The Linux Kernel Hackers' Guide

Alpha version 0.6

A hodgepodge collection of information, speculation, and ramblings about the Linux kernel. This is only a draft. Please mail any corrections, amplifications, suggestions, etc. to Michael K. Johnson, `johnsonm@nigel.vnet.net`, Editor.

Editorial comments look like this: [**This is an editorial comment.**] I invite answers to any questions in these comments. The more help I get on these, the fewer of these ugly comments newer versions of the guide will have. Some of these are merely large notices to myself to finish some task I started. If you would like to help by working on a section that has notes like this, please contact me to see what help I need.

This work is currently rather fragmented, and will remain in that state until most of the sections have been written, so that revision combining those sections can be done intelligently. Substantial revision to occur at that time should address the problems with unnecessarily duplicated information and lack of structure, and make the guide easier to follow and more succinct.

However, the section on device drivers should be helpful to some. Other sections are mostly a little out of date and in need of revision anyway. Please bear with me, or better yet, help.

UNIX is a trademark of X/Open
MS-DOS is a trademark of Microsoft Corporation.
Linux is not a trademark, and has no connection to UNIX™ or X/Open.
If any trademarks have been unintentionally unacknowledged, please inform the editor, Michael K. Johnson, 201 Howell Street, Apt. 1C, Chapel Hill, North Carolina 27514-4818, email `johnsonm@nigel.vnet.net`.

Introduction

The *The Linux Kernel Hackers' Guide* is inspired by all of us "kernel hacker wannabees" who just did not know enough about unix systems to hack the Linux kernel when it first came out, and had to learn slowly. This guide is designed to help you get up to speed on the concepts that are not intuitively obvious, and to document the internal structures of Linux so that you don't have to read the whole kernel source to figure out what is happening with one variable, or to discover the purpose of one function call.

Why Linux? Well, Linux is the first free unix clone for the 386 to be freely available. It is a complete re-write, and has been kept small, so it does not have a lot of the time-honored baggage that other free operating systems (like 386BSD) carry, and so is easier to understand and modify.

Unix has been around for over twenty years, but only in the last few years have microcomputers become powerful enough to run a modern protected, multiuser, multitasking operating system. Furthermore, unix implementations have not been free. Because of this, very little free documentation has been written, at least for the kernel internals.

Unix, though simple at first, has grown more and more appendages, and has become a very complex system, which only "wizards" understand. With Linux, however, we have a chance to change this, for a few reasons:

- Linux has a simple kernel, with well-structured interfaces.
- One person, Linus Torvalds, has control of what code is added to Linux, and he does this work gratis. This means that random pieces of code are not forced into the kernel by some company's politics, and the kernel interfaces stay relatively clean.
- The source is free, so many people can study it and learn to understand it, becoming "wizards" in their own right, and eventually contribute code to the effort.

It is our hope that this book will help the nascent kernel hacker learn how to hack the Linux kernel, by giving an understanding of how the kernel is structured.

Thanks to...

Linus Torvalds, of course, for starting this whole time sink, and for gently providing explanations whenever necessary. He has done a wonderful job of keeping the kernel source code understandable and neat. I can't imagine having learned so much in the past few years without Linux.

Krishna Balasubramanian and **Douglas Johnson,** for writing much of the section on memory management, and helping with the rest.

Stanley Scalsky, for helping document the system call interface.

Rik Faith, for writing the section on how to write a SCSI device driver.

Robert Baruch, for the review of *Writing UNIX Device Drivers* and for his help with the section on writing device drivers.

Linux Journal, for providing me with a Linux-related job, and for allowing me to do work on the KHG on their time.

Kim Johnson, my wife, for tolerating and encouraging me even when I spend my time on crazy stuff like Linux.

Copyright Acknowledgements:

Linux Memory Management: The original version of this document is copyright © 1993 Krishna Balasubramanian. Some changes copyright © 1993 Michael K. Johnson and Douglas R. Johnson.

How System Calls Work: The original version of this document is copyright © 1993 Stanley Scalsky. Some changes copyright © 1993 Michael K. Johnson

Writing a SCSI Device Driver The original version of this document is copyright © 1993 Rickard E. Faith. Some modifications are copyright © 1993 Michael K. Johnson. The author has approved the inclusion of this material, despite the slightly more restrictive copyright on this whole document. The original copyright restrictions, *which still apply to any work derived solely from this work,* are:

If you wish to make a derived work, please start from the original document. To do so, please contact Rickard E. Faith, `faith@cs.unc.edu`. The original is available for anonymous ftp as ftp.cs.unc.edu:/pub/faith/papers/scsi.paper.tar.gz.

Contents

Chapter 0

Before You Begin…

0.1 Typographical Conventions

Bold — Used to mark **new concepts**, **WARNINGS**, and **keywords** in a language.

italics — Used for *emphasis* in text, and occasionally for quotes or introductions at the beginning of a section.

slanted — Used to mark **meta-variables** in the text, especially in representations of the command line. For example,

```
ls -l foo
```

where *foo* would “stand for” a filename, such as `/bin/cp`. Sometimes, this might be difficult to see, and so the text is put in angle brackets, like this: ⟨*slanted*⟩.

`Typewