and the open mode. It should check that permissions allow the requested open mode, and prepare the file for I/O.

- `devpipe.c:185,192`
  A directory is opened (the root), only `OREAD` mode is allowed. If the mode is `OREAD`, it is noted in the channel for further use (by I/O routines) and the channel is flagged as open.

- `devpipe.c:194`
  An open for a data file. A process is about to read/write one end of the pipe. `p` is the `Pipe` structure for the channel.

- `devpipe.c:196,203`
  The process is going to use one queue. The `qref` array keeps reference counts for both queues—because several processes could open the pipe files, and these files should stay as long as at least one process is using them.

- `devpipe.c:206,209`
  `openmode` checks that `omode` has valid bits on it. The `offset` is set to zero so that any read/write would be done at the beginning of the file.

  In fact, the pipe device ignores offsets while servicing reads and writes because pipes are streams of bytes; as you learned before, a file server is free to ignore offsets in read/write requests. The only thing that matters is that the server should offer a consistent view of its files.

There is also a generic `open` routine provided by `dev.c`. Let's look at it.

`devopen()` *Generic open procedure for devices.*

- `dev.c:212,251`
  The `gen` routine is used to iterate through the directory, searching for the file being opened. When the entry is found (same path in Qid), the information found in the `Dir` structure filled up by `gen` is used to check permissions (you know: "rwx" bits for owner, group, others). The routine is doing nothing but for checking that the open can be done and to initialize channel flags and mode accordingly.

`devcreate()` *Generic create procedure for devices.*

`devremove()` *Generic remove procedure for devices.*

`devwstat()` *Generic wstat procedure for devices.*

- `dev.c:253,257`
  `pipe` uses `devcreate` as the routine for file creation. It simply denies permission to create files.

- `dev.c:287,297`
  The same happens for `remove` and `wstat` (9P transactions for removing files and updating attributes).

Read/Write

`pipewrite()` *write procedure for the pipe device.*

- `devpipe.c:306`
  `pipewrite` is the write procedure for `devpipe`. It mimics the `write(2)` system call.

- `devpipe.c:310,311`
  Interrupts should not be disabled. Looks like the author was bitten by a bug supposedly caused by calling `pipewrite` with interrupts disabled, and the author preferred to ensure that wouldn't happen.

- `devpipe.c:312,317`
  The code could be like it is, without line `:314`. However, if `CMSG` is set in the channel flag, the channel is being used to talk to a file server servicing a mounted file tree. In this case, the system would allow the write without posting a note. Why is this necessary? I think that the write could be done by a server to reply to a request issued by the client. Now, the client could have gone and its side of the pipe closed. It is not fair to kill the server with a note in this case.

- `devpipe.c:319,336`
  `qwrite` does all the job. Since only `OREAD` is allowed when opening a directory, this must be a pipe data file—there is a panic just in case. By the way, `qwrite` would block the process when the queue is full because the reader is slow.

`piperead()` *Read procedure for the pipe device.*

- `devpipe.c:265,282`
  `piperead` is the routine called for reading pipe files. Its interface is like `read(2)`. A generic routine `devdirread` is used when the read refers to a directory, and `qread` is the one doing the job of reading from a pipe (data file). `qread` would block the process is there is nothing to be read in the queue—because the writer is slow.

  A write to `data0` writes to `q[1]`, and a read from `data1` reads from `q[1]`. Thus, what is written to `data0` is read from `data1`—and the other way around.

`piperead`
  `devdirread()` *Generic read procedure for the device directories.*

- `dev.c:185,210`
  `devdirread` is called by `piperead` to read from the (root) directory. It reads an integral number of directory entries each time. Line `:191` determines the number of entries (each `DIRLEN` bytes) read so far. The loop controls that at most `n` bytes are placed in `d`, and also increments the file index for `gen`. `Gen` fills up `dir`, and `convD2M` places `dir` information into `d` in a machine-independent format.

There are two read/write routines that do not correspond with 9P requests. They are `bread` and `bwrite`. Their purpose is to read and write blocks of data. By using specialized versions of `read` and `write` that operate on blocks, block I/O can be made more efficient. This is mainly used by the code in `/sys/src/9/ip`, which reads/writes blocks of data received/sent through network devices.

`pipebwrite()` *Block write procedure for the pipe device.*

- `devpipe.c:338,368`
  The routine should write the block `bp`. It is like `pipewrite`, but `qbwrite` is used instead of `qwrite`. The ignored `long` argument is an offset—ignored because pipes are FIFO.

`devbwrite()` *Generic block write procedure for devices.*

- `dev.c:276,285`
  There is a generic `devbwrite`, which simply adapts the request to the device `write` procedure. It also releases the block (because the block is usually allocated by the source of the data and deallocated by the drain of the data). Pipe does not use this procedure.

`pipebread()` *Block read procedure for the pipe device.*

- `devpipe.c:284,299`
  The routine is like `piperead`, but uses `qbread` (not `qread`) to read pipe data files. To read the directory, a generic `devbread` routine is used.

`devbread()` *Generic block read procedure for devices.*

- `dev.c:259,274`
  The generic `devbread` allocates a `Block`, (releasing it on errors) and calls the device specific `read` routine to fill up the `Block`. In the case of data files, the block comes from the queue. This is simply an adaptor to the device `read` routine.

Attributes

`pipestat()` *Stat procedure for the pipe device. Gets file attributes.*

- `devpipe.c:153,175`
  `stat` is the 9P request to obtain file attributes. `wstat` allows to update attribute values, and you now know that this is not allowed for pipe files. Depending on the Qid, `devdir` is called to fill up `DIRLEN` bytes into `db`. The structure is the same that is obtained by reading the directory. `Qlen` is used to provide lengths for pipe data files.

`devstat()` *Generic attribute read procedure for devices.*

- `dev.c:145,152`
  There is a generic `devstat` routine (not used by `pipe`). It uses again a `Dirtab` and `gen` to locate the file of interest (same Qid), and fill up the stat buffer (`db`) with file attributes as filled up by `gen` in `dir`.

- `dev.c:153,171`
  The index was out of range—which is the case when the channel points to the directory and not to any file on it. If the Qid is for a directory, a name is given to it when the channel has no name, the name is set to "`/`" when the channel name is "`/`", and if the channel name is not "`/`", the name is set to the last name after the the last "`/`"—i.e. `elem` is the file name for the path in the channel name. If it was not a directory, it is a true error.

- `dev.c:174,181`
  The file was found—Qids match.

5.5.2 Remote files

What you have read about file systems works without depending on where is the file server (where are the files). The kernel (e.g. for `Fgrps` and the `Pgrps`) use channels to represent files being used, and channels use device operations to implement the functionality needed to operate on files. Until now, you have seen how all device operations are called by procedure calls dispatched by the device type. For remote files, the system works mostly in the same way; the `mnt(3)` device implements 9P operations for remote files.

Initialization

`mntreset()` *Reset procedure for the mount driver.*

- `devmnt.c:64,71`
  `mntreset` is the reset procedure for `mnt`, (`:920,939`). It is initializing the `mntalloc` global, which is an allocator for `Mnt` structures (`:portdat.h:241,253`) used to service remote mounts (Everything else in `mntalloc` was set to zero by the loader). Besides, it calls `cinit` to initialize a cache (caching is discussed later).

There is no `init` procedure (the generic null one is used).

Attach

```
sysmount
   bindmount
       mntattach()
```
Attach procedure for the mount driver.

- `sysfile.c:652,656`
  When `mount` is used instead of `bind`, the file tree bound is serviced behind a file descriptor. That means that 9P transactions must be issued through its

channel to operate on the file tree. The file server can be a local user process, a remote user process, or a remote kernel used as a file server; for the local kernel, it doesn't matter.

Remember that `bogus` contains a channel to the server, the `spec` for the file tree requested, and an indication of whether files are being cached or not; everything `attach` needs to get in touch with the file server and attach to it. The casting is needed because `attach` should receive a character string, but `mntattach` knows it receives an structure and not a string.

By the way, `mount` is the only way to attach the mount driver, as attach is forbidden from "`#M`" paths. Every time you attach to it, you are attaching to one particular file tree, which is in fact serviced through the network.

- `devmnt.c:73,86`
  `mntattach` knows it receives a `bogus` structure and recovers it from the pretended string `spec`. Perhaps a common header (or a comment saying who else is using the structure) would help to prevent errors. The `c` channel corresponds to the file descriptor being mounted. A 9P server sits at the other end of the channel.

- `devmnt.c:88,109`
  The `mntalloc` list is scanned, looking for a entry (`m`) with the same channel and a non-zero `id`. That list contains entries used to handle each mounted file tree. If such an entry is found, it is locked and checked to be still valid (`ref`). The checks for `id` and `c` are repeated because the lock was not held before—to avoid blocking other processes using the entry.

  This loop is an attempt to share the entry (when the same file is being mounted several times); In this case a reference is added, and another channel obtained using `mntchan`. That channel is the one to be used for the root directory of the file tree. The procedure `mattach` issues a 9P attach transaction to the remote file server, and the `CACHE` flag is cached in the channel. You will learn soon how `mntchan` and `mattach` work. Noticed how `eqchan` is not used to compare channels in `mntalloc`? The mount driver is interested on finding an entry with exactly the same channel structure.

- `devmnt.c:111,124`
  No entry with the same channel, so allocate a new entry from either the free list of `mntalloc` or from fresh new memory (the list used now is `mntfree` and not `list`). A new `id` is allocated and placed into the `Mnt` structure created. Can you see how `id` is non-zero, and `mclose` resets it to zero? The `id` check at `:90` is for safety.

- `devmnt.c:127,131`
  Starting to initialize the `Mnt` for this mount. The channel used to talk to the server is recorded in `m->c` (So the loop used above to locate an `Mnt` entry was searching for an entry using exactly the same 9P connection to the server). The flag `CMSG` in the channel states that it is being used to talk to a remote file server. Although the descriptor supplied by the user was closed by `sysmount`,

it could be `duped`, so the flag in the channel is needed to prevent interferences in the 9P conversation.

Remember the check at `sysfile.c:85,89`? When `fdtochan` is used to get a channel for a file descriptor, and it is being used by the mount driver, an error is raised. The mount driver is very jealous about this channel. That happens only when a true `chkmnt` is given to `fdto2chan`, which happens when `sysread9p`, `sysread`, `syswrite9p`, `syswrite`, `sseek`, and `sysfwstat` call `fdtochan`. Routines reading, writing or updating the state are forbidden for channels used by the mount driver.

- `devmnt.c:132,140`
  `blocksize` in `m` is set to the size for data blocks exchanged between the mount driver and the file server. Unless the `spec` string given to `mount` was `mntblk=n`, it is set to `MAXFDATA`. In this case `spec` is set to null, not to disturb the remote server. Perhaps a writable file for the mount driver would be better than using the `spec` string; although the method used by the author allows different block sizes for different mounts.

- `devmnt.c:141`
  All flags are kept in `Mnt` flags, but for `MCACHE`, which is kept in `c`. Surprisingly, only the `MCACHE` flag was set in `bogus.flags` by `bindmount`. Therefore, `m->flags` is now zero.

- `devmnt.c:143`
  Right now, the channel is still there, but when `bindmount` closes it it could go away. This new reference keeps the channel alive.

- `devmnt.c:149`
  `mntchan` returns the root channel to be given to the caller.

```
sysmount...
   mntattach
```

`mntchan()` *Returns a channel to the mount driver impersonating the channel to the remote file tree.*

- `devmnt.c:185,196`
  It uses `devattach` to build a channel for a directory in the `#M` device. This is very important. The user thinks it got a channel to the remote root directory, but he got a channel to the mount driver instead. The real channel to the server (not to the root, but to the server) is kept by the mount driver, as shown in figure 5.5. An important thing here is that `dev` in the channel built is set with an unique id. This id is used by the mount driver to recognize that mount entries remain valid.

- `devmnt.c:150,157`
  Note how to cleanup on errors, the “`cclose`” is done. The recursive call would be to `mntclose`, which would try to use the channel to send a `Tclunk` request. That would make no sense at this point. But perhaps a channel flag could be added to the channel so that `mntclose` would do nothing to a channel not yet set.

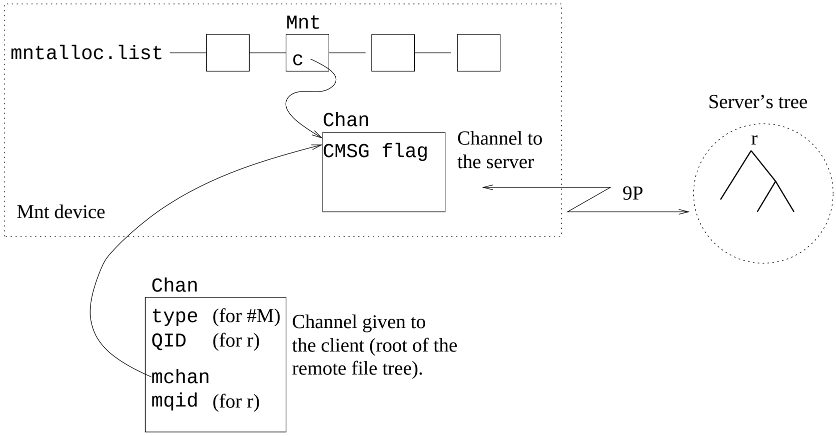


Figure 5.5: The mount driver sits between the server and the client. The kernel uses channels serviced by the mount driver. Requests for these channels are implemented speaking 9P with the file server.

- `devmnt.c:159`
  Starting to speak 9P. `mattach` performs a 9P `Tattach` transaction (client side). If it finishes without raising an error, the attach RPC has been done, `m->c` is a channel to the server, and `c` is a channel to the mount driver.

- `devmnt.c:170,177`
  If the `if` is entered, you knows that:

  1. The channel to the server (`m->c`) is a channel serviced by the mount driver. You know this is the case because the channel to the server has as `type` the index for the mount driver in `devtab`.
  2. The file server at the other end of the channel is `exportfs`. *Tricky!!*. You know this because in `/sys/src/cmd/exportfs/exportsrv.c:574`, `exportfs` is clearing the `CHDIR` bit from the QID for the served directory (`c->qid`). That should never happen because the server is servicing its root directory. However, `exportfs` clears that bit, and the mount driver notices that, does this hack, and repairs the missing bit at line `:172`.

     Perhaps it would be more clean if the `Rattach` reply could carry a string of information back to the client—in the same way the `Tattach` carries an `spec` string. Nevertheless, the protocol and the code are cool, aren't they?

  What is the hack doing? Well, the `Mnt` entry for `c` is closed, i.e. not going to behave as if the remote `c` was mounted through `m->c`, which it was. The `mntptr` (which points to the `Mnt` structure) of the client channel (`c`) is set to point to the one linked at the server channel. That means that `mc->mntptr` is going to be shared among all file trees serviced through `mc`.

  Should this be removed from the kernel, requests for the tree being mounted would be translated to 9P transactions by this mount driver, and such transactions would be written to `c->mchan` (which is `m->c` here). Now, to write the 9P request r to a file serviced by the mount driver, a 9P request s has to be issued to `m->c->mchan`, with the r request being the data. That makes no sense, when you could speak 9P right to the final server[1]. Hence the hack. This is one of the few places in the kernel source when a trick which is not clearly exposed in the system interfaces is being used to prevent the system from doing a silly thing.

Now, what if `mc` is remote (serviced by the mount driver), but the server is not known to be `exportfs`? The author cannot assume that the remote server would multiplex its connection among trees mounted from it, so the best the mount driver can do is to use 9P requests (to the server of the channel for the file server) to write 9P requests for the channel (to the file server).

```
sysmount...
   mntattach
        mattach()
```
Uses a `Tattach` to attach to the server file tree and initializes the channel to point to its root directory.

- `devmnt.c:198,210`
  We still have pending the discussion of `mattach`, which allocates an `Mntrpc` structure to manage the RPC done by the mount driver. The structure is deallocated on errors.

- `devmnt.c:8,22`
  An `Mntrpc` contains among other things two `Fcall` fields (`request` and `reply`) which are the request message and the reply message for the RPC (see `fcall.h`). Those are 9P transactions and replies.

- `devmnt.c:212,215`
  The `request` is a `Tattach`. The `fid` for the `Tattach` is the fid in `c`. Each kernel channel has its own fid (`chan.c:81`), so there is no ambiguity regarding which client “file descriptor” would point to the root of the remote tree. The user name is that of the current process (can `user` contain null characters?).

- `devmnt.c:216,218`
  Fields `ticket` and `auth` are set by `authrequest` to authenticate the client to the server. `m->c->session` is the structure used to maintain authentication/encryption information for the session maintained through the channel. `m->c` is the channel to the server, which is not `c`, that is the channel for the client. The value returned by `authrequest` is used later by `authreply` to check that the reply received for the request is valid and not a fake one. Should it not pass the security check, `authreply` raises an error. `mountrpc` does the actual work of sending the request and receiving the reply.

- `devmnt.c:220,226`
  The channel QID is set from the `qid` in the `reply` (so that it really points now to the server root file). Besides `c.mchan` is set to the channel for the server, now that it is attached. Future requests for `c` would use its `mchan` field to issue RPCs. `mquid` also holds the QID for server’s root directory.

[1] The thing is even worse because `exportfs` is used to export part of the local name space to other processes, and it is likely that `exportfs` would lead to more 9P requests to service its file tree.

RPCs

But how is the RPC done?

`mountrpc()` *Performs an RPC for a mounted file tree.*

- `devmnt.c:607,631`
  `mountrpc` issues the request in `r` and receives any reply for it. It is actually a wrapper for `mountio`, which does the job. First it sets the reply tag and type to poisoned values, in case anyone checks them before the reply arrives. Then `mountio` does the job. Finally, the reply type is analyzed: an `Rerror` causes a raise of the error reported (note that strings are portable!); an `Rflush` interrupts the request; an `Rattach` (`Tattach`+1) returns without errors; and everything else should not happen.

`mountrpc`
  `mountio()` *Performs I/O to issue a 9P request and receive the reply.*

- `devmnt.c:634`
  `m` is used to manage the remote mount point and `r` is the RPC to be done.

- `devmnt.c:638,646`
  The `while` restores the error label after an error is raised. The loop body executes each error, (unless the error is re-raised by `nexerror`). Ignore this by now.

- `devmnt.c:648,652`
  All RPCs r keep in `r->m` a link to the mount entry they are for. Besides, the mount entry keeps in `queue` a lists of RPCs being done. The list is done through the `list` field of `Mntrpc`, as shown in figure 5.6.

- `devmnt.c:655,657`
  The server and client machines could have different architectures. `convS2M` packs the `request` into the buffer at `rpc`, in a machine independent format. The buffer at `rpc` is allocated by `mntralloc` with `MAXRPC` bytes, which is the limit for a message. The panic shouldn't happen, but just in case 9P is changed and `convS2M` is not updated, let the author know.

- `devmnt.c:658,664`
  Some times, the channel used to talk to the server is local (e.g. the file `/srv/kfs` is a channel to the local KFS server). In this case, the channel `m->c` is not serviced by the mount driver and the request would be serviced by the device specific `write` (`:662`). But some other times, the channel used to talk to the server may correspond to a file which is mounted using the mount driver (e.g. `/n/remote/tmp/pipe`). If the channel to the server corresponds to a mount driver channel, the `rpc` buffer is written using `mnt9prdwr`. For mount driver channels, the device specific `write` is not called. Why does the author do so? The answer relates to message sizes as you will learn later.

- `devmnt.c:665,666`
  `stime` keeps the time when the request was sent. The `Tattach` is now traveling

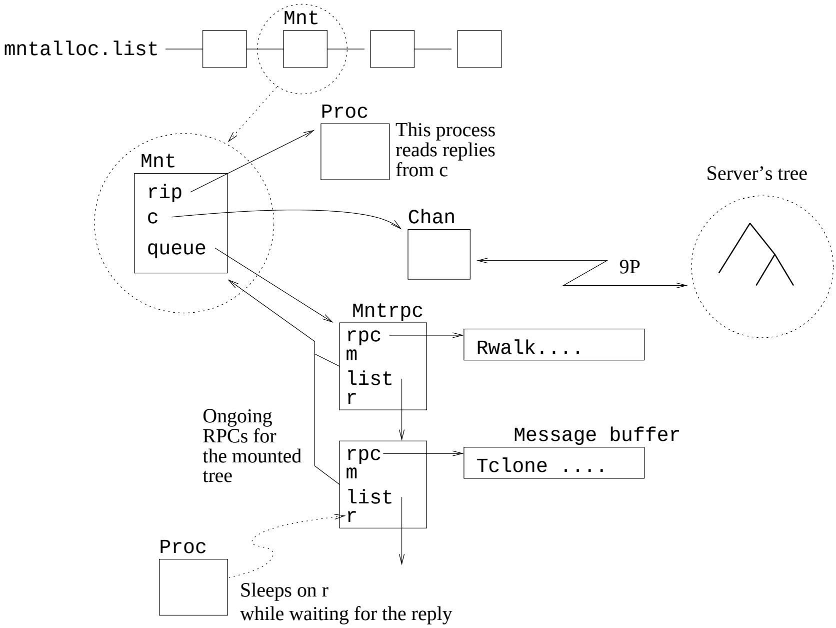


Figure 5.6: `Mntrpcs` are used to maintain the state for ongoing RPCs for a mounted file tree, represented by `Mnt`.

to the file server and `stime` has the current time. `n` bytes were written to the server.

- `devmnt.c:669,681`
  The routine did set `rip` (reader in progress) to nil before. So, unless other process changed the state in `m`, `rip` is nil and the loop is broken at line `:672`.

  If no process is reading a reply for this mount point, the current process is the one reading. From now on, other callers of `mountio` would notice that `rip` is not nil, and enter the loop. They will stay in the loop until the process in `rip` stops reading. Instead of spinning all the time, `sleep` is used to block them until `rpcattn` says that `r->done` or there is no `r->rip` process. So, processes awaiting for RPC replies sleep and read one at a time. `r->r` is the `Rendez` used to sleep. In figure 5.6 you can see a process servicing reads while another is awaiting its reply.

- `devmnt.c:683,686`
  `done` is set to false by `mntralloc`, it is set to true when the RPC should be considered to be done. So, unless the RPC finished, call `mntrpcread` and `mountmux`.

  The first process doing an RPC on `m` is calling `mntrpcread` and `mountmux`, and remaining processes are looping/sleeping waiting for their replies.

```
mountrpc
   mountio
```

`mntrpcread()` *Reads from the 9P channel until gets a reply message.*

- `devmnt.c:692,711`
  `mntrpcread` loops until a valid 9P reply message is read from the channel to the server. As it happen with write, `mnt9prdwr` or the device specific `read` is called depending on whether the channel to the server is a mount driver channel or not. Any reply message is read into the `rpc` buffer of the `Mntrpc`—now that the request was sent, the buffer can be reused for the reply. Lines `:708,709` call `convM2S` to convert the machine independent reply found at `rpc` into a `reply` structure. If the conversion fails (the reply message is not a 9P message), the loop continues and another reply is read. The mount driver is ignoring invalid messages. Once a valid reply is read, the `return` executes and `mountmux` is called.

`mountrpc`
  `mountio`
  `mountmux()` *Multiplexes the 9P channel among multiple RPCs.*

- `devmnt.c:728,763`
  `mountmux` multiplexes the channel to the server among multiple processes doing RPCs. `m->queue` was a list of ongoing RPCs through `m`. The list is searched for an RPC whose request had the tag found in the reply just read. If you read `intro(5)`, you know that 9P replies carry the tag of their request message. If no RPC in the list has a request with the same tag of the reply, no one is waiting for this reply. In this case the routine completes without doing anything, and `mntrpcread` would run again and read another reply message into the `rpc` buffer. The reply (without a request) has been ignored.

- `devmnt.c:740,749`
  If the reply message is for the RPC being done by the process that called `mountmux` (the first reader), `r` would be equal to `q` (the node with matching tag). If `r` and `q` differ, the reply is for some other process. In this case, the `rpc` buffer for the current process is exchanged with the `rpc` buffer of the RPC replied. Besides, the `reply` structure for the process replied is set to point to the `reply` structure containing the unpacked reply. In this case the current `Mntrpc` has an empty buffer in `rpc` to receive another reply.

  This buffer exchange is very important to avoid fragmentation and also to improve performance. The author tries not to allocate/deallocate buffers unless it is really necessary.

- `devmnt.c:750,759`
  `done` is set in the `Mntrpc` whose reply arrived. If it was another process, line `:757` would awake it; then it would notice that `done` is true and pick up its reply. If it was the current process, our `done` is set.

`mountrpc`
  `mountio`

- `devmnt.c:683,686`
  If the reply was for us, the loop breaks. In any other case, the process keeps on

reading from the channel and servicing replies to waiting processes (including himself).

Can you see how when one process has to do some work for himself it tries to do useful work for others too? It would be silly if all processes were spinning trying to read from the channel.

- `devmnt.c:675,679`
  Another processes awaken would check its RPC `done` field. Should it be true, the process has a reply in `r`, and `mountio` returns. Should it be false, the process re-enters the loop and sleeps again if there is another process reading from the channel (`rip`). If there is no other process reading (the reader got its reply and its `mountio` returned), the process breaks the loop and becomes the channel reader, servicing replies for others.

- `devmnt.c:687`
  Unless the reply to the RPC was serviced by another process (a channel reader on our behalf), `mntgate` is called.

```
mountrpc
   mountio
      mntgate()
```
Elects a new reader for the channel.

- `devmnt.c:713,726`
  What happens is that we were the channel reader (servicing requests to others), but our reply finally arrived and we are leaving `mountio`. Another process must read the channel, if other RPCs are pending. `mntgate` awakes the first process found with a pending RPCs in the list. It becomes the reader (see line `:719`).

```
mountrpc
   mountio
```

- `devmnt.c:638,646`
  Should an error occur during all this time, the process calls `mntgate` (if it's the reader) to let other process take its place, because due to the error the current process can abort the RPC and return. But how are errors handled?

  Consider that an `Emountrpc` is raised at line `:663`, after getting back to `:638`, the loop body is entered. The error is not `Eintr`, therefore an `mntflushalloc` is done on `r`.

  `mntflushalloc()` *Allocates a Flush request for a failed RPC.*

- `devmnt.c:769,784`
  `mntflushalloc` allocates an `Mntrpc` for a `Tflush` message, and links `r` (the RPC suffering the error) in the `flushed` field of the `Tflush` `Mntrpc`. A list of flushed requests is being built through the `flushed` field of `Mntrpc`. The field `oldtag` of the `Tflush` is set to the tag of the RPC failing. If the failing request was a `Tflush`, the RPC failing was the one that caused the `Tflush`.

- `devmnt.c:646`
  So on a error, `mountio` runs again with the request being a `Tflush`. If an RPC

fails, a `Tflush` is sent to the server. If the `Tflush` fails, another `Tflush` is sent. All those `Tflush` are for the (`oldtag`) RPC that failed. If the `Tflush` can proceed (the server is running and serviced the `Tflush`), either `:677` or `:689` call `mntflushfree`.

`mntflushfree()` *Releases flushed RPCs*

- `devmnt.c:793,808`
  `mntflushfree` removes from the RPC queue (`mntqrm`) any undone RPC flushed by `r` (a `Tflush`), and releases the structure (`mntfree`). The reply is set to `Rflush`, because `mntqrm` sets the `done` field of the RPC to true—the affected process might check `done` in the mean time, and it should notice that no actual reply arrived.

- `devmnt.c:641,644`
  Should the first `Tflush` abort with an error raised, another is sent. New `Tflush` requests are sent until an `Rflush` is received or an `Eintr` error is raised. In the RPC is interrupted, any flush request is released and the interrupt error re-raised.

`mountrpc`

- `devmnt.c:616,631`
  The RPC finished (note the `Rflush` case). Only if the transaction got a non-error reply, `mountrpc` finishes without raising an error.

Using remote files

After an `attach` is successfully done, the client has a channel to the mount driver, which from the client point of view is pointing to the root directory of the remote file server. Typically, the next things done by the client are `clones` and `walks` to navigate the mounted tree.

`cclone`
`mntclone()` *Clones a channel for a remote file.*

- `devmnt.c:228,229`
  The `clone` done by the client is done by `cclone`, which calls to the device specific `clone`: `mntclone`.

`mntchk()` *Checks that the file is still mounted.*

- `devmnt.c:235`
  `mntchk` (`devmnt.c:883,897`) checks that `c->mntptr` points to an `Mnt` whose id is not 0 (still allocated) and is less than the `dev` in the channel. Line `:192` sets the "device number" for the channel to the `Mnt` id. If the check fails, either the `Mnt` has been released (`id` set to zero at `:402`) or it has been both released and reallocated for a different mount (whose `id` would be bigger than the copy kept in the channel `dev`.

  All mount driver device operations for a remote channel use `mntck` to check that the connection is still alive.

- `devmnt.c:236`
  The routine allocates the RPC structure, as it was done for `mntattach`.

- `devmnt.c:237,240`
  `clone` can be called with or without the "cloned channel", so better be sure that `mntclone` has an `nc`.

- `devmnt.c:241,246`
  Release the RPC on errors, and `nc` if it was allocated by `mntclone`.

- `devmnt.c:248,251`
  A `Tclone` is sent and the reply is received (or an error raised). `newfid` is the FID for the clone, which was set when the clone channel was created.
  `devclone()` *Generic procedure for cloning channels.*

- `devmnt.c:253`
  `devclone` does the job of copying the state in `c` to `nc`. The RPC was just to let the file server know that `newfid` should be understood as another "file descriptor" pointing to the file where `fid` was pointing to.

- `devmnt.c:254,255`
  What? That was done by `devclone`, but it does not hurt. As another channel (`nc`) is using the `Mnt` for `c`, count one more reference. Even if `unmount` is used, `Mnt` will not go away until its reference count gets down to zero—because all channels going through it have been closed.

- `devmnt.c:257`
  To keep the compiler happy—`alloc` is used even if no error is raised. Otherwise, the compiler might issue a warning.

`walk`

`mntwalk()` *Generic walk procedure for devices.*

- `devmnt.c:263,285`
  Regarding `walks`, you now how the global `namec` and `walk` routines work. When the device specific `walk` routine is called, `mntwalk` executes. It issues a `Twalk` RPC, so that `c` would refer to the file named `name`, within the directory pointed to by `c`–i.e. within the directory for `c->fid`. After the `Twalk` is sent, and an `Rwalk` is received, the QID from the `Rwalk` is set in the channel. This kernel now knows that `c` points to the file represented by the new QID, and the server knows that `c->fid` is now pointing to that file.

File attributes

`mntstat()` *Stat procedure for remote files.*

- `devmnt.c:287,307`
  `mntstat` issues a `Tstat` transaction when an `stat` operation is done on the remote file. It works like `clone` or `walk`. The different thing is the call to `mntdirfix` with both the attributes read for the file, and the channel for the file.

`mntdirfix()` *Fix attributes for remote files.*

- `devmnt.c:900,909`
  `mntdirfix` changes some attributes read, to reflect that the file is serviced by the mount driver. In particular, it writes in the last two 'shorts' of `dirbuf`, the letter for the device (`M`) and the mount id. Although the `stat(5)` manual page states that those two shorts are for kernel use, they are used (`libc/9sys/convM2D.c`) to report the device type and device number for the file (Can you guess where does the "M" listed by `ls` come from?)

`mntwstat()` *Wstat procedure for remote files.*

- `devmnt.c:439,457`
  `mntwstat` is also similar, but it issues a `Twstat` instead.

Open and close

`mntopen()` *Open procedure for remote files.*

- `devmnt.c:309,337`
  The QID is set from that in the reply (because the server could have created a new file), the offset is reset to zero, and the channel is flagged to be open. Lines `:333,334` report to the cache that the file is in use for I/O—if the file is to be cached. That is to give the cache an opportunity to invalidate old versions for the file and do other things.

```
cclose
    mntclose()
```
Close procedure for remote files.

- `devmnt.c:427,431`
  This is the device operation called by `cclose` when the last reference to the channel goes away. It issues a `Tclunk` request to let the server know that the `fid` is no longer in use.

```
cclose
    mntclose
        mntclunk()
```
Issues a clunk RPC.

- `devmnt.c:368,388`
  There is no `close` in 9P. `mntclunk` issues a `Tclunk`, or a `Tremove` is the file is being removed (`devmnt.c:433,437`). Seems that `mntclunk` is being reused to issue both kinds of transactions.

Read and write

The actual device specific routines are `mntread` and `mntwrite`, but if you look at `read9p(2)`, you will notice that to encapsulate 9P on 9P without problems because of the maximum message limit, `read9p` and `write9p` have to be used to write 9P requests to a file serviced through 9P.

```
sysread...
        mntread()
```
Read procedure for remote files.

- `devmnt.c:466,500`
  A `read` to a file serviced by mount driver leads to `mntread` as the device specific `read` procedure. `cache` is set if the channel has the `CCACHE` bit set (i.e. if it comes from a tree mounted with `MCACHE`) and it is not a directory. Caching file contents is one thing, but caching directory entries is one of the things that makes distributed file systems complicated (race conditions, too much locking for clients using entries etc). Plan 9 sidesteps that problem by not caching directories. After all, the design of 9P (i.e. `walk`) allows the client to walk paths without needing to cache directory entries. This is also good in that if the file server changes its mind regarding which files exist, its clients would know without any problem.

  If the file is cached, the read is serviced from the cache by `cread`. If the bytes cached (`nc`) do not suffice to satisfy the read request (`n` bytes), a `Tread` is issued to read the bytes not kept in the cache. `cupdate` updates later the cache with the bytes read by the `Tread`. Besides, the device type and number for any directory entries read are set by calls to `mntdirfix`. By making directory reads return an integral number of directory entries, processing of entries is greatly simplified. Compare the routine with the one needed in case `Tread` could return any number of bytes.

```
syswrite...
   mntwrite()
```
Write procedure for remote files.

- `devmnt.c:508,512`
  A write is serviced by issuing a `Twrite` request. Both `Treads` and `Twrites` are serviced by `mntrdwr`. The author reuses code as much as he can: reads and writes have much in common.

```
syswrite...
   mntwrite
      mntrdwr()
```
Issues Tread/Twrite RPCs.

- `devmnt.c:552,565`
  either a `Tread` or a `Twrite` (`type`) on the channel. Note the checks for the mount point and caching.

- `devmnt.c:566,602`
  The routine loops sending `Treads/Twrites`, with the buffer for the request being the `buf` given as a parameter. `cnt` is set with the number of bytes read/writen.

- `devmnt.c:576,582`
  One fine reason for looping. The caller could want to read/write more than `blocksize` bytes (`MAXFDATA`), in which case multiple `Tread/Twrite` must be issued for at most `blocksize` bytes each.

- `devmnt.c:584,587`
  Will not read (write?) more bytes than requested.

- `devmnt.c:589,592`
  The only difference between read and write; not enough to justify two different

routines. For reads, copy the data read into the user buffer. For writes, let the cache know of the bytes writen to the file.

- `devmnt.c:596,601`
  Next time, read/write past the bytes read/writen. The procedure adjusts the file offset and number of bytes processed (`cnt`). The loop continues until `n` is zero, which means that `cnt` is the initial value of `n`; or until read/write could not service as many bytes as requested (no more bytes to read/disk full); or until a note has been posted for the process.

```
syswrite9p
  mntwrite9p
     mnt9prdwr()
```
Reads/Writes 9P requests (encapsulated in 9P).

- `devmnt.c:515`
  `mnt9prdwr` implements both `mntread9p` (`devmnt.c:459,463`) and `mntwrite9p` (`:502,506`); it is also used by `mntrpcread` and `mountio` to read and write 9P requests. It is not a `mnt` device specific procedure, but a generic 9P tool.

  The `sysread9p` and `syswrite9p` system calls call `mntread9p` and `mntwrite9p` to do the work when the channel to the file server is serviced by the mount driver. Otherwise, `sysread9p` (`sysfile.c:335,372` and `syswrite9p` (`sysfile.c:414,441`) call the device `read/write` procedure or `unionread` as they should.

- `devmnt.c:521,525`
  At most `MAXRPC-32` bytes read/written (and for write this limit should not be ever reached). The `Tread` (or `Twrite`) is sent as usually and the reply processed as usually too. So, what's the difference with respect to `mntrdwr`? First, no cache is ever used (would you cache a connection to a server?); Second, the routine does not loop, and it reads/writes a single chunk of at most `MAXRPC` bytes. The routines transmit a 9P message verbatim. If you compare `sysread9p` and `sysread`, you will notice how in no case the the mount driver device specific procedure is called to read the 9P request, and the same happens to `syswrite9p` and `syswrite`. Besides, note that `mntrdwr` uses messages of `blocksize` length (which can be much lower than `MAXRPC`) while `mnt9prdwr` uses messages of at most `MAXRPC` bytes, independently of the configured `blocksize`. Perhaps both routines could be unified into a single one, but the author preferred to keep them separate.

5.6 Caching

In the third edition of Plan 9, authors considered that it was important (due to performance reasons) to cache files mounted from remote file servers. That can save many 9P transactions by satisfying reads and writes from a local cache in the client kernel. The implementation stands at `cache.c`.

In this section, you will be reading the code related to caching in the kernel. Besides a kernel cache, Plan 9 has a user-level program called `cfs` (see `cfs(4)`) that caches remote files. `cfs` is started at boot time and services reads from a local cache (kept on a local disk). This is interesting when it is better to read files from the local

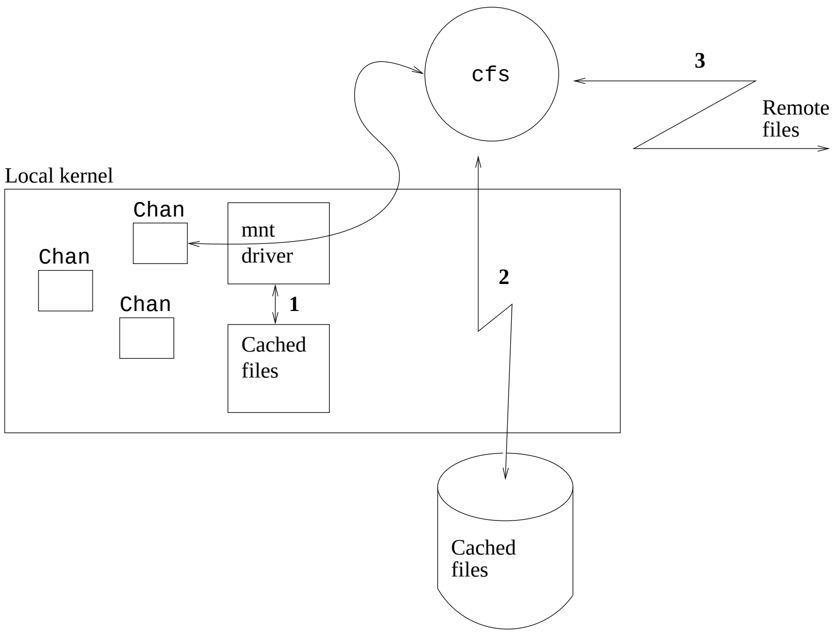


Figure 5.7: Caching remote files. The best thing (1) is to keep them cached in kernel memory. The next best thing (for slow connections) is to keep them cached in a local disk (2). Finally, you always have the network (3).

disk than to read them from the network. Surprisingly, this is not always the case, because when you have a fast network and a fast file server node (plenty of memory) it can be much faster to read a file from the network than reading it from the slow local disk. Nevertheless, for slow network connections the performance improvement can be dramatic.

Regarding the kernel, `cfs` is just a "remote" file server. Therefore, in what follows, I focus just on the caching done by the kernel. Figure 5.7 shows how all the pieces fit together.

```
chandevreset
   mntreset
       cinit()
```
Initializes the kernel cache for remote files.

- `cache.c:105,127`
  You already know that `cinit` is called by `mntreset` at boot time. It initializes the `cache` global (`:39,47`) by allocating `NFILE` `Mntcache` entries, double linking them through `next` and `prev` fields using `head` and `tail` as the list header. `xalloc` is used, and not `malloc`. When the machine is plenty of memory (more than 200MB), the maximum number of bytes to cache in a file (`maxcache`) is not set to its usual value (`MAXCACHE`) but to 10 times more.

5.6.1 Caching a new file

```
sysopen...
   mntopen
       copen()
```
Prepares a cached remote file for I/O.

- `cache.c:210`
  The cache starts to work when `copen` is called. A `copen` is meant to prepare the cache for I/O on a file. It is the mount driver that calls `copen` when a remote file is being opened or created (i.e. before doing any I/O on it). By "remote" you should understand "not in kernel" now. Even if the file is serviced locally by a user program, caching it can avoid unnecessary data copies and context switches.

  The cache does not keep copies of intermediate directories used to walk to the files of interest. Therefore, cache memory is used just for files being really used. The user controls which file systems should be cached (i.e. by means of the `MCACHE` flag).

- `cache.c:216,217`
  The mount driver checked the `CHDIR` bit, but the author ensures that it is innocuous to call `copen` on a directory.

  Plan 9 does not cache file attributes (`walk` works well enough). The alternative would be to cache attributes (including directory contents) and perform `walks` locally. However, this would require that all contents of all intermediate directories walked be sent to the client. Moreover, this would introduce severe coherency problems (all file server clients should see the same set of files, with the same attributes).

- `cache.c:219,223`
  Entries in the cache are linked at `NHASH` hash buckets (`:40,47`). The hash function is a modulus on the channel `qid.path`. Multiple entries are linked at the bucket through the `hash` link of `Mntcache`. The routine searches the hash bucket for any entry for the same file. The check is done using `qid.path`, `dev`, and `type` fields of `Mntcache`, which keep the `qid.path`, `dev` and `type` fields for the cached file. If multiple channels point to the same file, these fields would be same in all of them. `vers` is not compared. The cached file could be a previous version of the file, but it would still be its cache entry. Figure 5.8 shows the structures involved.

- `cache.c:224`
  An entry found. The `mcp` (`Mntcache` pointer) of the channel is set to point to its cache entry; read and write procedures can avoid scanning the whole cache to lookup the `Mntcache` for the channel. The assignment is needed because multiple channels could be opened for the same file. All of their `mcp` fields would have the same value after `copen`.

  `ctail()` *Sets an `Mntcache` at the tail of the LRU.*

- `cache.c:225`
  Remember that `Mntcaches` are double linked on a queue starting at `Mntcache.head`

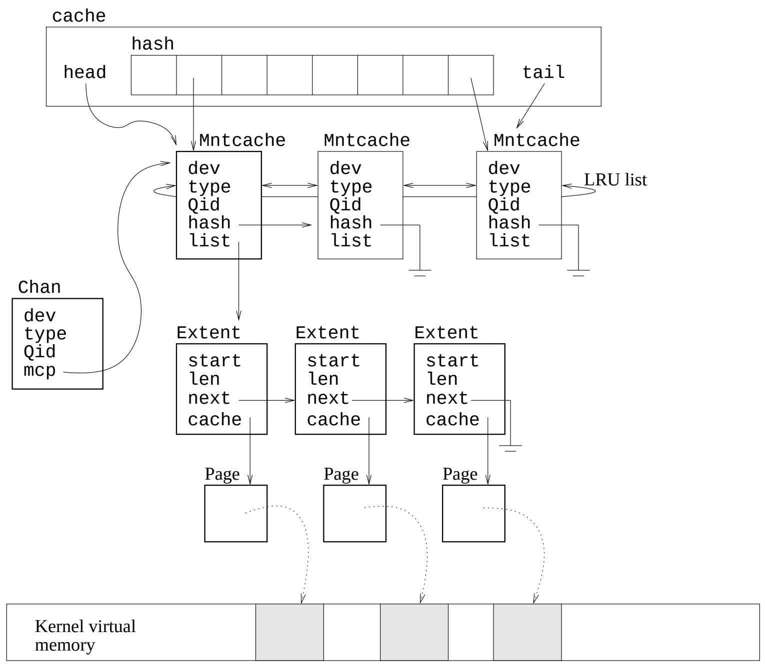


Figure 5.8: Caching for remote files. `Mntcache` structures are caching one file each. They are kept in an LRU list and rely on `Extents`, which use kernel pages, to cache file contents.

and `.tail`. `ctail` (:182,207) unlinks the node given from the list (hence the double links) and links the node at the tail. By doing so, the file last opened is found last on the list of entries. Probably, the author would reclaim cache entries starting from the head of the list. It makes sense not to reclaim this entry because it has been just opened, and is likely to be needed soon.

- `cache.c:226,235`
  Once that the entry is placed on the list, the lock for the cache can be released. If `vers` (also copied in the `Mntcache`) is older than it is in the channel, the file has changed and cache contents are useless. The `vers` field in `Mntcache` is updated whenever new file contents are written through the cache. The routine `cnodata` does the job of disposing any (useless) cached bytes for the file—more soon. The cache is prepared to service the file.

  One quick note about `vers`. `vers` in the channel is updated whenever 9P requests carry back a QID for the file. If other nodes are using the (cached) file too, it could be that the cache contents are actually out of date, and the kernel wouldn't notice. The actual responsible for this lack of coherence is the user, who mounted the file with caching, or the server owner, who did not set the `OCHEXCL` bit in the file being cached. To maintain a set of distributed caches in coherent state is just too expensive and complex. The tools the author gives you allow you both to cache files and to use them coherently, you only have to use the tools in the right way. Other distributed file systems tried to do distributed caching, but they either supplied "session coherence" (i.e. only after a close can others see our changes) or were so complex that a node failure could bring the whole system down.

- `cache.c:239,248`
  No `Mntcache` entry found for the file. Should use one of the existing entries to service the file. The hash bucket for the head of the `Mntcache` list is searched. If the head entry is there, it is removed from the hash list. You should note here that it is the head of the list the one reused. A file could be loosing its entry in favor of another one. You should note also that used entries are linked into the hash bucket (hashing with the QID).

- `cache.c:250,252`
  The old file (if the entry was used) is forgotten. Now this entry is for the file represented by the channel.

- `cache.c:254,257`
  The entry is linked into the hash bucket for the `c` channel, where it belongs now. `ctail` is used to move the entry to the tail of the list, as it has just been used. By allocating from the head, and moving the entry to the tail whenever it is used, the author is doing a "Least recently used" policy to maintain cache entries.

- `cache.c:259,269`
  Finally, `mcp` in the channel is set to the entry allocated for it and any extent linked into the `Mntcache` (which would be for the previously cached file) is

released. The `cache` is unlocked as soon as no pointer is being moved. The entry has to be kept locked because extents are being released.

Extents

`copen`
  `cnodata()` *Invalidates a cache entry.*

- `cache.c:166,180`
  We had pending the discussion of `cnodata`. The cache does not cache bytes, but `Extents`. (`cache.c:16,24`). An `Extent` is a `len` bytes portion of the file starting at `start` offset. All cached extents for a file are linked through their `next` pointers, at the `list` field of the file's `Mntcache`. The routine is simply calling `extentfree` for the whole list.

  `extentfree()` *Deallocates an extent.*

- `cache.c:63,71`
  `extentfree` is releasing the extent. It links the extent into the head of the extent cache (`:50,56`). The next time the cache allocates `Extents`, the ones from the `ecache.head` will be reused. `extentalloc()` *Allocates an extent.*

- `cache.c:73,102`
  Initially, the `ecache` has all its fields set to zero by the kernel loader; the first time `extentalloc` is called, it would notice that the free list (`head`) is empty, and allocate `NEXTENT` extents.

  The cache could do the same for the `Mntcache` entries; or alternatively, `cinit` could also contain lines `:81,92`. One reason the author could had to initialize extents this way, is that the user could never use the `MCACHE` flag for mounts, and the cache would never be used. But in that case, there is no need to keep `Mntcache` entries allocated.

  The actual allocation of the extent is done at lines `:95,98`. The routine clears the contents of `e` (security first!).

5.6.2 Using the cached file

The next time the cache works, is when `mntread` calls `cread` (`devmnt.c:480`) and `cupdate` (`devmnt.c:489`), and when `mntrdwr` calls `cwrite` for `Twrites` (`devmnt.c:592`).

Reading from the cache

`sysread`
  `mntread`
    `cread()` *Services a read from the cache.*

- `cache.c:288`
  You know `cread` is called to read from the cache instead of using `Treads`, when feasible—i.e. when the cache has the bytes.

- `cache.c:297,298`
  `off` is the file offset where to read and `len` is the number of bytes. As only the initial `maxcache` bytes for the file are cached, `cread` ignores any request which passes that limit. Perhaps the condition is too strong, as `off` could be below `maxcache` and part of the `len` bytes could be cached.

- `cache.c:300,302`
  Either the `mcp` has the `Mntcache` to use, or it is nil—stating that the channel is not being cached.

`cdev()` *Checks that the cache is still for a channel.*

- `cache.c:304,308`
  `cdev` tells whether the entry is valid as a cache for the given channel (`:273,285`). It checks `path`, `dev`, `type`, and `vers`. `Mntcache` entries are stolen from the head of the LRU list. The entry for the channel could have been reused and the only way to know is to check these fields. (The alternative would be to iterate through channels, or to link channels using an `Mntcache` entry, which is more complex and inefficient).

  When an `Mntcache` is stolen from a channel, that channel remains uncached—`cread` (also `cupdate` and `cwrite`) ignores it and does not allocate another `Mntcache` for it. This is a fine way of preventing `Mntcache` entries from being stolen repeatedly due to reads and writes; that only happens during `open`.

- `cache.c:310,321`
  The list of `Entents` for the entry is searched for an entry containing `offset` (the first byte read). If there is no such entry, there is nothing of interest cached for the file. Extents are sorted accordingly with their addresses ([`start`/`start+len`]).

- `cache.c:323,361`
  Starting to use the cache. `cread` copies bytes from the `Extent` located previously (and following ones if necessary) to the read buffer. `total` keeps the total number of bytes serviced, and `len` maintains the number of bytes yet to be read.

- `cache.c:324,331`
  Each extent uses at most on page to cache file contents. By using extents instead of pages in `Mntcache`, the author can keep track of byte ranges cached for the file. `cpage` returns the `Page` structure for the extent; should it be nil, the extent does not have memory caching anything and it is removed from the list (`:327`) and released. If an extent did loose its page, the author assumes that any following one (which could contain cached bytes) has lost its page too. `cpage` (`cache.c:156,164`) just calls `lookpage`, which is discussed in the memory management chapter, to lookup the page used as a cache.

  The author uses just pages to do caching. Other systems use several kinds of caches, which in the end, are using pages too. By using always pages to do caching, actual caching has to be implemented just once: by caching pages. In the next chapter you will see how `Images` (used to keep a memory image of used files) are using pages too, as segments do. Simple, isn't it? If you don't agree, try to think how caching could work if the author did cache "files" or "blocks" instead of pages coming from files—you should exercise open/close/read/write requests on this imaginary cache hierarchy.

- `cache.c:333,336`
  The extent has a page caching some bytes. `o` is set as the offset within the extent corresponding to the `offset` in the file. `l` determines how many bytes can be read from the `Extent`.

- `cache.c:338,344`
  `kmap` ensures that the cached page can be read from the kernel (you know it's a nop here), and the author ensures that the page is released and kunmapped (nop) on errors. `cpage` returns a page, and it should not be unmapped while the routine is using it; `putpage` lets the memory system know that the page is no longer in use by the caller of `cpage`. See lines `:348,351` too.

- `cache.c:346`
  Here is the read from the cache! memory copied from the extent to the read buffer.

- `cache.c:353,361`
  After memory has been copied from the extent, if the read request needs more bytes, go to the `next` extent which has the following bytes. Note the check: there could be no next extent, or it could be that the next extent is not caching the bytes right after the current one (it does not start at offset, but starts later). In this case there is nothing more cached for this read.

  When some of the following bytes are missing from the cache, the author refuses to check if any posterior extent would have bytes within the range read. The benefit may not be worth the effort. Besides, the caller of `cread` assumes that it reads a contiguous initial portion of the region being read; the trailing portion is read by the mount driver. Any change here to bypass holes in the cache would require changes in the mount driver too. The author assumption is used to keep the code simple, yet provides effective caching (You should take into account that most applications read entire file contents).

Updating the cache

```
sysread
   mntread
       cupdate()
```
Updates cache contents. (for `cupdate()`)

- `cache.c:460,478`
  `cupdate` is called by the mount driver after reading the trailing portion of the file region being read—that portion was missing from the cache and `cupdate` adds it to the cache. Any further `cread` would now find that portion too. Initial checks are like those in `cread`, for the same reasons.

- `cache.c:483,489`
  The `Extent` list for the `Mntcache` entry is to be kept sorted by file offsets cached.

- `cache.c:491,499`
  `f` is the extent starting past the offset added to the cache (if any), and `eblock` the end of the new portion cached. So, if the portion read overlaps with the next extent, only bytes missing up to the start of that extent must be inserted (remember that the author stops at the first hole while reading). If all the portion being updated in the cache overlaps with the next extent, just forget about the update. This could happen because several processes could read from the channel, find the hole in the cache, issue `Treads` for the file, and try to update the cache. Only one copy should be added.

- `cache.c:501,509`
  These lines are the special case for insertion when it has to be done at the head of the extent list. `cchain` does the actual work of allocating extents and memory to cache the bytes updated. It returns a pair of pointers to the first (returned value) and last (in `tail`) nodes linked as a (sub)list to be inserted. When inserting at the head, the next of the last new node is the first node of the list (`f`, although using `m->list` would be more clear); and the new head node is the first of the list allocated. `cchain` is discussed later.

- `cache.c:511,522`
  Not the first node, so `cupdate` ensures that this does not overlap with any previous node (as it was done before with any posterior node).

- `cache.c:524,540`
  `ee` was set as the end of the previous extent (`:512`). If `offset` is precisely that, the updated portion is contiguous to the previous extent. If the previous extent length is less than the page size, it could accommodate up to `BY2PG - p->len` bytes for the updated portion. Do so. It could be the case that it could accommodate all the updated portion, in which case there is no need to allocate more extents (`cchain` would be never called). `cpgmove` is just a `memmove` that gets and kmaps the extent page while copying memory (`:435,457`).

- `cache.c:524,547`
  As it was done with the head, `cchain` allocates more extents to accommodate bytes updated. This means that the previous node was either not contiguous or not with enough space on its page to cache all the updated bytes. How does `cchain` work?

```
sysread...
   cupdate
       cchain()
```
Creates a chain of extents to update the cache.

- `cache.c:368,381`
  You know its interface. It loops creating extents to keep more bytes from `buf` until the number of bytes yet to be updated is zero. It is not considered a serious problem if no new extent can be allocated; in this case, nothing more is cached. When all `NEXTENT` extents are in use, nothing more is cached.

- `cache.c:383,387`
  An extent caches at most one page. `auxpage` returns a page that can be used by the caller. Should no page be available, there is no need to keep the the extent.

- `cache.c:388,409`
  After noting in the extent what it would be caching, the `bid` field of the extent is set to the `pgno` counter in `cache`. This value is also noted in the `daddr` (disk address) field of the `Page` used, and can be used both for consistency checks (`:160,161`) and to lookup a page with a given `bid`. `pgno` is incremented not by one, but by `BY2PG` instead; it is being used as the address in a fictitious device where all pages used by the cache are kept in allocation order.

- `cache.c:410,417`
  The memory updated in the cache is copied into the extent page.

- `cache.c:419,420`
  Before releasing the page, `cachepage` is used to add the page to a page cache. That is discussed in the virtual memory chapter, but note how extent pages are kept in the page cache. Extents are just adapting the page cache to serve as an extent cache for remote files.

Writing to the cache

```
syswrite
   mntwrite
      cwrite()
```
Updates the contents of a cached file (new version).

- `cache.c:551,574`
  `cwrite` is called to write to cached files. It increments `vers` both in the `Mntcache` and in the channel QID, as the file is being updated.

  The mount driver calls `cwrite` just for `Twrites` on cached files, and that is the case when it increments the QID `vers` for the file; if somebody else is using the file and the server is incrementing its `vers` field, this client wouldn't notice. Other file systems tried to enforce coherence to the limit, and as a result, forced clients to be blocked while transactions for other clients were ongoing. In Plan 9, if this relaxed coherence model is a problem, the application should use `OEXCL` to ensure that only one process at a time has the file open—or rely on any other synchronization means.

- `cache.c:576,599`
  The first extent for the portion written is located. If such extent is not the first one or does not exist (`:583`), the routine tries to use any final hole in the page for the previous extent—if contiguous. This is the same of `cupdate`.

- `cache.c:601,621`
  The portion written could overlap an extent which would be past the one being written. This could happen if the file has been read into the cache (updated) and another process writes the file or the reader does a seek to rewrite the file. The new bytes written are the valid ones, and any previous copy has to be released. `extentfree` is used to release the extent, rather than reusing it. In that way, `cchain` can be used to add more extents to the cache as it was done in `cupdate`.

I think that `cupdate` and `cwrite` could be serviced by a single routine, like happens typically with `read` and `write`. But the author may disagree.

5.7 I/O

You now know how files work in Plan 9, but you still have to look at how actual I/O is done. Prior to the 3rd edition, Streams [13] were used as the framework to do I/O (i.e. to read/write from devices). In the 3rd edition, streams were replaced by a more simple (and less flexible) queue based I/O module. In this chapter you saw how pipes used `qio` facilities to do I/O, and in previous chapters you saw how the console and serial lines used `qio` too. The kernel uses queues for I/O wherever there is an I/O flow of bytes from a source to a drain. This happens when using devices and also when using artifacts like pipes. I suggest you revisit the pipe device after reading `qio` in this section, so you could fit the pieces together.

5.7.1 Creating a queue

- `qio.c:25,50`
  a `queue` is a flow of bytes from the source to the drain (e.g. from the keyboard device to the reader of the console keyboard). The queue maintains `Blocks`, each with a block of bytes. Queues perform flow control activities; they block the reader when the queue has nothing for the drain, as well as they block the writer when there is no room in the queue. Let's see all this while you learn how routines to use `Queues` work. If you look at the file, the initial part is implementing routines that operate on `Blocks` and the final part is implementing queue routines.

`qopen()` *Opens (prepares) a queue for I/O.*

- `qio.c:740`
  A queue starts its service when a call to `qopen` creates it (`devpipe.c:67` and `:72`). `qopen` receives the maximum number of bytes to be kept buffered by the queue (`limit`). If you look the comment, you will notice that `qopen` uses `malloc` and should not be called from an interrupt handler, because `malloc` could try to acquire locks to allocate memory and cause a deadlock with the process being interrupted.

  `msg` is true if the queue is queueing messages and not bytes. This is an important parameter. For a pipe, it does not matter whether the bytes fed to the pipe were fed in a single write or in a couple of writes; if the reader reads N bytes, it should get those N bytes if they are present in the queue—independently of how were they written. However, for a network transport and other devices, it is important to read what was written, no more, no less. If a network device places two network packets in a queue, and a network transport wants to get the next packet received, it should be able to read just the last "message" written (i.e. the last packet queued). Other way to say this is that queue can either delimit data from different writes or not. `msg` is a way to make `Queue` a generic queueing tool.

The `kick` and `arg` parameters are used to let the queue user do something before flow controlled processes are awaken. This is a convenient thing to have to make more simple the code of the queue users. `devpipe` has nothing special to do and passes `nil` as `kick`. However, the `ns16552` serial device passes `ns16552kick` as its `kick` procedure (to restart serial output).

- `qio.c:744,759`
  `inilim` is the initial value for `limit`; more soon. `state` holds the state for the queue, which is initially `Qstarve` (no bytes in) and `Qmsg` if the queue is message oriented. `eof` and `noblock` are cleared and you will see what this means.

5.7.2 Read

`qread()` *Reads bytes from a queue.*

- `qio.c:860,873`
  `qread` is the procedure to read bytes from a queue. Read the comment. It calls `qbread`, who does the actual job of reading `len` bytes and returning a `Block` with the bytes. `qbread` returns zero if there is nothing more to be read (the queue was closed and nobody would write more bytes on it, and the queue is also empty). `BLEN()` *Returns the length of data in a block.*

- `portdat.h:134`
  `BLEN` is defined to take a pointer to a `Block`, and return the difference between its `wp` and its `rp`. That is the number of bytes yet to be read in the `Block`.

- `portdat.h:123,133`
  It is clear what is happening here, a `Block` contains a series of bytes at `base` (the first one is `base[0]`, and the last one is `base[lim-1]`). Bytes written into the block are written starting at `wp` (which would be initially `base`). Bytes read from the block are read from `rp` (which would be initially `base`, and should never go beyond `wp`).

- `qio.c:873,877`
  `qread` copies the bytes in the block (`len` bytes) to the user buffer (`vp`), and then it releases the block. `freeb` is discussed later.

`qbread()` *Reads a block from a queue.*

- `qio.c:773`
  `qbread` does the actual work of reading from the queue. Besides helping `qread`, it is a queue read routine on its own—it is very useful to implement `devbread` procedures.

- `qio.c:778`
  A queuing lock is gained on `rlock`, where queue readers synchronize. `qlock` maintains a list of blocked processes so that the first who called `qbread` would be the first getting bytes from the queue, if it blocks while acquiring the lock.

- `qio.c:785,807`
  On this loop, an `ilock` is done on the queue, this prevents any interrupt handler from using the queue, because no interrupts can arrive while holding the lock. Once the lock on `q` is gained, no one is messing up with the queue block pointers and they can be used safely. If the list of blocks in the queue (`bfirst`) is not nil, `b` is the block to read from, so break the loop. If there are no blocks (`!b`), and the queue state is `Qclosed` there will never be a block, so the author releases the locks, sets the `eof` flag in the queue and the routine returns zero as the number of bytes read. A read count of zero is the convention in Plan 9 for signalling EOF. When EOF is signalled more than three times, the reader could be ignoring EOF and the routine raises an error instead. `q->err` contains the error string to raise; e.g. `qclose` sets it to `Ehungup`.

  If the state is not `Qclosed`, and there is no block in the queue, this process must wait until the source (the writer) puts more bytes in the queue. So, the author sets the `Qstarve` flag to let the writer know that a reader is starving (waiting for bytes), the `q` lock is released (so that the writer could add more bytes to the queue) and the process sleeps on `rr` until `notempty`. `notempty` lets `sleep` know that it shouldn't sleep if there are bytes to be read in the queue. The parameter for `notempty` is the queue being read. When the process wakes up later, it would repeat the loop, check for `Qclosed` again (the writer could close the queue instead of writing to it) and that process repeats until there is a block in `q->bfirst`.

  `BALLOC()` *Returns the number of bytes allocated to a block.*

- `qio.c:809,814`
  Got a block. The routine removes it from the head of the list (Blocks are linked through their `next` field). `dlen` counts the number of bytes in the queue, as a block of `n` bytes has been removed. `len` counts the number of *allocated* bytes in the queue (i.e. number of bytes from `base` to `lim-1` in all blocks linked)—`BALLOC` (`portdat.h:135`) is a macro counting the number of allocated bytes in a block).

- `qio.c:818,837`
  If the block has more bytes and the queue is not message oriented, remaining bytes (unread) should be kept in the queue. If the queue is message oriented, the reader should read the first message; no matter how many bytes it wants. Any unread portion of the first message is discarded. Line `:819` checks that there are more bytes in the block (`n`) than wanted (`len`); the next line checks that the queue is not message oriented.

  Note the `iunlock` (lock held since `:787`). The lock is released while allocating a block for `n-len` bytes, and copying those bytes into the block allocated (`wp` is advanced to point past the bytes written in the block). The `ilock` is done to mess up with block pointers again while inserted the new block in the queue—holding the unread portion of the previous block. By releasing the lock, the queue can be used by others in the mean time.

  Line `:836` sets `wp` in the block being read to point after the bytes to be read from the block. Should the application write to the block, it would not overwrite the data already placed in the block.

- `qio.c:839,846`
  The writer could be sleeping because there was no free room in the queue to write more bytes. That is signaled by `Qflow` in `state`. Should that be the case, if the used space in the queue is below half its maximum number of bytes, the writer can be awaken. Even though the queue may have (less than `limit/2`) free room, the writer is kept sleeping. That is to avoid sequences of sleep/wakeup/sleep/wakeup/... because the writer is awakened too soon, fills up the empty room, and has to sleep again. The queue state is changed before releasing the lock. After the lock is released, other processes can lock the queue and interrupts can arrive (if they were enabled before the `ilock`).

- `qio.c:848,853`
  With the lock released, any writer sleeping awakened. If a `kick` procedure is supplied to `qopen`, it is called. Usually, a process waiting to write the queue corresponds to a process doing output to the queue (possibly drained by a device). On the other hand, when it is a device the one doing output to the queue (an input device), the kick procedure can be used to resume the device and allow it to put more of its data into the queue—device input would happen at interrupt handlers and it makes no sense to block (put to sleep) an innocent process just because it happened that it was running while a device received an interrupt stating that there are more bytes to add to an input queue.

- `qio.c:856,857`
  Until now, any other reader would be blocked on the `rlock`—one reader at a time!. Now other readers are allowed to enter the critical region and the block with the bytes being read is returned to the caller. If the queue is message oriented, all the bytes in the first block of the queue are returned; otherwise, just the bytes wanted.

5.7.3 Other read procedures

`qconsume()` *Reads from a queue even within interrupt handlers.*

- `qio.c:451,517`
  Within an interrupt handler, `qread` cannot be used. `qconsume` is like a `qread` which can be used by interrupt handlers (e.g. `devns16552.c:475`). First, `qconsume` does not call `qlock`, which may call `sleep` and perform context switches. (Is there any context to switch while you are running within an interrupt handler?) Second, `qconsume` does not call `sleep` to block when there are no bytes to be read—`qconsume` would return `-1` instead. Both routines, `qread` and `qconsume`, can be used on system-call (and trap) handling kernel code, within the context of a process.

  As you see, routines for use on interrupt handlers (and those that must synchronize with them) use `ilock`. If the kernel is servicing a regular system call and a queue routine uses `ilock`, no interrupts are allowed, and no interrupt handler can even try to use a queue routine: no deadlock. If the kernel is servicing an interrupt and the queue calls `ilock`, no interrupts would arrive and there can be no context switch, so there can be no deadlock. That is why the author avoids carefully any call to `sleep` and context switches here.

Of course, other processors can still try to use the queue, but would notice the lock, and would not interfere (they would either block the process or spin to wait a bit).

`qconsume` takes care of being as lightweight as possible. Unlike `qbread`, which would split the initial block into two ones when only part of the block is read, `qconsume` uses the `rp` block pointer to read only part of the block when `len` dictates that. In that way, `qconsume` avoids block allocation. Along with this line, any initial empty block is skipped and linked into the `tofree` list passed later to `freeblist`.

`qget()` *Gets a block from a queue.*

- `qio.c:369,403`
  There is yet another read procedure for queues, `qget`. It is the most simple read procedure, and it is specialized just to get the first block in the queue, if any. It never blocks, and is appropriate for use on interrupt handlers too. `qconsume` can read any number of bytes, but `qget` can only get a block. `qget` is used mainly by the code in `../ip`. The tcp/ip protocol stack uses blocks to store protocol data units (e.g. network packets). `qget` and other queue routines operating on queue blocks, allow the tcp/ip code to use queues to do packet (i.e. block) i/o.

`qdiscard()` *Discards bytes from a queue.*

- `qio.c:408,445`
  `qdiscard` is not a "read" routine, but removes bytes from the queue. It iterates through the queue blocks until `len` bytes are discarded. If a whole block is discarded it is `freeb`'ed, otherwise the `rp` pointer is used to "read" the bytes discarded. There is no synchronization regarding `qread`, and the routine does not block. It is also appropriate for interrupt handlers. This routine is useful for `../ip` code, which discards data when it is known to be received by the peer node (e.g. when data is acknowledged).

`qcanread()` *Is there anything to be read from the queue?*

- `qio.c:1160,1164`
  `qcanread` is a small procedure, albeit an important one. Some queue users would not like to read the queue when that would block them (e.g. `devcons`). `qcanread` returns non-zero when there is something to read. The caller can later call `qread`. There is no guarantee (specially on multiprocessors) that the queue would be still non-empty when `qread` is later called. The caller of `qcanread` must ensure that by any other means if that is important.

5.7.4 Write

`qwrite()` *Writes bytes on a queue.*

- `qio.c:961`
  `qwrite` is called to write bytes in the queue (e.g. `devpipe.c:327`). If it is message oriented, the bytes written are considered to be the message.

- `qio.c:970,989`
  The routine writes at most `Maxatomic` bytes (32K) at a time in the queue. It is reasonable to limit how many bytes can be placed in the queue at a single `qbwrite` to avoid a writer flooding the queue with a request so big that locks are going to be held while acquiring resources to queue an unreasonable amount of bytes (e.g. lots of pages in memory, etc.) It is also good to keep this limit to put a reasonable limit on message lengths, so that readers of message oriented queues do not have to cope with unreasonable long messages. Important lines are `:976`, which allocates a block for the `n` bytes being added at this pass; `:982,984`, which copy the bytes from the user buffer into the block using the `wp` pointer; and `:986` which calls `qbwrite` to do the job of queuing another chunk of (at most `Maxatomic`) bytes. The `while` condition checks for `Qmsg`; if the queue is message oriented, and a message is at most `Maxatomic` bytes, the routine would not `qbwrite` any byte that does not fit into a message.

`qbwrite()` *writes a block in a queue.*

- `qio.c:891,901`
  `qbwrite` adds the block to the queue (e.g. `devpipe.c:354`). The lock used is `wlock`. The queue can be read and written at the same time, but writers serialize their access to the queue (in the same way readers do). Like `qbread`, this routine is useful to implement `devbwrite` procedures.

- `qio.c:904,927`
  While adding the block, a lock on `q` is held. Again, an `ilock` is used (know why?). If the queue is closed, there is no point on writing on it, so the block is released and the queue error returned. The loop keeps the writer there until the block can be added; but if the length of the queue is beyond its limit, no more bytes should be added.

  If `noblock` is set (it was set initially to false by `qopen`), a write on a "full" queue is discarded and `qbwrite` pretends that `n` bytes in the block were written. If `noblock` is not set, the writer of a full queue sleeps until the queue is below its limit—and `Qflow` is flagged so that a reader would wake up the sleeping writer.

- `qio.c:929,936`
  The block is added to the queue and the queue `len` and `dlen` fields are updated. Blocks are added at `blast`—they are read from `bfirst`.

- `qio.c:939,942`
  If a reader is sleeping waiting for bytes in the queue, the routine wakes it up. If there are multiple readers, the first one holds `rlock` while sleeping, so other readers would not even enter to read the queue until the first one is awakened and gets the bytes. That is the reason for having just one bit to signal "readers waiting" "writers waiting".

5.7.5 Other write procedures

`qiwrite()` *Writes to a queue (for console).*

- `qio.c:999,1045`
  `qiwrite` is a version of `qwrite` folded with `qbwrite`. It exists because console routines may want to write on queues during boot time, even before there are real processes (see `devcons.c`). `ilock` is used (to prevent further interrupts too), but there is no `qlock` (read the comment). That means that only the lock on the queue is gained, but in no case the "current process" (which could be simply the flow of control existing at boot time) would call the scheduler within `qlock`. Besides, flow control is not obeyed, the caller will never block because the queue has gone beyond its limit.

  In any case, it works like `qbwrite`, and would wakeup any reader waiting.

`qproduce()` *Writes bytes on a queue even within interrupt handlers.*

- `qio.c:634,683`
  There is yet another routine that writes to the queue, it is `qproduce`. Like `qiwrite`, it does not use `qlock`. It does not enforce `Maxatomic`, and never sleeps. `qproduce` is to `qwrite` what `qconsume` is to `qread`. `qproduce` is intended to be called by interrupt handlers (E.g. `devns16552.c:725`). When the queue goes out of limits the block is not added and an error signalled by returning `-1`. Besides, `iallocb` is used to allocate a block, and not `allocb`. The `iallocb` version of `allocb` knows it runs within interrupt handlers.

`qpass()` *Writes a (list of) block to a queue.*

- `qio.c:519,564`
  `qpass` is the counterpart of `qget`. It writes a block in a queue. There is no lock on `q->wlock`, and there is no call to `sleep` for a full queue. The routine is also appropriate for interrupt handlers. One interesting thing is that the routine adds not just one block, but a list of blocks (`:534,537`) and accounts for that (`:541,546`). Another interesting thing is that it enables `Qflow` not when `len` goes above `limit`, but when it goes above `limit/2` (`:551`).

  Queues get full when the limit is passed. Usually, it is a write which makes the queue go *above* limit the one that sets `Qflow`. In this case, the routine is used to write whole blocks (maybe more than one), so the author takes care not go too far above limit, and half the limit is used as a limit.

  This routine is very useful for the code in `../ip` , to place protocol packets (queue blocks) into queues to be serviced later.

`qpassnolim()` *Writes a (list of) block to a queue without obeying limits.*

- `qio.c:566,606`
  `qpassnolim` is exactly as `qpass`, but does not check for `limit`. In this case the author wants the list of blocks to be written, no matter the queue fill state. I don't know why the author did not add a flag to `qpass`, to avoid checking the limit, and wrote instead `qpassnolim`. Perhaps nobody cared to do code

cleanup in `qio.c`, or perhaps careful measuring suggested the multiplicity of routines. By the way, `qpassnolim` seems to be used for the `../ip` code when the author knows it is okay to overflow the queue to pass data to another part of the protocol stack.

`qwindow()` *Is there room in the queue for writing?*

- `qio.c:1146,1155`
  `qwindow` is to be used like `qcanread`, a caller of `qwrite` can use `qwindow` to see if the `qwrite` would block or not.

5.7.6 Terminating queues

`qhangup()` *Hangs up on a queue.*

- `qio.c:1095,1109`
  `qhangup` is used to state that no one else will write anything more to the queue. However, the queue is kept with any block not yet read. The `err` field is used to report that the queue is hunged up (or the message supplied by the caller), and `state` is set to `Qclosed`. Any reader would notice the `Qclosed` and it will not block waiting for more bytes. Any writer will just discard the bytes being written. `notempty` (`:766`) returns true when the queue is closed, so that `sleep` will consider that there is no need to sleep on a closed queue. The two `wakeups` would wake up any reader or writer sleeping, and they will behave as I just said. Did you noticed the `ilock`?

`qclose()` *Closes a queue.*

- `qio.c:1062,1088`
  `qclose` closes the queue—like `qhangup`. Unlike `qhangup`, it releases any block in the queue (`:1083`). `qhangup` is intended to be used when one end of the queue hangs up, and how `qclose` is more like a "free" routine (e.g. `devpipe.c:228,229`). Another way to see it is that `qhangup` can be used by writers to signal that there is nothing more to come; while `qclose` can be used to shutdown the queue. `Qflow` and `Qstarve` are cleared. Perhaps they should be cleared by `qhangup` too.

`qreopen()` *Reuses a closed queue.*

- `qio.c:1123,1132`
  `qreopen` can be used to undo the effect of a close. It clears the `Qclose` and sets the `Qstarve` and `eof` fields as in `qopen`. The purpose is to reuse closed queues instead of allocating new ones (e.g. `devpipe.c:247,248`).

`qfree()` *Deallocates a queue.*

- `qio.c:1051,1056`
  When the queue is no longer needed, `qfree` does the close and then calls `free`. It can be called for an already `qfree`'d queue, as `free` checks for nil pointers and `qclose` does so too. The comment suggests that perhaps `qclose` should add reference counting and free the queue when the reference goes down to zero.

5.7.7 Other queue procedures

`qcopy()` *Copies bytes from a queue.*

- `qio.c:688,734`
  `qcopy` is used to copy bytes from the queue into a new block. Bytes are not read from the start of the queue. Instead, `qcopy` copies bytes from the given `offset` in the queue. Lines `:701,715` locate the block and "rp" (`p`) where to start reading, and later lines perform the copy. The queue blocks and their `rp` are kept untouched. This routine is used by the `../ip` code, which usually likes to copy data out of network messages.

5.7.8 Block handling

Routines early in `qio.c` perform operations on blocks. They are mostly of interest to protocol stacks using queues as their I/O mechanism. For instance, code adding headers, extracting data, etc. from network messages use these routines. I think you should be able to understand these routines:

- `qio.c:91,133`
  `padblock` takes a block and returns a new block (or the same `bp` if it has enough space) with `size` extra bytes of padding at the front or at the back. That can be used to add headers or trailers.

- `qio.c:138,150`
  `blocklen` uses `BLEN` to return the length of a list of blocks. Some times, specially when a message is traveling through a protocol stack, a message may end up being a sequence of blocks.

- `qio.c:155,174`
  `concatblock` takes care of merging all the linked blocks into a single one. Some routines assume that a message is contained within a single block, `concatblock` can be used to ensure that.

- `qio.c:179,232`
  `pullupblock` checks that there are `n` bytes after the bytes in the block (after `rp`). It allocates a new block if needed. This is useful to add `n` bytes to a message without turning the message into a block list.

- `qio.c:237,273`
  `trimblock` trims a block to a subset of the bytes on it. Useful to remove unwanted headers and trailers. This can be used to trim bytes at the front, at the end, or at both sides.

- `qio.c:278,302`
  `copyblock` copies bytes to a new block.

- `qio.c:304,330`
  `adjustblock` truncates the block to `len` bytes. Perhaps, `trimblock` could be used instead of providing a new routine, although this routine would run faster.

- `qio.c:334,364`
  `pullblock` removes bytes from the front of a block list.

Finally, note how most queue routines update statistics declared at `qio.c:109,14`. Those counters tell the author how intensively are used the routines involved. For instance, if `qcopycnt` goes too far, it may be a symptom that queue copies should be avoided if there is a performance problem involved. Statistics are important in that they let the author know the real usage of the code; most of the author assumptions would not correspond to the real system usage as seen by the statistics.

5.7.9 Block allocation

- `allocb.c:24,56`
  `allocb` is the routine allocating blocks always but for interrupt handlers. The memory allocated is for the `Block` itself, and also for the data to be kept in the block. `Hdrspc` empty bytes are kept allocated besides the `n` bytes requested (the total allocated space is `size+Hdrspc+sizeof(Block)`). That is to allow protocol stacks to place their headers before the data in the block. Should the author not do so, almost every step in a protocol stack would require allocating new blocks, concatenating them, and releasing previous blocks. The system I/O for networks would go unbearablely slow. Another interesting thing is that the routine raises an error (unlike `iallocb`).

  `base` is set pointing past the `Block` in the allocated memory, and `rp` and `wp` are set pointing after the `Hdrspc` (which is computed by subtracting `size` from the limit of the allocated memory).

  The dance around `BLOCKALIGN` is ensuring that pointers are aligned to `BLOCKALIGN` (8) bytes. That can prevent alignment errors on machines that are picky regarding where can integer values be placed in memory. Not the case, but this does not hurt.

  The memory held by the block would be released when `free` is called in the block—the `free` block routine is appropriately set to nil.

- `allocb.c:61,108`
  `iallocb` is a version of `allocb` for interrupt handlers. The difference with respect to `allocb` is that `ialloc` does not raise any error (returns `nil` instead) and sets the `BINTR` flag in the block (some `ialloc` accounting too, admittedly). The flag is only used by `freeb` to do accounting, but is not used for other things.

- `allocb.c:110,140`
  `freeb` calls the `free` procedure, does accounting for `iallocated` blocks, and releases the block. If a `free` procedure is provided, `free` is not called on the block; providers of block storage are responsible to reclaim unused storage. All the pointers are set to `Bdead`, which is a meaningless value that can be recognized quickly when the debugger prints pointer values.

5.8 Protection

In Plan 9, there are several system calls (see `auth(2)` and `fauth(2)`) that have to do with protection. However, before looking at their code, it is better to understand the overall architecture of Plan 9 regarding security. Read also `auth(6)` and `cons(3)`. What I comment here is just what I think you need to know to understand the code.

5.8.1 Your local kernel

Each Plan 9 machine is either a terminal, a CPU server, or a file server. Each machine run its own Plan 9 kernel, customized to perform well for the given task. Terminals are machines used to interface the Plan 9 network to its users. For example, each user runs `rio(1)` (the window system) at its terminal. Terminal machines use services from other nodes in the network. In particular, a terminal uses a CPU server to execute commands on it, and a file server to get files from it. Besides, machines where you run your programs in the network (e.g. cpu servers) use files serviced from your terminal (e.g. your mouse).

Everything is a file in Plan 9, and file permissions are what the system uses to provide protection. Each file has `rwx` bits for its owner, its group, and others (you already know how that works, since you did learn that for UNIX). Thus, one barrier of protection is placed at the file server that services the files accessed.

In Plan 9 there is no 'superuser' as in UNIX. In UNIX, a user with uid 0 is granted special privileges by the system, which has conditionals in the kernel to allow such uid to do almost anything. In Plan 9, no user is granted permission to do everything.

Each Plan 9 kernel is booted by a a user, and that kernel only trusts that user. The user who boots a node is referred to as "eve", as you know. Each kernel services some files, and eve is granted special permissions on those files—noticed the checks for "eve" while reading the code?

By trusting only the user who did boot the node, Plan 9 does not allow other users (nor other kernels) in the network to do things to your local files. Why does Plan 9 give special permissions to eve?

If you have physical access to the system and can boot it, nobody can prevent you from using another system (e.g. `msdos`) and access the disk files without Plan 9 even knowing. Therefore, there is no security breach in allowing you to bypass permissions for local files.

To check permissions for a process accessing a file, each process has a user identifier (`Proc.user`). The initial process belongs to the user `eve`, who booted the machine. That user, types its user name and its password and the boot process uses that information to authenticate the user. Before this point, the boot process belongs to the user "none". Say that the typed user name was `nemo`; once authenticated, the boot process continues and processes forked would run on `nemo`'s name too. To authenticate, the user process uses `auth(2)` services to get in touch with the authentication server (another machine) and gain tickets for the user. While authenticating, `authwrite` (`auth.c:422`) is called to respond to a challenge, and if the reply is ok, the `user` field of the process is changed according to the user who is authenticating. So, each machine trusts the authentication server and itself; it usually trust nobody else.

For terminals, this is mostly what happens. For CPU servers, the user who boots the CPU server has some processes on its name, and must be able to create processes for other users willing to compute on the CPU server considered. What happens, is that the user owning the CPU server is granted permission (by the authentication server) to speak on behalf of the user that wants to compute on the CPU server.

5.8.2 Remote files

According to what I said, other nodes will not trust you. How could they trust you? Actually, the Plan 9 kernel does not care—mostly. As far as the kernel is concerned, your files come from a server (a set of servers actually) speaking 9P through a set of mounted file descriptors. It is you who get those file descriptors by setting up network connections to file servers. If you can authenticate to a file server in the network, and convince it to speak 9P for you, you can later give the descriptor to `mount(2)` and bring the server files to your name space. In principle, your local kernel does nothing to let you authenticate to the remote server and get your 9P session up. What your local kernel does is to check protections for your local files.

As an example, you must first authenticate to a Plan 9 file server to use its files. (e.g. you authenticate with a file server kernel to access your files; you authenticate with your local kernel to get access to files serviced by the local kernel; etc). This can be considered to be a first barrier of protection: convincing the file server to speak 9P with you. Later, the file server will be checking permissions, given the attributes of its files and your (authenticated) identity; you can consider this as a second barrier of protection.

By placing authentication mechanisms outside the system (which only has to handle 9P), and letting you obtain the authenticated connections to file servers, Plan 9 can be as secure (and as insecure) as you want it to be.

One thing the kernel does for you is to keep your tickets—after you gave your user name and password while booting—to authenticate connections for which you already have tickets. Of course, you can still use any other means to protect connections with your file servers, and then mount the connection descriptors.

The code keeping your user id and your ticket (generated from your password) is found at `/sys/src/9/port/auth.c`, with console files serviced by `/sys/src/9/port/devcons.c`. I think you should be able to read that and understand it, provided you understood `auth(6)` and `auth(2)`.

Chapter 6

Memory Management

Plan 9 uses paged virtual memory. Although on Intels there is segmentation hardware, hardware segments are used just to implement protection ring 0 for the kernel and ring 3 for the user—go back to the introduction chapter if you forgot. Hardware segments are not to be confused with process segments, which is an abstraction implemented in software by Plan 9.

Before discussing memory management system calls like **`segbrk`**, **`segattach`**, **`segdettach`**, **`segfree`** and **`segflush`**, I start by discussing how memory management works. You already know a bit about this, from chapter 3. I hope that way you will learn what data structures are involved, and you will understand better the code related to memory management system calls and memory management trap handlers.

During this chapter, you will be reading these files:

- Files at `/sys/src/9/port`

  `fault.c`
  : Page fault handling.

  `page.c`
  : Paging code.

  `segment.c`
  : Process segments.

  `swap.c`
  : Swapping code.

  `sysproc.c`
  : Process system calls.

  `devproc.c`
  : Process device.

  `devcons.c`
  : Console device.

- Files at `/sys/src/9/pc`

  `mem.h`
  : Memory management definitions.

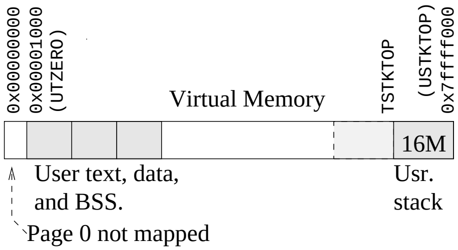


Figure 6.1: The user view of a virtual address space: A text segment with the program code, a data segment with initialized data, a BSS segment with uninitialized data, and a stack segment.

`memory.c`
: Actually discussed at chapter `ch:start`, but you may want to reread it here.

`mmu.c`
: Memory Management Unit handling code.

`trap.c`
: Entry points for MMU faults.

`dat.h`
: Machine dependent data structures.

6.1 Processes and segments

- `../pc/mem.h:29,54`
  To remind you, the kernel uses the paging hardware (two-level page tables) to implement virtual memory. Each process has its own virtual address space, split into two regions, one for the kernel and another for the user. The user portion of the virtual address space is using addresses from 0 to 2G (`0x00000000` to `0x7fffffff`). The kernel portion goes from 2G up to 4G (`0x80000000` to `0xffffffff`). The last two Gbytes, for kernel usage, are shared among all Plan 9 processes, which means that their entries in the hardware page tables are the same[1]. You should remember among other things the identity mapping for physical memory.

  From the point of view of the process, things are different (see figure 6.1). Its 2G of the virtual address space (what it can see) are structured into segments. A process knows it has a set of segments attached at concrete virtual addresses with concrete lengths. For instance, all processes have a text segment (with instructions) at address `UTZERO`, past the first page—which is kept unmapped to catch dereferences for nil pointers. Besides, processes have stack, data, and BSS segments.

[1]It does not work exactly this way, but you will know.

- `mem.h:60,77`
  Do not confuse the process (software) segments with the hardware segments used by Plan 9.

6.1.1 New segments

Let's start by looking at how are new segments created, considering first a stack segment.

- `../port/sysproc.c:324,329`
  The `boot` process was given segments by hand by the kernel bootstrap code and the first thing it did was an `exec` system call to execute the code for the boot process. `sysexec` then calls `newseg` to create a stack segment for the new program. To remind you, segments for a `Proc` are kept linked to its `seg` array, which has `NSEG` entries. If you remember from the chapter on processes, `ESEG` is an slot for an extra segment (`SSEG` is the slot for the stack segment).

`newseg()` *Creates a segment.*

- `segment.c:47,54`
  This procedure creates a segment of a given `type`, `base` and length. It aborts if the size is beyond the maximum size allowed for a segment—`size` is in pages, as segments must contain an integral number of virtual memory pages because the paging hardware is used to implement them.
  - `portdat.h:365,383`
    If you look at `Segment`, you can see how there is a `map` array with pointers to `Pte` structure (see figure 6.2).
  - `portdat.h:323,329`
    A `Pte` contains at most `PTEPERTAB` pointers to `Page` structures, each one responsible of a (virtual) memory page.

    What is happening is that a segment is using a virtual MMU as its data structure. So, the `map` in `Segment` is like a two-level page table that lets the segment hold pointers to all `Page` structures for the pages it has. The reason for using this two-level structure is the same reason the hardware has for using two-level page tables: to save memory yet to be efficient when looking up entries.
  - `../pc/mem.h:104`
    As `map` will have at most `SEGMAPSIZE` entries, and each entry has at most `PTEPERTAB` pointers to pages, the maximum number of pages is the limit checked at `segment.c:53`.

`swapfull()` *Running out of swap space?*

- `segment.c:56,57`
  To implement virtual memory, all pages that do not fit into main memory are kept in a swap file (which could be a swap partition, since partitions are files). `swapfull` (`swap.c:406,409`) returns true when the swap file has less than a one tenth of free space. The kernel refuses to create new segments when it thinks that there will be no space in swap for backing up the segment.

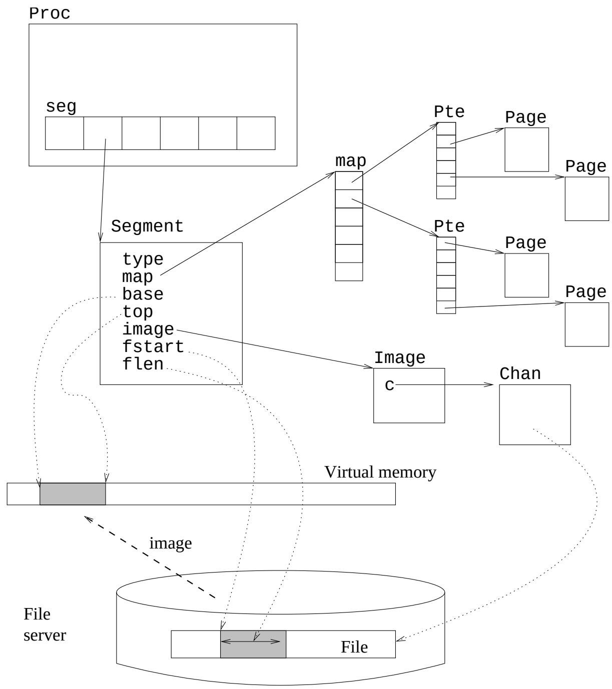


Figure 6.2: A `Segment` maintains a virtual MMU data structure holding `Page` structures for pages in the segment.

- `segment.c:59,64`
  The new `Segment` is created. It is reference counted because processes can share segments. The type, base address, and top address (the first address past the last address in the segment) are kept in the `Segment` structure.

- `segment.c:66`
  `mapsize` is set to the number of `map` entries (`PTEPERTAB` entries each) needed to hold `size` pages. Part of a the last `Pte` in `map` could be unused.

- `segment.c:67,73`
  `nelem` is a macro returning the number of entries in an array (`portfns.h:173`). If more entries are needed than the number of entries in the (small) `ssegmap` array kept in `Segment`, `map` is allocated to contain twice the entries needed—unless that value goes over the maximum number of entries, in which case, just the maximum is allocated.

  The author is allocating twice the space required because he thinks that in the future the segment could grow. In that case, the author wants to be sure that allocated space would suffice most of the times. That makes unnecessary to reallocate existing entries. Right now, all pointers in the map are nil, because `smalloc` is used. `mapsize` holds the number in entries in the `map` array.

- `segment.c:74,77`
  What happens when there are less entries in `map` than entries in `ssegmap`? `map` is set pointing to `ssegmap`, instead of allocating a fresh new `map`. The author made a provision for small segments, so that they do not incur in the overhead of allocating/deallocating maps. Small segments are serviced just with the `Segment` structure. This is also a help to fight fragmentation, not just execution time, because less structures have to be allocated.

- `segment.c:79`
  Finally, a new `Segment` structure with enough entries in `map` (all nil) is returned.

6.1.2 New text segments

`sysexec`

- `sysproc.c:381`
  During `sysexec`, a new text segment for the code found in the `tc` channel is created.

`sysexec`
  `attachimage()` *Creates a segment attached to a file image.*

- `segment.c:246`
  `attachimage` tries to create a new segment of the given `type`. Unlike `newseg`, `attachimage` attaches a file image to the segment, so that the segment would appear to contain whatever is contained in the file referenced by the channel `c` (see figure 6.3).

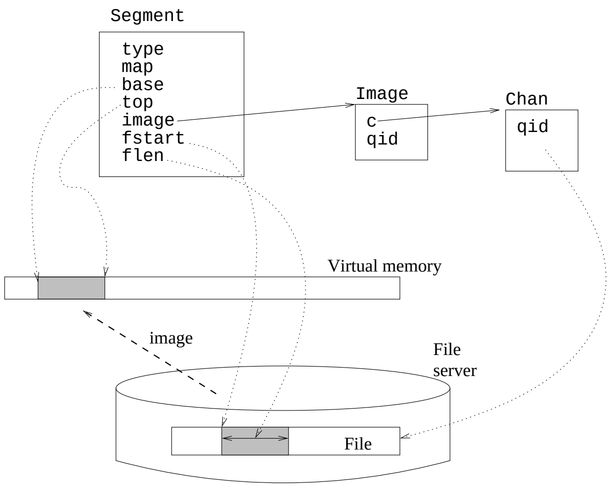


Figure 6.3: A `Segment` can be attached to a file image.

- `segment.c:251,252`
  `imagechanreclaim` is discussed later—it is just closing unused channels which were used to fill up other images.

- `segment.c:254`
  Locking the `imagealloc`, which is an allocator of `Image` structures.

- `segment.c:19,31`
  The `imagealloc` contains a free list of `Images`, and a hash table for `Images`. You will be seeing how it is used.

- `portdat.h:309,321`
  An `Image` represents the image in memory for a portion of a given file. As `attachimage` takes a channel to a file and builds a segment upon its contents, `Images` are very important here. Just note how an `Image` contains a channel to the file used as the source of data for the image (`c`), and how there is a link to the `Segment` which is using the image of the text file.

- `segment.c:260,261`
  All `Images` under `imagealloc` are kept hashed on the QID of they file they come from. `ihash` selects the appropriate `hash` entry in `imagealloc`. Each hash bucket has images linked through the `hash` field of `Image`. The author is searching for an `Image` which comes from the same file; therefore, the `qid.path` of the channel is compared to the `qid.path` of the `Image` (Images keep the `qid` for the file they are maintaining).

- `segment.c:262,270`
  Should an `Image` for `c`'s file be found, the image is locked and the QIDs compared (with `eqqid` this time). The check at `:261` was a quick guess to avoid locking all images in the cache just to find out that they are not the ones of interest.

In the real check, the QID, the QID for the channel to the mounted file (which could be zero if not mounted), the channel to the mounted server and the device type are compared. If you remember, two files are the very same file if their QIDs match, they are serviced by the same device type, and the actual server is the same. This is the check being done here. Lines `:264,265` are needed to distinguish between different channels going through the mount driver, but pointing to different files on different servers.

If such an image is found, a reference is added and the routine continues at `:298`, with the image locked. The author knows that only a copy of the file text has to be kept in memory. If you run different processes for the `/bin/rc` program, the text has to be in memory just once (because it can be shared due to its read-only nature). Therefore, all such text segments would be sharing their `Images`, which would be just a single `Image` for a channel going to `/bin/rc`.

- `segment.c:274,285`
  No `Image` was found for the text, so the author allocates a new one. The loop, which calls `imagereclaim`, tries to deallocate `Images` back to the free list. The process doing an `exec` would be looping calling `imagereclaim`, and letting other processes run until it can get an `Image` from the free list. This process could do not much else, because `exec` did commit to execute a new process and it cannot be even aborted. The choice is either wait for an image or die. The `imagealloc` lock is released while allowing others to release images.

- `segment.c:287,299`
  The new image is locked, a reference added to it, and its fields initialized. The `Image` is linked into the hash bucket corresponding to the file's qid. `imagealloc` is unlocked but the image remains locked.

- `segment.c:301,312`
  `i` could be a newly allocated one, or one reused (shared) from the hash. If the `Image` comes from the hash, its `s` field points to a text segment, which would be shared, so a new reference is added to it. If the image is new, it has no segment yet, so `newseg` allocates a new segment of the type desired, and its `image` field is set pointing to the image. You can go from the segment to the image using `image`, and back to the segment using `s`. The `waserror` handling code does not call `nexterror`, because the caller is gone during `sysexec`. Instead, the process is killed on errors.

- `segment.c:314,315`
  All set. Either a fresh new image created or a previous one shared.

`sysexec`

- `sysproc.c:382,387`
  `attachimage` returns the image locked. The caller adjusts the fields `fstart` and `flen` in the `Segment` using the image to record the portion of the Image which should be used to fill up segment memory. In the case of the text segment, its bytes come right from the beginning of the file—even the header is "mapped" within the text segment (looks like the real text file, doesn't it?)

- `sysproc.c:393,397`
  In the case of the data segment, its bytes come from the text file, past the code.

  To summarize, an `Image` is an image of a program file kept on memory, it is attached to one or more segments, and each segment attaches to a portion (`fstart`, `flen`) of the file image. We will get back to images later.

6.2 Page faults or giving pages to segments

6.2.1 Anonymous memory pages

I use the term “anonymous memory” (as others do) to refer to memory which does not come from a file. For example, text and data segments have their contents coming from the file with the program being executed. Stacks and BBS segments, on the other hand, are created with cleared memory, which does not come from any file. Let’s pick up the stack segment as an example to see how it gets some pages.

`sysexec`

- `sysproc.c:329,333`
  Once the process has a segment, it can reference addresses within the segment. When are pages given to segments? When does a segment get actual memory? If you read the comment, you’ll get a hint.

  Segments are made of virtual memory pages. Those pages can be either at physical memory, or at secondary storage. For the stack segment just created at `:329`, there are no pages yet. But the code below will write to stack addresses!

`trap`

- `../pc/trap.c:218`
  The first time the kernel writes to an address within the first page of the stack (the last page in the segment due to stack growing direction on intels), a page fault trap is generated. That makes sense since the hardware MMU has the translation for the stack page marked as absent.

- `trap.c:242,245`
  The handler in the `Vctl` for the page fault trap is called—the handler was set at `:170` to be `fault386`.

`trap`
`fault386()` *Services a 386 page fault.*

- `trap.c:435,442`
  `fault386` receives the `Ureg` for the faulted processor context (which would be code running within the kernel in the example). Due to Intel nature, the faulting address is not saved by the hardware in the `Ureg`, but is located in the `cr2` register instead. The routine saves the address in `addr`—another page fault in the mean time would overwrite the `cr2` and loose the previous faulting address.

- `trap.c:443,445`
  `user` is true if the saved context had the `UESEL` as the code segment selector (i.e. it was code running at user-level). If the page fault happened while running inside the kernel and `mmukmapsync` can handle `addr`, nothing else is done—the page fault should be fixed. `mmukmapsync()` *Synchronizes kernel maps for the MMU.*

- `mmu.c:273,288`
  We will see later, but `mmukmapsync` tries to get the hardware page table entry (`pte`) for the faulting address. It uses the `pdb` pointer for the boot processor, which points to the "prototype" page table kept for processor 0. `mmuwalk` walks through the page table to get the entry. If `pte` is nil, there is no second-level page table and `mmukmapsync` does nothing. If the entry in the second level page table is is nil, the same happens. In our stack page fault example, there is no entry added for the new stack page. Therefore, in our case, `mmukmapsync` does nothing.

  What is `mmukmapsync` doing then? Try to guess, later I'll tell you.

- `trap.c:446`
  If bit 2 is set in the trap error code pushed by the processor, the fault was due to a write operation. So `read` means that it was a read the operation faulting at `addr`.

- `trap.c:447,449`
  `insyscall` records whether the faulting process was executing within the kernel (e.g. `sysexec`) or was running user code. In any case, you are now running within the kernel. `fault` is called to do the actual processing...

```
trap
  fault386
  fault()
```
Services a page fault.

- `../port/fault.c:9`
  ...and it is given the faulting address and an indication of whether it was a read the operation causing the fault or not.

- `fault.c:14,15`
  The routine saves the previous process 'ps' state (which would be restored later) and lets 'ps' know that the process is faulting. As you can see, the author uses `psstate` to be more descriptive regarding the process state; the process scheduling state is a different thing.

- `fault.c:16`
  Until now, interrupts were disabled—which also prevented context switches to a different process. Now that the page fault is being handled (and has saved `cr2`), interrupts can be allowed. Servicing a page fault may take a long time.

- `fault.c:18`
  Accounting for the local processor.

- `fault.c:19,38`
  `seg` locates the segment where `addr` stands, if there is no such segment, the address was outside segments used by the process, and the fault cannot be repaired (hence the return `-1`). If the fault was for write and the segment was a read only segment, the fault cannot be repaired either. Otherwise, `fixfault` would do its best to repair the fault—e.g. by allocating a new page frame, filling it with the contents of the faulting page, and repairing the address translation. As `fixfault` may fail due to allocation failures, etc., `fault` loops until the fault is either repaired, or is known not to be repairable. Finally, the saved 'ps' state is restored and `fault` returns 0 to indicate that it repaired the fault (because `fixfault` returned zero and the loop was broken).

- `../pc/trap.c:450,459`
  Before looking `fixfault` and `seg`, note that when `fault` returns, if it returns zero, the `insyscall` state is restored, and `fault386` returns to `trap`, which would return (or context switch to another process) causing a return from interrupt. The `iret` restores the processor context and the faulting instruction is retried. However, when `fault` returns `-1`, `fault386` would either cause a panic (if the faulting instruction was within the kernel) or post a "`sys:trap:fault`" note to the faulting process. That note can kill the faulting process. In our current example, the fault will be repaired.

`seg()` *Locates a segment given the address.*

- `../port/fault.c:359,380`
  First, `fault` calls `seg` with the pointer to the current `Proc`, the faulting address, and `dolock` set to true. `seg` iterates through the `seg` array of `up`, looking for a segment in use (they have a `Seg` hanging from `seg[]`) whose addresses contain `addr` (note `n->base` and `n->top`). If such segment is found, a pointer to the `Segment` is returned.

  If `dolock` was true, the `Segment` is locked and the check is repeated; to ensure that the segment was still there and its addresses still contain the faulting address.

```
trap...
   fault
       fixfault()
```
Tries to fix a repairable page fault.

- `fault.c:50,51`
  Should a segment contain the faulting address (`seg[ESEG]` in our case), `fixfault` is called for it. In this case, `doputmmu` is true—because `fault` wants the address translation to be ok for the hardware too.

- `fault.c:61,64`
  `va` is the faulting address; `addr` is set to the page address (by clearing the offset bits). `soff` is set to be the offset in `s` for the faulting address. `p` is a pointer to the `map` entry for the segment offset.

  Segment `maps` contain entries relative to the base address of the segment. The segment offset is used as an address to be translated by the `map`—very much like the hardware does with its page tables. `PTEMAPMEM` is the number of bytes addressed by each `map` entry (`../pc/mem.h:102,103` defines it as 1M, and defines `PTEPERTAB` as the number of pages needed to cover that Mbyte).

- `fault.c:65,66`
  One thing which can happen (that's the case for us), is that the segment does not even have a `map` entry allocated for the faulting offset. `ptealloc()` *Allocates a Pte.*

- `page.c:466,475`
  In this case, `ptealloc` allocates a `Pte` structure with `PTEPERTAB` entries to link `Pages` on it. This is like allocating the second level page table for the "virtual MMU" used to implement the segment. All entries in the `Pte` are still nil.

- `fault.c:68`
  `etp` is now a pointer to the `Pte` which should contain the `Page` structure for the faulting address.

- `fault.c:69`
  `pg` is set to point to the entry in `pages` where the `Page` for the faulting address should be. The index is the offset within a `map` for the segment offset, divided by the number of bytes in a page. In our case, `*pg` is nil as nobody allocated a page for the stack.

- `fault.c:70`
  `type` contains the kind of segment handled. More later.

- `fault.c:72,75`
  A `Pte`, contains `first` and `last` pointers that point to the first and last used entries. They are used to iterate through all used pages without having to iterate through all entries in the `Pte`—which could contain just a few contiguous pages. These lines update `first` and `last` accordingly.

- `fault.c:77,80`
  How to repair the fault, depends on the kind of segment; note the defensive programming once more.

- `fault.c:82,88`
  For page faults within text segments, the page should be paged in from the text file. `pio` does the job, as discussed later.

- `fault.c:90,105`
  For BSS segments, shared segments, stack segments, and segments mapped (from devices?), which is our case, the `pg` is checked. If it is nil, there was no page for the segment and a new one should be added. This is called "demand loading" or "zero-fill on demand" depending on whether the new page should be loaded from a file or should be just initialized to all-zero; the "on demand" part means that the system does it only when a page fault shows that the process demands the page involved.

```
trap...
   fixfault
       newpage()
```
Allocates a new page (frame) for a segment.

- `page.c:119,131`
  `newpage` is called to add a page to the segment at the given page address. More precisely, the segment had its page "officially", but that page had no page frame; `newpage` allocates a `Page` that represents a page frame and gives it to the segment virtual memory page.

  `Clear` was set to true to request the new page be cleared. By now, consider that there are more free pages than the value of `swapalloc.highwater` and the loop trying to allocate a page breaks at `:131`. Forget most of `newpage` now, which is discussed later, but note that by allocating a `Page`, the author allocates a page frame too (the one at `p->pa`).

- `page.c:185,197`
  The other thing of interest for us now is that a reference is added to the `Page` (it is being added to a segment), its `va` is set to the virtual address for the page, `modref` set to zero (the cache of the hardware bits in the page table) and the actual page frame for the page (at `VA(kmap(p))`) set to al zeroes. The page frame is allocated for the page, and the page is represented by the `Page` structure. More clear now?

```
trap...
   fixfault
       newpage()
```
Allocates a new page (frame) for a segment.

- `fault.c:100,101`
  Back to `fixfault`, after `new` is our new `Page`, it could be that `newpage` did set `s` to zero because it had problems to allocate a new page for the segment. In this case, the fault cannot be repaired and the routine returns `-1`. Despite `fixfault` saying that the fault is not yet repaired, `fault` retries later–hopefully, a `Page` for the segment could be allocated in the future.

- `fault.c:103`
  The entry in the segment `Pte` for the page (`pg`) is set to point to the new page. The segment has a new page which has a page frame for it. The `goto` does not need to be there because the `case` would fall through the next `case`. Probably in a previous version there was a `goto common` at some other point in the routine.

- `fault.c:107,108`
  For our stack segment, a page is allocated on demand if no page was there (the stack is growing), and the same happens for BSS segments (which are all zero, so zero filled pages can be allocated on demand). For a page fault on a data segment, processing would start here.

  `pagedout()` *Is the page paged out?*

- `fault.c:110,111`
  `pagedout` (`portdat.h:350`) returns true if the segment actually has the page,

but the page is not really in memory because either it was paged out (its page frame reclaimed for other uses) or it was never paged in (never read from whatever file it comes from). `onswap` (`portdat.h:349`) checks whether the `PG_ONSWAP` bit is set in the pointer to the `Page`; which is the convention for pages paged out. Thus, if the page was never paged in, `pagedout` notices that the pointer is nil and says that it was paged out (it lies). If the page was actually paged out, its page exists, but the pointer has the `PG_ONSWAP` bit set. In any case, the page has to be brought into a page frame, before it could be used. `pio` does the job of paging in the page.

In our current example, `pagedout` would return false.

- `fault.c:113,117`
  If the access was for read, `mmuphys` is set to be the contents of the page table entry (for the hardware MMU) for the faulting page. `PPN` returns the page frame (physical page) number for the entry, and bits for "read-only" and "valid" are set on it. Besides, the software copy of the "referenced" bit is set. I defer the discussion of `copymode` until later. Just note that if the page fault was because the page was missing, it is now in-memory, and the "valid" bit is set. The `switch` is broken because that is all that has to be done to repair the fault—but for updating the MMU page table entry.

- `fault.c:119,148`
  The number of references for the page is computed. For us `image` in the page is nil, so the number of references is just the `ref` field in the `Page`. In our case, there is just one reference to the page and code in `:127,140` does not execute. As there is no image for this page, only the `unlock` is done—no `duppage` called. All this will become more clear for you later. But let's concentrate on how are pages added to our stack segment.

- `fault.c:149,151`
  The fault was for a write access, so fill up the entry (`mmuphys`) for the hardware page table with the page frame number and the write and valid bits. The "modified" and "referenced" software bits are set in `modref`. `putmmu()` *Updates an MMU entry.*

- `fault.c:173,176`
  If the caller requested that the hardware MMU page table should be updated, `putmmu` updates the entry for `addr` with the the prototype in `mmuphys`. The `Page` structure is passed because on some architectures `putmmu` might need to use/update `Page` information, but that's not the case for the Intel. After `putmmu` returns, the hardware page table has a valid address translation from the faulting page to the just allocated page frame. `fixfault` returns zero to state that the fault was repaired, and `fault` would return to `386fault` which would return to `trap`.

  At last, the faulting processor context would be reloaded and resumed by the `iret` in `l.s`. In this case, the faulting instruction was one in `sysexec.c`, filling up the stack for the process; execution would continue from that point on.

The processing just described would also be the one when addresses within the BSS segment are first referenced. The BSS is a data segment initialized to all zeroes. By attaching pages to it as they are used, and initializing their page frames to all zeroes, the process can believe that the whole BSS segment was there right from the beginning. The same happens for stack segments, as you now know.

6.2.2 Text and data memory pages

```
trap...
    fixfault
```

- `fault.c:83,88`
  If the process first references a page within the text segment, these lines are reached. Processing is mostly like servicing a page fault for the stack segment, but there are important differences. Assuming that the page faulting was never brought from the text file to the text segment, `pagedout` would find `*pg` to be nil, and return true. `pio` is called to do page I/O on the text segment. After it loads the program text that should go in the page from the text file, `mmuphys` is updated with a translation to the page frame for reading (that means "execute" permission too). Later `putmmu` will install that translation in the MMU page table.

```
trap...
    fixfault
        pio()
```
Performs I/O to do a "page-in" for a page.

- `fault.c:180,191`
  `pio` tries to get into `*p` a `Page` (with the associated page frame) with its corresponding memory filled up according to what is said in `s`.

- `fault.c:192,199`
  If there is no `Page` pointer (which means that the `Pte` had a nil pointer for this page), the page contents must be brought in from the image attached to the segment. `daddr` is the address in disk for page contents. The address is `fstart` (the address in the image corresponding to the start of the segment) plus the offset within the segment for the faulting page. (Remember that there are different `fstart` values for text and data segments?). `lookpage` takes the `Image` attached and the address on it, and returns a cached `Page` for that image portion. Hopefully, the `Image` would be caching that page most of the times, and no access to disk would be required: `new` would be non-nil and `pio` is done. Should `new` be nil, you have to go to disk to read page contents. This is the `if` arm taken for the first reference to a text (or data) page.

- `fault.c:200,208`
  Should there be a pointer to a `Page`, that means that the page was paged out. The pointer (as you will see) is not really a pointer to a page, but a `daddr` with the `PG_ONSWAP` bit set. When low on memory, Plan 9 reclaims page frames from user pages. If a stack or a BSS page is reclaimed its page frame, page contents

must be stored somewhere else[2]; i.e. on the swap area. In this case, `swapaddr` returns the address in swap where the page copy stands, and `lookpage` uses the `swapimage Image` instead of the segment image. `putswap` marks the space allocated for the page within the swap area as no longer used.

Swap space is allocated just to keep the pages moved out from system memory. Unlike other systems (e.g. some UNIXes), Plan 9 does not keep the swap space allocated when the page is kept in memory. If the page ever needs to be paged out again, another piece of swap space would be allocated for it at that point in time.

- `fault.c:211,215`
  The page was not found in the `Image`. It is definitely not in memory. A new `Page` (with a fresh new page frame) is allocated. The page frame is mapped at `kaddr`.

- `fault.c:217,252`
  The page has been first referenced (see above).

- `fault.c:218,225`
  About to read from the channel to the file where `s->image` memory comes from. In case of error, release the page just allocated and call `faulterror`—which would either `pexit` or post a debug note. If page contents cannot be retrieved, there is no much else to do. The routine cannot return by calling `nexterror` because, in the end, the faulting context would be reloaded, another page fault happen, another I/O error for the channel happen, etc.

- `fault.c:227,231`
  The `read` procedure for the channel is called to read into `kaddr` (the page frame), `ask` bytes, starting at offset `daddr` (the address for the page in the file). Lines `:227,229` set `ask` to the page size or the number of bytes from the faulting address to the end of the segment—whatever is the minimum. The end of a segment does not need to be aligned at page boundaries. For example, a compiled file can have initialized variables (data segment contents) which could occupy just 1K bytes, much less than a page size; it would not make sense to read more than that Kbyte from the file.

- `fault.c:234,235`
  Remaining bytes in the page (3K in the example) would be set to zero. After these lines, the page is loaded in memory.

- `fault.c:239,251`
  While the page was being read into memory, the lock on `s->lk` was released—reads take a long time. That means that some other process could fault on the page too, and start to read it too. The first process reaching line `:239`, would notice that the `Pte` entry (`*p`) is still nil. So that process takes the responsibility of attaching the page to the segment: its `daddr` is set to the `daddr` computed, `cachepage` is called to let the `Image` keep the page cached, and the `Pte` entry is set to point to the page. The second process arriving here, would notice

[2] This also happens for other pages, as you will see

that `*p` is not nil, which means that the work is done, so it does nothing but to return (`cachectl` is not discussed here). The call to `putpage` releases the reference to `new`, which could cause the page to be deallocated when the number of references becomes zero.

The author prefers to let one process do some unuseful work some times (when faulting on a page being faulted by other too), than to keep the whole segment locked (which would block processes using that segment) just to avoid this (not so probable) race condition.

One more note, `pio` does not fill up any MMU page table entry. It just handles the virtual page table used by the segment, and does Page I/O. The caller should call `putmmu` to let the hardware know.

```
trap...
   fixfault
```

- `fault.c:110,111`
  For data pages first referenced, `pio` is called too, and processing is like above.

- `fault.c:144,145`
  However, for data segment pages first referenced (unlike stack pages), `duppage` is called when there is enough space in the swap area. What is happening is that data pages can be written. If the data page is written, its contents would differ from the disk file contents.

  Now that the page is still fresh (it is just read), the author prefers to employ a bit of time and memory to make a copy of the page. The copy is to be kept by the `Image`, so that when another process faults on this page, the initial contents do not need to be read from disk, but from the `Image` page cache instead.

6.2.3 Physical segments

- `fault.c:154,169`
  If the segment is a bunch of physical memory, servicing the page fault is done by allocating a `Page` structure for the physical page already assigned to the segment. The way to allocate the `Page` depends on whether the segment has a `pseg->pgalloc` function or not. If it has one, it is the provider of `Pages`, otherwise a `Page` is allocated and its `pa` is set to point to the `pseg->pa` address of the segment plus the offset for the faulting page in the segment. Physical segments are discussed together with `segattach`.

6.2.4 Hand made pages

```
main
   userinit
      segpage()   Adds a page to a segment eagerly; not on demand.
```

- `segment.c:222,243`
  `segpage` is used only during boot to add a page to a segment. `Page` is supposed to be initialized by the caller, and `segpage` only plugs the page in the appropriate `Pte` for the segment.

6.3 Page allocation and paging

You now know that segments are filled up with pages on demand. Let's see now in more detail how are page frames allocated when segments reclaim more memory.

6.3.1 Allocation and caching

`auxpage()` *Allocates a page frame.*

- `page.c:240,246`
  `auxpage` is called to allocate a `Page` structure, along with an associated page frame. It is used by the code in `cache.c` to allocate page frames for extents. Pages come from a free list of pages in `palloc`. They are taken from `head`.

- `page.c:247,250`
  `freecount` was initialized by `pageinit` to the number of free page frames. For each page frame, a `Page` structure was initialized (with its `pa` set to the page frame address) and linked into `palloc` list (`palloc.head`/`palloc.tail`). `swapalloc.highwater` was also initialized by `pageinit` to be 5/100 of the number of page frames available for users (not for kernel). So, if the number of free pages goes down a 5% of available memory for users, `auxpage` refuses to allocate one of the (now precious) free page (frames).

  When the author uses `auxpage`, allocation could fail. An example is `cchain` (`cache.c:383`), which does caching only if free pages are available. So, what is happening is that the kernel cache for remote files consumes only free pages, but refuses to grow when memory is scarce.

  Could the author use virtual pages to do caching? Yes, but in that case they could go to disk, and reading from a disk can (sometimes) be slower than reading from the network. Besides, in any case, a local file server can be used to cache remote files (e.g. `cfs`).

  `pageunchain()` *Removes a page from the `palloc` list.*

- `page.c:251`
  `pageunchain` (`page.c:65,80`) removes `p` from the `palloc` list and adjusts `freecount` (one less page). The list is double linked using the `next` and `prev` fields of `Page`—to remove any entry. Saw how the routine checks that the `palloc` lock is held?

- `page.c:253,261`
  The page should not be used (hence the `ref` check). A reference is added to the page, `uncachepage` called, and the page returned to the caller. The reason to call `uncachepage` is that the caller is going to use this page for new stuff. Let's see what this means.

Images are used to represent an image in memory (read: cache) of file contents. You should remember that `lookpage` is called with an `Image` and a disk address to recover a cached `Page` for that offset within the image. How can that be done?

`cachepage()` *Adds a page as a cache for part of an Image.*

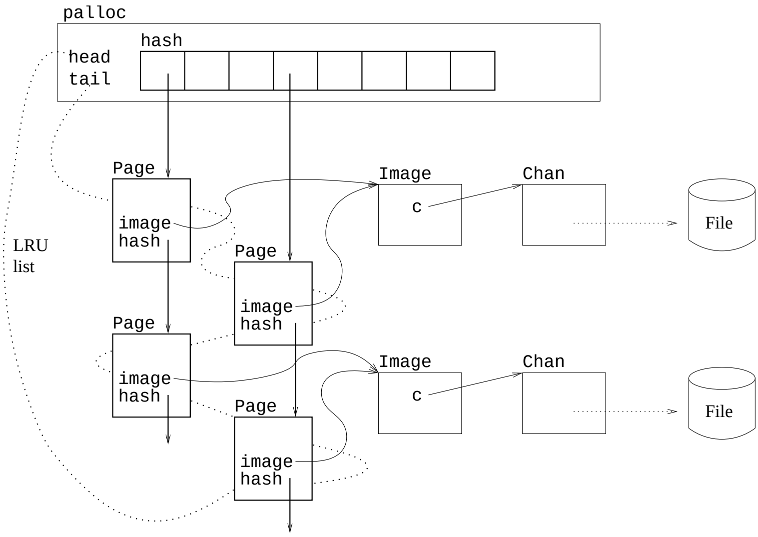


Figure 6.4: `Palloc` cooperates with `Images`, so that a cache of pages is kept for Images. Pages are hashed on their `daddrs` and maintained on a LRU list.

- `page.c:365,385`
  `cachepage` is called (as you know) when an image page is read from its file. It locks the `palloc.hash` table, and adds the page to the hash bucket for the page. `pghash` uses the `daddr` (ignoring page offset) to hash the page. Since the page is being kept for caching part of `i`, its `image` field is set to point to the image and a new reference added to the image. The whole picture is shown in figure 6.4.

`lookpage()` *Lookup a page in the cache.*

- `page.c:411,439`
  When later a routine wants a page from the image, `lookpage` scans the hash bucket (given the disk address) for a page with the same `Image` and the same `daddr`. If the page is found (note the double check to avoid locking all the pages) a reference is added to it (`:428`).

  When an image is released by the last segment using it, its pages are still kept in `palloc.hash`. If no one is using such pages, their reference count would be zero. However, such pages still contain file data which would save disk (or network) reads. Besides being kept in the hash, free pages are linked through the list starting at `palloc.head`. The call to `pageunchain` removes the page from its list (the `palloc.head` list). If this is the first new reference, the page must be removed from the free list, but second and posterior new references would find the page out of the free list (the page might be reused multiple times if found by `lookpage` for different segments).

`uncachepage()` *Removes a page from the cache.*

- `page.c:342,363`
  Finally, pages leave the cache (are removed from the hash) when `uncachepage` is called (e.g. after `auxpage` allocates an unused page for a different purpose). At this point, the page is no longer caching part of the file image in memory.

To summarize, pages are initialized to represent fresh page frames during boot. They are allocated later from the free list (initialized at boot) and added to the hash for use as a cache. When their last reference goes away, they are added again to the free list. Only when they are used for a different purpose, they are removed from the cache—in the hope that the same file will be used again (e.g. `ls` would be executed once more). As long as there are free page frames, nothing else happens. But what happens when physical memory is not enough for segment pages and to cache file contents?

`newpage()` *Allocates a page.*

- `page.c:118,119`
  `newpage` is called to allocate a new page for the given segment.

- `page.c:130,133`
  For user processes, less than a 5% of free user pages is considered as "no more free pages"; kernel processes would still get their pages until really out of memory. During this loop, the caller process would be waiting to get a free page for its usage.

- `page.c:135`
  While waiting, release the `palloc` lock. This allows other processes to use it (e.g. to release pages).

- `page.c:136,141`
  If the caller supplied a pointer to its `Segment*`, which means that it was allocating a page for that segment, the author releases the segment lock, clears the `Segment*` used by the caller and sets `dontalloc`. Later, after trying to get more free pages, the routine would return (`:154,161`) without trying to allocate a page. Can you guess why the author does this?

- `page.c:142,152`
  In this critical region, the process calls `kickpager` and sleeps for a while waiting for free pages. Hopefully the pager process notified by `kickpager` would make more pages available, by stealing pages from someone else. If there are several processes allocating free pages, they will enter this region too, until the point when one of the checks at `:130,133` succeed.

- `page.c:154,161`
  When memory is considered to be full, the process would loop, waking up the pager and sleeping for a while repeated times, until the pager gets free pages. This usually requires disk or network I/O and can be a really slow process. Now, if the caller is servicing a page fault on a segment, it would make no sense to keep the whole segment locked just to await for a free page. The segment could be the text of `ls` and many other `ls` processes could perhaps keep on running

without the faulting page. What the author does is to let `freepage` release the lock of the segment (`:138`), and tell the caller (`:139`) that the segment is no longer locked. For example, in `fault.c:99,101`, `fixfault` would notice that it had to wait, and did loose the segment lock, for slow I/O. It would fail and let `fault` retry later.

- `page.c:166,169`
  Got some free pages (according to `freecount`). Get one from the free list. Due to processor caching issues, not all pages are the same. The algorithm assigns "colors" to pages, so that it's best to allocate a "red" page frame for a "red" page (`:127`). Only a few colors are needed (like colors in a map).

  To remind you, this is because the processor (hardware) cache sometimes use the same cache entry depending on the (physical, or virtual, depending on the architecture) address of a page. For example, a processor with two cache entries (ridiculous, but just an example) could use entry 0 for even page frames, and entry 1 for odd page frames. If a process is using a data structure between two pages, it is better to allocate even page frames to even pages, and odd page frames to odd pages. This way, the two pages can be kept in the processor cache at the same time. I hope you get the picture with this silly example, but note that for the Intel, `getpgcolor` always gives 0 as the color (`../pc/mem.h:120`). So the author does no page coloring for the PC.

- `page.c:171,176`
  No free page for our color, just take the first one. `ct` is used to control the cache. By now, note that `PG_NOFLUSH` or `PG_NEWCOL` is set depending on whether a page of the right color was found or not.

- `page.c:178`
  The page removed from the free list.

- `page.c:180,187`
  The author checks that the page was indeed free (no references), and removes it from the cache. This page could belong to a different image and be placed on the free list by the pager. In any case, the page is going to be used for a different purpose now.

- `page.c:188,189`
  The page has a `cachectl` array with an entry per processor. For all processors, entries are set to either `PG_NOFLUSH` or `PG_NEWCOL` depending on the page color. On Intels, `cachectl` is not used, but other architectures might use it to determine whether the cache entry for the page should be flushed or not. This is because there are architectures that fill up the cache using virtual addresses, if a page has changed its virtual address (the frame is reused for a different page), its entry on the cache might need to be flushed if the hardware wouldn't notice that and flush the entry automatically.

- `page.c:193,197`
  Zero the memory if requested.

6.3.2 Paging out

`kickpager()` *Starts the pager process or wakes it up.*

- `swap.c:90,101`
  `kickpager` is used to ask the pager to do its job. Its purpose is to get some free pages. Because of the static `started`, the first time the system is running out of (physical) memory, a kernel process is started to run the `pager` function[3]. Next times, only a `wakeup` for `swapalloc.r` is issued. Let's see `pager`.

`pager()` *Main routine for the pager process.*

- `swap.c:103,118`
  `pager` keeps running under the `loop` to get more free pages. To prevent it from running even when there is free memory, the author makes it sleep at line `:118`. It will stay there until awakened by a process calling `newpage`. When the first call to `kickpager` starts the pager, `needpages` can prevent the pager from sleeping.

- `swap.c:120`
  The pager will not sleep until it has managed to free some pages.

- `swap.c:122`
  `swapimage.c` is the channel to the swap file (or partition). If there is such channel, some pages can be paged out to swap space (i.e. copied to the swap file and their frames reused as free memory).

- `swap.c:123,125`
  `p` is going in a round-robin fashion among existing processes. All configured process entries are scanned (`:133,114`).

- `swap.c:127,128`
  Dead processes are not using memory and are skipped (note that dead is not broken), and kernel processes are given kept untouched (they run within the kernel, don't they?)

- `swap.c:130,131`
  If `canqlock` acquires the look, it is the turn for this process to donate some of its pages to the Plan 9 cause. Otherwise, the process is touching its segments. Instead of blocking the pager process, the author chooses another victim. Although the algorithm is not fair, it is better to use an unfair algorithm than it is to let the pager block while free memory is needed.

- `swap.c:133,137`
  Starting to iterate over the process segments, seeking for pages to steal. As soon as `needpages` says so, no more pages are stolen. The segment lock is kept for the whole search.

[3]The console device may start also the pager.

- `swap.c:139,160`
  Got a segment for the process (an used entry). Only segment types mentioned in the `switch` get pages stolen. `pageout` is the page thief. When it is a text page the one stolen, the 'ps' state is not changed (because the page comes from the text file (read-only) and there is no need to maintain the process locked in `pageout` for too long. For data pages (including stack and others), the 'ps' state is set to `Pageout` during the call to `pageout` and maybe later to `I/O`, during the call to `executeio`. Let's defer a bit `pageout` and `executeio`. As you just saw, all (user) processes are candidates to get their pages stolen in a round-robin manner.

- `swap.c:164,174`
  If there is no swap file defined yet, `freebroken` is called on terminals to kill broken processes and reuse their memory. On CPU servers, `killbig` is used instead. A message is printed to let the user know that either more memory should be added to the system, or a swap file configured. The call to `tsleep` prevents the message from appearing too frequently.

`pager`
`killbig()` *Kills a big process and reclaims its memory.*

- `proc.c:1156,1190`
  `killbig` locates the process with the biggest memory image (sum of the lengths for its segments), and kills it. The author knows this is not fair, and prints a diagnostic to let the user know. However, the author hopes that by killing this big process, the CPU server could get out of the out of memory condition. Nevertheless, the injured user could ask the CPU server administrator to configure a swap file if that's the problem. The action really killing the process is a `procctl` order of `exitbig`, so the process will kill itself later when it checks its `procctl`. But in any case, as the memory is needed now, `mfreeseg` is called to release the memory of all user segments in that process, so that it be available now; `mfreeseg` is discussed later.

`pager`
`pageout()` *Steals pages from a process segment.*

- `swap.c:179,180`
  `pageout` tries to steal pages from the given process and segment.

- `swap.c:186,187`
  When the author services page faults, `lk` is acquired and `newpage` can be called. Now, `newpage` can awake the pager which can try to get the lock to do some page outs—skipping locked segments, (note that this actually means `Pte` map locking, and not `seg` locking).

  By using `canqlock`, the worst thing that may happen is that other segment is used to steal pages from, not a big deal when compared with a deadlock. You should note that the pager does not want to block as it should get free memory soon. Besides, by avoiding races against page operations on the locked segments, the author can forget about race conditions in that respect.

- `swap.c:189,192`
  `steal` is non-zero while `procctlmemio` (`devproc.c:981,1049`) is doing I/O on segment memory. That is precisely to prevent the pager from paging out segment pages under `devproc` feet.

- `swap.c:194,198`
  Only if `canflush`, are pages stolen from this segment. As `canflush` adds an extra reference to the segment, `putseg` must be called to release the reference. The reference avoids segment deletion while it is being used to page out some of its pages.

```
pager
    pageout
        canflush()
```
Anyone running on pages from this segment?

- `swap.c:235,265`
  `canflush` returns true if `canpage` returns true for all (alive) processes using the given segment. When there is more than one reference, all processes must be searched to find other users of the segment.

- `proc.c:283,299`
  `canpage` returns true only if the process is not running, and it sets `newtlb` to true in such case.

  What is going on? If the process is running (pager is just another process, and there can be multiple processors), the author refuses to check if the page is really being used or not. It is more simple to remove pages from processes not running (note the `mach` check!). When the process runs again, it will notice the `newtlb` flag and flush its MMU (because page translations are going to change in `pageout`). So, only when none of the processes using the page is running, can `pageout` steal the page.

  What if a process which said it `canpage` runs after the call to `canpage` but before `pageout` completes? That is no problem, `mmuswitch` would notice `newtlb` and will call `mmuptefree` to set as invalid the entries in the MMU page table for every user page in that process. So, `pageout` really hurts to the process affected. Even if it gets just a few pages paged out, it will suffer many page faults.

```
pager
    pageout
```

- `swap.c:207,228`
  For all (user) `Pages` in the segment (`first` and `last` are used to avoid scanning all the `map` entries), if the `Page` is `pagedout` (never paged in, or paged out) it is ignored. If the page has the referenced bit set, it is forgiven and it has a new chance to be referenced again before the next pass of the pager for this page. If the page is not referenced (or was forgiven and not referenced again before being reached once more), `pagepte` steals the page. This is a second-chance paging out policy. If the author did not forgave pages referenced, pages really in use by the process could be paged-in soon, and then paged out again, and the system could end up trashing (it would do nothing else but to service page-ins and page-outs).

- `swap.c:225,226`
  `ioptr` points to the current page I/O transaction. If `nswppo` page I/O requests are in place, the system refuses to do more page I/O. I think that the author tries to avoid trashing here too. If after finishing the current transactions, memory is still scarce, more page outs would happen. Another good reason not to do too much I/O to get free memory is that the user is waiting for his application to run, and the application is waiting because the processor is being used for the pager too.

```
pager
  pageout
     pagepte()
```
Updates a `Pte` for a page out, adding the page to the I/O list.

- `swap.c:267,274`
  Paging out a page. A pointer to the pointer used by the caller is passed, so that `pagepte` can update it for the caller.

- `swap.c:275,278`
  If the page is a text page, it can be paged-in later from the text file. The caller `Page*` is cleared and the reference to the page released. That's all to do here. This is the reason why `psstate` was not changed for the process while doing a page out. Because, in fact, there are no "page outs" for text pages, they are simply discarded.

- `swap.c:280,290`
  For these segments, a copy of the page memory must be made before reusing its frame. `newswap` allocates a new disk address (within the swap file) where to copy the page.

  `newswap()` *Allocates space for a page in swap.*

- `swap.c:35,56`
  `newswap` scans a bitmap `swapalloc` for a zero byte—which represents room for a page in the corresponding offset for the swap file. `last` and `top` are used to do kind of a next-fit policy for swap space allocation; `last` is kept set as the last slot found at `look`. Initially, `last` is initialized to point to the start of the "byte-map" in `swapinit`. Line `:51` is marking the byte as allocated. The author trades space for time, by using bytes and not bits in the map. It is more simple to use a whole byte than it is to use a bit (it should be masked and checked). Memory is cheap these days.

  By the way, now you can understand why `fault.c:122,123` called `swapcount` to account extra references for `Pages` that had `swapimage` as their image. `swapcount` (`swap.c:84,88`) returns the "1" or the "0" in the swap "bytemap" entry for the page. So, code in `fault.c:122,123` accounts one extra reference for a swap file `Page` if the swap bitmap states that such page is being used. As you will see soon, when (non-text) pages are paged out, their `ref` could reach zero.

- `swap.c:291,292`
  `newswap` returns -1 (in two's complement, note the unsigned return value from

`newswap`) when there are no more free swap slots for pages. The routine refuses to page out if there is no place to copy page contents.

- `swap.c:293`
  `cachedel` removes any cached page for the swap image at disk address `daddr`. Remember that `lookpage` can try to get a cached page for a disk address? In the past, this disk address could haven been used to keep other page, and it could be that the `palloc` hash (cache) still has a cached pages pretending to be the contents for this file slot. No longer the case. The pages caching `daddr` would still be in the free list, but it would not be in the hash list any more.

- `swap.c:295,298`
  In the same way, the page being paged out can be linked into the `palloc` hash, as a cache for part of its image. `uncachepage` removes the page from the hash, and sets its `image` and `daddr` to zero. Now the page is no longer for its old image—`uncachepage` expects the page to be locked.

- `swap.c:300,317`
  The comments say it all. But note that `PG_ONSWAP` is set (see `pagedout`). Although the page is not yet sent to the swap file, the kernel considers it as swapped out. The space for the pointer to the `Page` in the segment is used to keep the `daddr` for the page in the swap file.

  Now `pagepte` completes and `pageout` would perhaps loop adding more I/O requests to `iolist` by calling `pagepte` more times. When the configured I/O limit is reached, or when the segment is entirely scanned, `pageout` completes too. Later, lines `:154,156` would call `executeio`.

`pager`
  `executeio()` *Performs I/O for page outs.*

- `swap.c:331,366`
  All I/O requests placed in `iolist` are serviced at a go. For each page set in `iolist`, `kmap` is called to set a temporary mapping so that `write` can be called for the swap file channel to write the page contents. The segment reference to the page is not released (`putpage` called) until write returns; i.e. after page contents are safe in swap. Besides the segment (`Pte` entry) reference, there was another extra reference which is removed at `:361`. As the process had the segment unlocked before `executeio` executes, the segment could call `putpage` on its own.

  If you look at `putpage`, (`page.c:209,238`), you will see that it checks the `PG_ONSWAP` bit and calls `putswap` to release pages with the bit set. However, `pagepte` did set the bit in the `Segment` pointer to the page, but not in the `iolist` pointer to the page. Therefore, the `putpage` call from `executeio` really does a `putpage`.

  By deferring page I/O requests until after the segment is unlocked, the time the segment lock is held is reduced a lot (I/O takes a long time). In the mean time, processes could service other page faults for the segment. This is very important since `mmuptefree` clears page table entries for the process affected and it will have to service many page faults for the segment.

As a final comment, note that the `PG_MOD` bit is not checked to avoid calling `write` for a page which has not been modified since it was last read (e.g. from the data section of the text file). Although that could save some I/O, the author probably thinks that it is not worth to do so. Take into account that he duplicate made for data pages before they could be written helps here.

Should the author change his mind in this respect, pages backed up by swap space would need to keep swap space allocated even while in memory, so that pages not changed since their last page-in could be discarded without I/O.

6.3.3 Configuring a swap file

```
syswrite
   conswrite
```

- `devcons.c:847,864`
  A swap file is configured by a write to `#c/swap`. Usually, the string written is the number of an open file descriptor for the file to be used as a swap area. A write of the string `start` would `kickpager` instead of configuring the swap file. For CPU servers, only the boot user (Eve) can configure a swap file—otherwise, any user could read memory from other user's processes by configuring a swap file and forcing the server into a low-memory condition. Since terminals run processes on the name of eve, the author does not check anything for them. The work is done by `setswapchan`.

```
syswrite
   conswrite
      setswapchan()
```
Configures a swap channel.

- `swap.c:374,403`
  The important work is done at line `:402`. Lines up to `:386` take care of unconfiguring a previous swap file if it was not used (All `nswap` pages are `free`); the new file will be used instead. Lines `:392,400` limit the number of pages in the swap file to be those that fit in the partition. Surprisingly, if a previous swap file existed, `nswap` would only decrease, and not increase. It will always be at or below the value configured at boot time. By the way, the check for '`M`' is because all 'files' come from the mount driver—this is a CPU/terminal kernel; so, if the device is not `#M`, it is likely to be a kernel device who provided file, and that's usually a disk partition. Perhaps a more explicit check against the storage device could be made—but what about using a `kfs` file? Hint: what if `kfs` data was swapped out?

6.3.4 Paging in

You already saw most of the code needed to page-in some pages for user segments, while learning how are pages added to segments: Plan 9 uses demand load for pages. Let's see now the part of the code we didn't read.

To remind you, `fixfault` called `pio` to do a page-in for the faulting page. You saw how the cache for the image (or the swap file image) was tried by `pio`, and how `pio` worked when there was not a `Page` for the page.

```
trap...
   fixfault
      pio
```

- `fault.c:253,269`
  Now, when it appears to be a pointer to a `Page` hanging from `loadrec` (from the segment map), the page faulting was paged out. Moreover, the page faulting is not a text page because text pages have their `Pages` deallocated on page-outs. So, the page must be read from the swap file pointed to by (`swapimage.c`). Remember that at this point in `pio`, `new` has a fresh page (frame) to be used for the faulting page.

  The calls to `qlock`/`qunlock` within the error recovery block at lines `:258,259` seem to be to wait until sure that no other process is holding the `lk` lock. This routine acquires that lock. `faulterror` can call `pexit` to kill the process. Therefore, if the I/O fails to do a page-in, the process is killed after nobody is running within the critical region.

- `fault.c:271,286`
  During the call to read (which can block), the segment was unlocked. Another process could initiate a page in. This case is easy to check because it that did not happen, `loadrec` would still be `*p`, the entry in the `Segment`. If `loadrec` differs, that must be because at line `:290` the other process did set the entry to the page it allocated. The first one getting past line `:269` wins. If another process did the page-in, the routine releases the page allocated (and read!!) by this process: all done. It could also be that the pager stole this page, which is known because `pagedout` returns true (and `*p` changed too!). In this case the page has to be paged in again, hence the `goto`.

  Perhaps the author could save some reads by allowing at most one process to do the read for a page-in. Nevertheless, it is not clear that would be worth because several processes must be faulting on the same page for it to be worth. Besides, the saved time would be minimized because of caching. Remember? Measure and then optimize, not vice-versa.

6.3.5 Weird paging code?

If you are curious, you already noticed how there are still portions of paging code that remain to be read. In particular,

```
trap...
   fixfault
```

- `fault.c:113,117`
  This code restores the hardware MMU permissions for the faulting page when it is a read-fault and `copymode` is zero and avoids doing any other thing.

- `fault.c:127,140`
  This code does something which is not calling `duppage` when there is more than one reference to the `Page` (including as references pages swapped out). If you look at the code, it allocates a new page and calls `copypage` to install a translation to a copy of the faulting page.

To understand what is going on, you must read `dupseg` first. That procedure clones a segment during a `rfork`.

6.4 Duplicating segments

`sysrfork()`
  `dupseg()` *Duplicates or shares a segment.*

- `segment.c:143,144`
  During `rfork` (`sysproc.c:107`), `dupseg` is called to duplicate a segment or request that it be shared with the parent process. `seg` is the process segment array. The routine is expected to return a pointer to the duplicated/shared segment.

- `segment.c:153`
  Segments are duplicated one way or another depending on the kind of segment.

- `segment.c:154,159`
  For segments that are read-only (text), shared (`SG_SHARED` and `SG_SHDATA`), or physical memory (which is shared), the reference count is incremented and the parent's segment is used as-is by the child. It does not matter what `share` says.

- `segment.c:161,169`
  Stack segments are never shared. `newseg` creates a new stack segment with the same base and size as the original segment. This new segment is still empty (although it should be a copy of the parent's stack).

- `segment.c:212,219`
  Later, `dupseg` calls `ptecpy` to copy each `Pte` in the original segment to the new (duplicate) segment. Let's see what happens to other segments before looking at `ptecpy`.

- `segment.c:171,186`
  For BSS segments (and MAPs), one of two things can happen.

  If the segment is to be shared, and nobody is sharing the segment (its reference is one), the segment type is changed to be `SG_SHARED`. From now on, calls to `dupseg` for this segment would just add another reference, as seen before. You now know that a `SG_SHARED` segment is a BSS or MAP segment that is being shared. The routine adds the extra reference and returns the segment as the duplicate one.

  If the segment is not to be shared (`!share`) a new segment is created, and later, `ptecpy` would copy `Pte` entries for the new segment. Let's see now `ptecpy`.

```
sysrfork()
   dupseg
      ptecopy()
```
Copies PTE entries.

- `page.c:442,464`
  `ptecpy` is an innocent looking function, which allocates new `Ptes` for the duplicated segment and iterates through all (used) `Pages` in the old `Pte`. For each entry used, if the page was swapped out, `dupswap` is called. But otherwise, an extra reference is just added to the `Page`, and the the `Page*` in the destination entry is copied from the source entry! Noticed that the segment is being duplicated, although `Pages` are shared? Although the duplicate segment was expected to get a copy of the pages, it gets the *same* pages. What's going on?

  The duplicated segment has no real MMU page table entry updated, so it is going to page fault on the pages "copied", although its entries are set in its `Ptes`.

```
trap...
   fixfault
```

- `fault.c:126,141`
  When the process later has a page fault due to a write on the "copied" BSS segment, the page has more than one reference (for read accesses, the MMU entry is given read permission and nothing else is done).

  In this case, `fixfault` knows that the extra references are there because although the page is being shared, it should not be shared officially (pages really shared on shared segments have a reference count of 1, it is the `Segment` that has the reference count bigger than 1). This is called "copy on write", or COW. When the segment is duplicated, the segment and its `Ptes` are duplicated, but the pages are shared (with a reference count bigger than 1). On a (write) page fault, the page is copied (`copypage`) to a new page frame (`new`), and the new page is given to the faulting process. there is a call to `putpage` because this process is no longer using the shared copy of the page. In figure 6.5 you can see a COW segment and a shared segment.

  If other processes copied the segment (on write), the number of references in the page after `putpage` is still greater than one, and more copies will be made, if the page is referenced. If no other process is sharing (copying on write) the page, its reference is one, and lines `:143,150` would execute instead: the page is used as is, after saving a copy for the image cache and restoring permissions in the MMU entry.

  The check for `copymode` at line `:113` is deciding when to really copy the page. I have been saying that the page is copied on *write*. I lied. Only when `copymode` is not-zero is the kernel restoring MMU permissions (without any copying) for read faults. When it is zero, even a page fault for reading causes the page to be copied. So, if `conf.copymode` is zero, Plan 9 does *copy on reference*, otherwise, *copy on write* is used.

  `copymode` is zero unless `archinit` or `mpinit` set it to one, which happens only for multiprocessor machines. So, Plan 9 uses copy on reference for monoprocessors and copy on write for (Intel) multiprocessors.

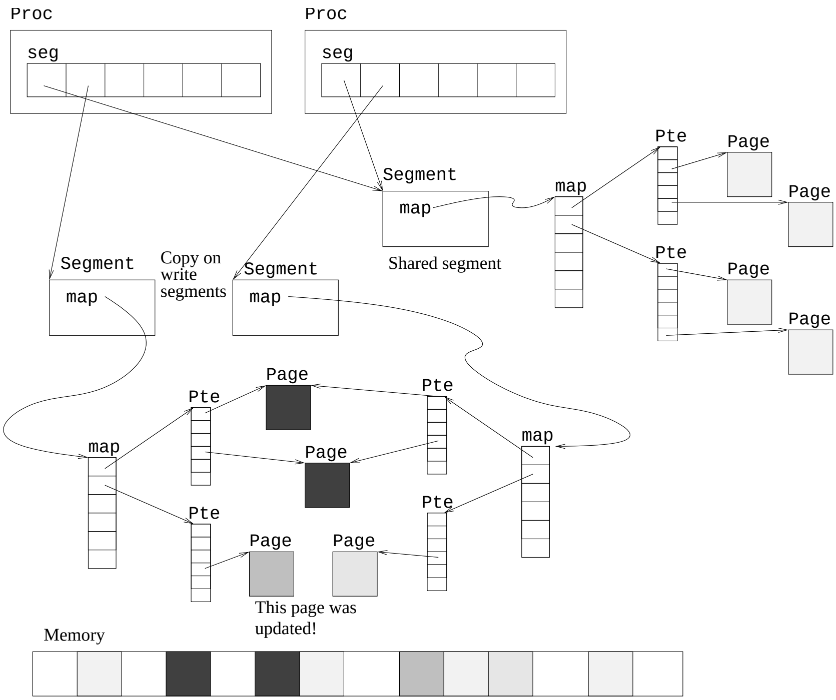


Figure 6.5: A copy on write (or copy on reference) segment is like a copy of another segment: but both segments share unmodified (unreferenced) pages. This is not to be confused with a shared segment. The figure shows how each `Page` has an associated page frame.

It is clear that the copied segment has all its MMU translations invalid; but what about the original segment copied on write? Any write to its pages should also cause a page fault, but since the segment was read/write, translations would still have write access in the MMU.

- `sysproc.c:182,187`
  Is it clear the comment now? All translations are set invalid by `flushmmu`. Page faults for read would just repair the MMU translations. Page faults for write would copy the COW pages. Although it would be faster to downgrade the translations to read-only for COWed segments, the author prefers to keep the code simple. Should this be a performance problem, perhaps `flushmmu` would be replaced by a more clever routine which could downgrade entries too.

```
sysrfork()
  dupseg
```

- `segment.c:188,211`
  Ignore the `data2txt` thing. If the segment is to be shared, add a reference to it and change its type to `SG_SHDATA`; you now know what's a `SG_SHDATA` segment. If the segment is not being shared, but copied, COW is used and a new segment is created as before. The difference with respect to COW for BSS segments is that the image is added an extra reference and attached to the segment too. BSS segments have their storage initialized to zero, and they are never paged in from any image (well, they are paged in from swap, but you know how that is done). On the other hand, data segments page their pages in from the image corresponding to the text file with initial contents for the data segment. Even though the segment is COW, the copied segment should also page in from the image those pages which have never been referenced. So, the segment must be attached to the image too. Besides this reason, it is good design to keep the copied segment attached to the same image, as it would be if it was the original segment: it is a copy, isn't it?

- `segment.c:189,190`
  Back to the first two lines, you must read a bit of `devproc` to understand what is going on.

`procctlmemio()` *Does I/O to memory of a process.*

- `devproc.c:998,999`
  If `procctlmemio` is servicing a write for the text segment, it calls `txt2data`.

`txt2data()` *Replaces a text segment with a data segment.*

- `devproc.c:1051,1078`
  `txt2data` takes a text segment and replaces it by a data segment. The data segment is a regular `SG_DATA` segment but, if the text segment was the one in `seg[TSEG]`, the entry in `TSEG` now contains an `SG_DATA`, which is not usual. As a data segment, it accepts writes, and that seems to be the reason why the author is replacing an `SG_TEXT` with an `SG_DATA`. Although the author could have set a `mode` field in `Segment`, and provide some means to change it, it is more clear to have the type of the segment determine what can be done to the

segment. As this kind of write seems to be most useful for debugging, it is not likely to be a frequent operation and its efficiency is not an issue.

It is useful to be able to write to the text segment to change instructions, and to set breakpoints on it.

- `segment.c:189,190`
  However, if the process forks, the child should get its regular `SG_TEXT` segment in `TSEG`. `dupseg` knows and calls `data2txt` to recover the text segment corresponding to the weird data segment.

  `data2txt()` *Restores a text segment from a replacement data segment.*

- `devproc.c:1080,1093`
  The only thing `data2txt` does is to recreate the text segment from the image, as happened before during `exec`.

By the way, the checks for `ref==1` besides checking `share` during `dupseg` seem to be more of defensive programming; since the segment type would be changed just the very first time. But I may be missing something here.

6.5 Terminating segments

`putseg()` *Drops a reference to a segment.*

- `segment.c:83`
  When a segment is being released, `putseg` drops the reference to it.

- `segment.c:91,100`
  Segments and images are linked circularly, the convention is to lock the image before the segment. If the last reference to the segment is being released, it will go away and the image should no longer point to the segment. In this case, `i->s` is cleared.

- `segment.c:108,123`
  The last reference is going. The routine clears this thing up, including a call to `putimage`, if there is an image attached. `putimage` does not call to `putseg` to release its reference to the segment. You know why, don't you?

`putseg`

`putimage()` *Drops a reference to an image.*

- `segment.c:385,391`
  `putimage` releases the image. It does nothing for images with `notext`. You already saw in the starting up chapter that `notext` was set for the first process text image; it is also set for the swap image by `swapinit` and for the `fscache` image used to cache remote files in `cache.c`.

- `segment.c:394,426`
  For images with "text", after their last reference is gone, their QID is cleared, and their channel to the text is closed. The Image is also removed from the

Image hash table and linked to the free list (`attachimage` would no longer find this image when searching for other users of the same text). One thing to note here is how the channel is not really closed here, but placed into `freechan` (which is resized on demand) to be closed later. Closing a channel may block and also may take a long time. The author wants that to happen after all locks have been released.

`attachimage`
  `imagechanreclaim()` *Releases channels used for images (as well as images).*

- `segment.c:358,382`
  `imagechanreclaim` is the routine actually calling `close` for these channels. It is called from `attachimage`. Read the comment regarding locks, it is very explicative. Although keeping these channels open require file servers to keep resources that are not really useful (channels are about to be closed), the author prefers to close the channels on calls to `attachimage` rather than after calling `unlock` in `putimage`. One fine reason can be that putimage is also called during page faults (which already may take a long time). Perhaps the pager could be also in charge of closing image channels.

  By the way, you now know why `imagereclaim` puts pages instead of images to get some `Image` structures released, because pages keep the cached contents of the image and the image will not go away until all its pages have been released.

`putseg`

- `segment.c:112,115`
  Back to `putseg`, `freepte` releases the `Pages` used by the segment.

`putseg`
  `freepte()` *Releases pages in a Pte.*

- `page.c:478,516`
  For physical segments, a `pgfree` function is called to deallocate pages (in the same way that the `pgalloc` routine was used by `fixfault`). Besides calling `pgfree`, the `Page` reference count is decremented and it is freed if no more references. `putpage` is not called, because the physical segment allocates and deallocates pages in a rather specific way.

- `page.c:508,514`
  The usual thing. `putpage` is called for all `Pages` found in the segment.

`putseg`
  `freepte`
    `putpage()` *Releases a page.*

- `page.c:208,238`
  if the page was swapped out, `p` is not a pointer to the page, but a swap address.

  `putswap()` *Releases a swapped out page.* `putswap` (`swap.c:58,73`) marks the swap page as no longer allocated in the swap "bytemap".

Otherwise, `p` can point to a real `Page`, and reference counting is used. If the image for the page is not `swapimage`, the page is linked to the tail of the free list, otherwise it is linked to the head. Since pages are allocated from the head, the author is trying to keep the page in the list for a longer time if the page still caches part of a image. Since the swap image is just used as backing storage, their pages are to be reused soon. It should be clear now, but note how the page free list is actually used as a cache.

Finally, the author wakes up a process that was sleeping waiting for pages—if any. The reason to wake up a sleeping process is that there is now a free page available for allocation (Remaining requesters could still be blocked in `palloc.pwait`, only one of them did pass and sleep in `palloc.r`). The reason not to issue the `wakeup` when no process is sleeping is that in that case, nobody is waiting for memory. Nevertheless, `wakeup` would no nothing in that case and calling it wouldn't hurt, or did the author measure that it would hurt performance? Or am I missing something here?

Segments are usually released by `putseg`, however, there is another routine (the one called by `killbig`, which you already saw) that is called to release memory held by a segment. It is also used by a couple other routines besides `killbig`.

`mfreeseg()` *Releases the memory used by a segment.*

- `segment.c:503,536`
  First, `mfreeseg` scans all entries in the segment that are for the `pages` starting at `start`. Unused `map` entries are ignored (they have no memory to free); all entries in the segment Ptes are set to nil and pages kept linked at `list`.

- `segment.c:537,551`
  Second, `mfreeseg` calls `putpage` for all pages linked (all segment pages). Before doing so, `procflushseg` is called if the segment is shared. The reason is that when the segment is shared, page reference counts are one, but pages should not go away before letting all processes using the segment know that the segment memory is being released. Other processes using the segment could be even running right now at a different processor.

`mfreeseg`
`procflushseg()` *Flushes MMU for processes using the given segment.*

- `proc.c:973,998`
  `procflushseg` iterates through all processes, searching for `seg` entries with the `s` segment. `newtlb` is set for all processes with such a segment, so that its entries are invalidated before it runs again. Besides, (`:991`) if the process is running at any processor, `flushmmu` is set for that processor and `nwait` incremented.

- `proc.c:1007,1001`
  If `nwait` was not zero, the current process waits until `flushmmu` is reset to zero for all processors. The current processor can only 'kindly request' to other processors that their MMU entries be flushed, they are running and they will flush such entries when they notice the `flushmmu` flag. Hopefully, that will happen soon.

6.6 Segment system calls

You now know how typical text, data, bss and stack segments are created, copied during `rfork` and how are page faults serviced. That is most of what you should know about memory management. However, there are several system calls related to memory management that remain yet to be seen.

6.6.1 Attaching segments

`syssegattach()` *Entry point for the* `segattach` *system call. Attaches a new segment.*

- `sysproc.c:614,618`
  `syssegattach` is used to create a new memory segment. System call arguments are passed verbatim to `segattach`.

`syssegattach`
`segattach()` *Attaches a new segment.*

- `segment.c:600,601`
  It is `segattach` who does the job.

- `segment.c:607,608`
  `va` is given as zero when `segattach` should choose the address where to map the segment in the process address space. If it is not zero, the author checks that the address is not a kernel address. I don't know the exact reason for the "BUG" comment, but it seems to me that the author plans to be able to attach segments within the address space of the kernel.

- `segment.c:610,611`
  Checking that `name` is a null terminated string at existing virtual memory.

- `segment.c:613,615`
  `sno` is the slot for the new segment. The first empty slot is used. The check for `ESEG` at line `:614` is ensuring that the "extra segment" used during `exec` is kept available. Otherwise, the process could not use `sysexec`.

- `segment.c:617,618`
  No more segments for this process.

- `segment.c:620,622`
  `len` is now an integral number of pages.

- `segment.c:624,642`
  The comment is very descriptive. It is very likely that a big hole exist in virtual memory right below the user stack. Most checking is done by `isoverlap`. By the way, wouldn't it be better to ensure that virtual addresses used by `ESEG` are kept unused? It doesn't matter too much because that portion of the user address space is used while the maps are set for the temporary stack mapping.

  `isoverlap()` *Would the segment overlap with an existing one?*

- `segment.c:554,571`
  `isoverlap` must iterate through the whole segment array for the process, checking that neither `va` nor `newtop` are contained within any segment. During all this time, `seg` is not locked. However, only the current process could deallocate its `seg` entries or attach new entries, and the current process is currently doing the `segattach`, so the lock is not really needed. Nevertheless, the lock could prevent future BUGs, in case the author changes his mind and uses `segattach` for a non-current (not `up`) process.

- `segment.c:644,646`
  A hole found, `isoverlap` is called once more after rounding `va` to a page boundary. Perhaps `va` should be truncated before line `:634`, and these lines avoided. Besides, it is not clear for me why `Esoverlap` (and not `Enovmem`) is raised, although the author knows why, I think that `Enovmem` could be perfectly raised here too.

- `segment.c:648,650`
  `name` is the segment "class" specified by the user. On each machine, an array of physical segments is kept at `physseg`. If the segment class name matches the name of one of these physical segments, the routine continues at `:653`. Otherwise `Ebadarg` is raised. You see how a `segattach` can only be done for segments declared in `physseg`. By the way, since the `found` label is used only to avoid returning `Ebadarg`, perhaps an `if` could have been better. The code is clear anyway, isn't it?

- `../pc/segment.h:1,8`
  For Intel PCs, the only known segments are "shared" and "memory" (there are more ones, as you will see). But for other architectures, `physseg` may contain exotic physical segments (read the manual page). I think that the name is `phys` because usually, processes attach to segments representing physical memory like device-provided memory, memory locks, etc.

- `../port/segment.c:654,664`
  Unless the segment length be bigger than `SEGMAXSIZE`, the segment is attached. `newseg` is creating the new segment, it will be either a `SHARED` or a `BSS` segment, and page faults with zero fill it on demand.

Physical segments

`addphysseg()` *Adds a new segment class.*

- `segment.c:573,598`
  I told you that `../pc/segment.h` had just a couple of "shared" and "memory" segments (classes) defined. There is a routine `addphysseg` which is called by device drivers to declare the existence of physical memory segments important for them. For example, in `../pc/vgas3.c:103`, the S3 video card driver calls `addphysseg` to add an entry to `physseg`, with attribute `SG_PHYSICAL` (physical memory used for the card) and name `s3screen` (the video memory). This call is done by `s3linear`, which is the S3 `VGAdev` procedure used to enable a

linear mode in the card (see `../pc/devvga.c` and `../pc/screen.c`). After this segment is added by calling `addphysseg`, a segattach can be done for `s3screen` to get to the video memory.

The routine `addphysseg` only checks that there is free room after the initialized entries (those with a name) and before the last entry (which must be a null entry) to add the new segment. Should there be space, the `new` segment given is linked into the array.

6.6.2 Detaching segments

`syssegdetach()` *`segdetach(2)` system call. Detaches a segment.*

- `sysproc.c:621,631`
  `syssegdetach` is the counterpart of `syssegattach`. It detaches a segment from the address space. The `seglock` must be acquired now. Routines that do not want `seg` entries to disappear under their feet acquire this lock too.

- `sysproc.c:633,644`
  The first argument is an address contained in the segment to be detached. The `seg` array is searched until the entry number (`i`) is found and `s` is set to the segment being detached.

- `sysproc.c:646,651`
  `arg` is a pointer to the user arguments for the system call, therefore it resides within the user stack. if this address is contained in the segment to be detached, it is the stack segment the one detached. In this case, the system refuses to detach the segment. A process *always* needs a stack. Although the check is nice, perhaps a check against `SG_STACK` would be more clear.

  The system does not seem to refuse detaches for the text segment, although the manual page suggests so.

- `sysproc.c:652,660`
  The segment is released by the call to `putseg`.

  By the way, perhaps a `segdettach` routine could contain most of `syssegdetach` code, as it happens with `segattach`.

`syssegfree()` *`segfree(2)` system call. Releases (part of) a segment.*

- `sysproc.c:664,686`
  `syssegfree` releases (note the call to `mfreeseg`) part of the memory held by the segment. This routine calls `seg` instead of locating the segment itself to find the segment containing `from` and lock it. Perhaps the routine could be generalized to return the index for the segment in the `seg` array so that others (e.g. `syssegdetach`) could benefit from it too. What happens to the portion of the address space released depends on the kind of segment. For instance, should it be a BSS segment, pages would be later zero-filled on demand.

6.6.3 Resizing segments

`syssegbrk()` *segbrk(2) system call. Resizes a segment.*

- `sysproc.c:588,608`
  `syssegbrk` resizes the segment. Only `SHARED`, `BSS`, and `SHDATA` segments can be resized. The reason is that text and data segments correspond (and are paged from) an image of a text file. Only for "anonymous memory" does `brk` make sense. `ibrk` does all the work.

`syssegbrk`
`ibrk()` *Resizes a segment.*

- `segment.c:431,454`
  `addr` is the new end address for the segment. If should be at least the segment `base`. The comment says that BSS might be overlapping the data segment, in which case `:449` is checking that `addr` is smaller than `base` for a `BSEG` and `addr` is bigger than `base` for the `DSEG` (otherwise an error is raised). However, `segbrk` finds the first segment where `addr` is contained, and `:448` would never be true.

  Nevertheless, in a previous implementation of `segbrk` (note `:sysproc.c:688,693`), the segment resized was always the BSS segment, in which case `:448` could be true for old Plan 9 binaries and lines `:448,454` would leave `addr` being the start of the BSS segment. Perhaps it would have been better to let `sysbrk_` do the check, and either keep `ibrk` assuming that `addr` always resides within segment bounds, or keep `ibrk` checking that `addr` is within the segment. The reason for doing so is that should `sysbrk_` disappear, the weird BSS check in `ibrk` may be forgotten and kept there.

- `segment.c:456,463`
  The new `top` and `size` for the segment is computed. If the segment is to shrink, `mfreeseg` releases the (now unused) memory of the segment. Since segment `base` and `top` are not updated, any page fault on the released part of the segment would make the segment grow again. Perhaps lines `:494,495` should be copied before line `:462`. Although the current behavior is perfectly reasonable (and compatible with what is said in `segbrk(2)`.).

- `segment.c:465,468`
  The segment is growing. Since the only segments resized (`sysproc.c:600,607`) are `SHARED`, `BSS`, and `SHDATA` segments, which have the swap file as their backing store, no resize is allowed if there is no free swap space.

- `segment.c:470,478`
  The whole list of segments is scanned searching for a segment containing `newtop`. If `newtop` is not contained within other segment, the space from the current `top` up to `newtop` is available. However, there could be a case when there is a segment starting after `top`, but ending before `newtop`, in which the check would miss that there is another segment overlapping the portion of the virtual address space being allocated. Perhaps the check could be changed to see if `base` for any segment is between `top` and `newtop`.

- `segment.c:480,492`
  `mapsize` recomputed and `map` is reallocated to have enough space for the new `Ptes`—`map` could be using the small `ssegmap` array.

- `segment.c:494,497`
  Segment bounds updated. Segments grow, but they really never shrink—only memory is reused if they are pretending to shrink.

6.6.4 Flushing segments

`syssegflush()` *segflush(2) system call. Flushes a segment cache.*

- `segment.c:681,729`
  `syssegflush` does a flush of the processor cache for memory within the segment. This seems to be used only by instruction simulator commands. The routine flushes a range of the user address space, which may spawn several segments. For each segment, `pteflush` is called to flush `Ptes` affected.

`syssegflush`
  `pteflush()` *Flushes the cache for a Pte.*

- `segment.c:667,678`
  `pteflush` sets the `cachectl` entry for the page (if not paged out) to `PG_TXTFLUSH`. As you may remember, this means that for several architectures (not the Intel), the processor cache entries for the given pages may be flushed later by the `flushmmu` call at `:727`.

6.6.5 Segment profiling

`segclock()` *Update segment profiling counters.*

- `segment.c:731,745`
  Not really a system call, but each time the clock ticks, if the processor was running user code, `../pc/clock.c:76,79` would call `segclock` giving the current saved program counter for the user process. The call is made after adding to the word in the bottom of the user stack the number of ticks in a millisecond. The kernel is maintaining a "clock" for the running process in the bottom of its user stack. If `profile` is set in the `TSEG` for the process, `segclock` adds the time passed to the value in `s->profile[0]`. Moreover, if the program counter for the user was within this segment, it is converted to a relative offset within `TSEG` and another time counter incremented. The `>>LRESPROF` is used to have one time counter per `LRESPROF` instructions in the text segment.

  If you think of it, the author is filling up `profile` with an array of clocks for the process. The first clock counts the time the process has been executing (one ms added every time a ms happens and the process was running), following clocks have the time spent by the process within the first, second,... group of `LRESPROF` instructions. The user can later inspect these clocks and learn where is his program spending time. This is crucial to let the user know what should be optimized, and what not.

6.7 Intel MMU handling

During the chapter about system startup you already read some code for MMU handling in the Intel PC. In this section you will be reading the code which has been used by the machine independent code for memory management, and was not discussed previously.

6.7.1 Flushing entries

`flushmmu()` *Flushes MMU translations.*

- `../pc/mmu.c:80,89`
  `flushmmu` was called whenever the page table was changed, and translations should be updated. Remember that the TLB may keep cached entries. All it does is to set the `newtlb` flag and switch to the current page table. On the Intel, a load of the page directory base register flushes the TLB too.

`flushmmu`
  `mmuswitch()` *Switches the address space in the MMU.*

- `mmu.c:110,127`
  `mmuswitch` (in this case), notices the `newtlb` and calls `mmuptefree` to flush the current set of entries. Later, the switch is done. As you should know by now, if the process has a `mmupdb` installed, that page table is used, otherwise, the prototype page table for the processor is used. The virtual address of the `Page` used to keep the first level page table is used to access its entries, and its physical address is used to set the `pdbr` (aka `cr3`).

  One important thing here is that the routine maps the `Mach` structure (together with the small scheduler stack) for the current processor at `MACHADDR`. `PDX` is getting the index in the first level page table for the virtual address `MACHADDR`. Each processor has at `MACHADDR` a map of its `Mach`, as you saw in the starting up chapter.

`flushmmu`
  `mmuswitch`
    `mmuptefree()` *Releases translations.*

- `mmu.c:91,108`
  `mmuptefree` starts with the `Page` noted in `mmuused`, and iterates through the whole set of pages linked through the `next` field. For each such page, its entry in the `pdb` is set to nil (i.e. invalid). `pdb` is set to be the virtual address of the `mmupdb`, to update later its contents. The `Page` keeps in `daddr` the index in the PDB for it. `mmuused` is a list of `Pages` used to keep secondary page tables for the process. So, `mmuptefree` is just clearing (invalidating) entries in the first level page table. Al those pages are linked later into `mmufree`. All second level page tables linked through `mmuused` are free now. The process will have to suffer page faults to get new second level tables again. This time, the machine independent MMU code will instruct `mmu.c` to fill up the entries according to changes in the `Segment` entries.

Can you see how the portable virtual memory data structures dictate what the machine dependent code should do? Can you see how they can differ from what the MMU has installed?

- `dat.h:90,95`
  `mmupdb`, `mmufree`, and `mmuused` are just `Pages`. The fields `va` and `pa` of such pages are used to update entries using the virtual address space and to get the page frame address for giving it to either the MMU or to an entry in PD. (By the why, see `../port/portdat.h:628` if you don't understand why I told you to look into `PMMU` and not into `Proc`).

6.7.2 Adding entries

`putmmu()` *Updates a translation.*

- `mmu.c:195,196`
  `putmmu` is called by the machine independent code to add an entry to the MMU page table for the current process. This happens when `fixfault` is repairing a page fault. How does this work?

  You know that each processor has a prototype page table, which includes mappings for kernel space as well as an identity mapping for physical memory. The kernel uses this prototype page table if the process has not its own one. So, when a new process is created, it starts using this page table until it suffers a page fault and `fixfault` calls `putmmu`.

- `mmu.c:203,204`
  If the process did not have its own page table, one is created. `mmupdballoc` does the job. Following calls to `putmmu` due to page faults will find (and use) the existing PDB allocated here.

`mmupdballoc()` *Allocates a PDB .*

- `mmu.c:173,193`
  `mmupdballoc` takes a new `Page`, either by allocating it (with `newpage`) or by using one from a free list at `pdbpool`. `va` is set in the `Page` as the kernel virtual address for the page frame and the page is kept mapped for kernel usage (no-op on Intels). It is initialized by copying the prototype page table from the processor, which you know was initialized. Initialized PDBs are placed back to the `pdbpool` when they are no longer used by the process, so that their allocation could be faster. Since `newpage` is given a nil `s` pointer, it will keep on trying to allocate a page instead of returning failure (as it does when called from `fixfault`).

- `mmu.c:205,206`
  Back to `putmmu`, `pdb` is now a kernel pointer to the PDB and `pdbx` is the entry in the first level page table for the virtual address to be mapped to `pa` using the `Page` supplied (note it's not used on Intels).

- `mmu.c:208,217`
  If there is no second-level page table (entry invalid in the PDB), must allocate

one. They are allocated on demand. Again, `Pages` for second level page tables are reused. If there is one in the free list (`mmufree`) that is used, otherwise `newpage` is called to allocate a new one. `clear` is set for `newpage`, so it is zeroed, setting all entries as invalid. If the page is new, a kernel map is made, otherwise, the kernel map was already done at page allocation time.

- `mmu.c:218`
  The entry in the PDB is set as valid, usable for protection ring 3, and allowing writes. The Intel checks permissions on the entry in the PDB, and then it checks permissions on the entry in the second label page table, every time an address is used (of course, TLB caches such things).

- `mmu.c:219,221`
  `mmuused` is updated to contain the new page, so that `mmuptefree` could know which entries are used. As only a few entries of the 1024 existing entries are really used, that saves a lot of time. `daddr` (not used here as a disk address) is used to keep the index in the PDB. PDB used entries can be found pretty quickly.

- `mmu.c:224,225`
  At this point, the second level page table exists. `pte` is set pointing to the second level page table. `page->va` should be equal to `KADDR(PPN(pdb[pdbx]))`. However, the author prefers to obtain the pointer through the entry in the PDB, probably to be sure that things are working properly. The entry in the second level page table for `va` is set with `pa` together with the `USER` bit, which says that ring 3 can access the page. When `fixfault` calls `putmmu`, `pa` is not just the page frame address for `va`, `fixfault` sets some of its offset bits to specify which permissions should be enabled (e.g. `PTEWRITE`, etc.), so the author ORs `pa` to keep those bits set.

- `mmu.c:227,231`
  Again ensuring that the `Mach` page is mapped, although it should be mapped already if the PDB did exist, and it will be mapped by `mmuswitch` in any case. Just for safety.

6.7.3 Adding and looking up entries

`mmuwalk()` *Walks to an MMU entry (and perhaps creates it).*

- `mmu.c:233,234`
  You saw how `mmuwalk` was called to get a pointer to the entry in the MMU page table for a given `va`. If `level` is 1, the entry wanted is the one in the first level page table, should it be 2, the entry wanted is the one in the second level page table. Other architectures may accept more than two levels (note the clean generic interface for the routine), but intels have just two. `create` can be set to request that the second level table be created. `mmu.c` itself uses `mmuwalk`, and `meminit` uses it too to create a prototype table. Unlike `putmmu`, `mmuwalk` does not require `up` to have a valid current process.

- `mmu.c:245,247`
  `table` is set to point to the entry in PDB for the given `va`. If the entry is nil, and it shouldn't be created, there is nothing to do.

- `mmu.c:249,268`
  Just two levels on Intels. For level 1, `table` is the pointer to the entry. For level 2, if the entry in `*table` was not valid, a second level page table is allocated and the entry in the PDB updated. This only happens when `create` was set. Finally, `table` is set to point to the virtual address for the second level page table, and a pointer to its entry for `va` returned.

`mmukmap()` *Adds a kernel map to the MMU.*

- `mmu.c:307,308`
  `mmukmap` adds a translation to the MMU page table for a page within the kernel portion of the address space. You saw how `mmukmap` was called during boot time, while initializing memory.

- `mmu.c:314,318`
  `mach0` is set to point to the `Mach` structure for the boot processor. If it has bit 0x10 set in its `cr4` and the processor is not so old that does not support it, super-pages can be used (Remember, using entries in the first-level page table to map 4MB). `pse` is set if super-pages can be used.

- `mmu.c:321,326`
  If `va` is not given, `mmukmap` understands that it is doing the identity map at `KZERO`. Otherwise, it is truncated to be a page address.

- `mmu.c:328,399`
  This loop maps the range of physical memory going from `pa` to `pa+size-1`. The routine can be used both to map a single page and to map a whole range of (4K) pages.

- `mmu.c:331`
  `table` is pointing to the first level entry in the prototype PDB at processor zero.

- `mmu.c:335,379`
  The entry is valid. If the entry had the `PTESIZE` bit, it was a map for 4M without using a second level page table (`:337,357`). In this case, `x` is the start of a 4M super- page frame, which should be the same of `pa`—at each pass, `pa` is the physical address being mapped. `pa` is set either to the end of the mapped memory or to the end of the super-page, preparing things up for the next iteration or for terminating the loop. The loop continues after the portion mapped by the super-page. If `pa` is already `pae`, all remaining pages to be mapped were already mapped by the super-page and the loop finishes.

  If the entry (`:336`) did not have the `PTESIZE` bit, it is pointing to a second level page table (`:360,376`). In this case, `mmuwalk` is used to get the entry in the second level page table (note again: processor zero PDB). If the entry is valid, `pa` is advanced past the already mapped page.

  `sync` is not-zero when a second level page table entry has been scanned.

- `mmu.c:382,298`
  The entry was not mapped. If `pse` says so, a whole super-page is used to map the desired address. This assumes that all the 4MB of physical memory (4MB aligned) where `pa` stands are to be mapped; hence the checks at line `:387`. It is very important to use super-pages since the Intel has different TLB entries for super-pages and regular pages. By making the kernel use super-pages, the user TLB entries are not disturbed while servicing interrupts and system calls. If no super-pages can be used (or the mapping does not contain a whole 4M super-page), `mmuwalk` is used to create the second level entry. `pgsz` is set so that `:396,397` can prepare the next pass. `sync` is not-zero when any entry has been added to the processor zero PDB.

- `mmu.c:402,407`
  `mmukmapsync` is called if `mmukmap` was called and either the mapping was at a second-level page table, or an entry was updated—you also saw a call to `mmukmapsync` before.

`mmukmap`

`mmukmapsync()` *Synchronizes kernel maps in page tables.*

- `mmu.c:272,273`
  `mmukmapsync` updates prototype page tables for all processors by looking at the boot processor page table. It fixes any missing entry. So, apart of initial memory mappings created during boot time, any kernel map added later only has to be set at processor 0. That is precisely what `mmukmap` does!

  Other processors will fault when using memory mapped at a different processor, but `mmukmapsync` will notice that there is a map in `mach0->pdb`, and will copy such map to the faulting processor page table. Note also how it flushes the tlb, and how the page table for the current process is updated to repair any page fault there.

  Faults at these kernel mapped pages will happen only for memory not mapped initially during boot time.

Epilogue

And now what?

Although we have gone a long way already, you still have devices in the Plan 9 kernel that remain to be read. You should try to understand them.

Besides, Plan 9 is much more than its kernel. For example, the code for the local file system provider (`kfs`) is really interesting to learn how to structure a partition into a set of files.

It is also illustrative to read the code for user commands, including the `plumber`, `rio`, `sam`, and `acme`. You can learn a lot by doing so; not just a lot about operating systems, but also a lot about good programming practices.

Finally, you could be so kind to let me know whatever suggestion you may have about this document.

Have fun with Plan 9!

Bibliography

[1] Francisco J. Ballesteros. Advanced Operating Systems Course Web site. http://gsyc.escet.urjc.es/docencia/asignaturas/ampliacion˙ssoo, 2000.

[2] Francisco J. Ballesteros. Operating Systems Design Course Web site. http://gsyc.escet.urjc.es/docencia/asignaturas/osd, 2000.

[3] Intel. Pentium iii processors - manuals. http://developer.intel.com/design/pentiumiii/manuals, 2000.

[4] Brian W Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley Publishing Company, 1999.

[5] Kerninghan and Ritchie. *The C Programming Language*. Prentice-Hall, 1988.

[6] Rob Pike. A manual for the Plan 9 assembler. *Plan 9 Programmer's manual, 3rd ed.*, vol. 2, 2000.

[7] Rob Pike. Acme: A User Interface for Programmers. *Plan 9 Programmer's manual, 3rd ed.*, vol. 2, 2000.

[8] Rob Pike. How to Use the Plan 9 C Compiler. *Plan 9 Programmer's manual, 3rd ed.*, vol. 2, 2000.

[9] Rob Pike et al. *Plan 9 Programmer's Manual: Volume 1: The Manuals. 3rd ed.* Harcourt Brace and Co., New York, NY, USA, 2000.

[10] Rob Pike et al. *Plan 9 Programmer's Manual: Volume 2: The Documents. 3rd ed.* Harcourt Brace and Co., New York, NY, USA, 2000.

[11] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[12] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in Plan 9. *Operating Systems Review*, 27(2):72–76, April 1993.

[13] D. L. Presotto. Multiprocessor streams for Plan 9. In *UKUUG. UNIX - The Legend Evolves. Proceedings of the Summer 1990 UKUUG Conference*, pages 11–19 (of xi + 260), Buntingford, Herts, UK, ???? 1990. UK Unix Users Group.

[14] et al. Rifkin, A.P. Rfs architectural overview. In *Summer Usenix Conference*. USENIX, 1986.

[15] Dennis M. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984. Also available from http://cm.bell-labs.com/cm/cs/who/dmr/st.html.

[16] Sun Microsystems, Inc. NFS: Network file system protocol specification. RFC 1094, Network Information Center, SRI International, March 1989.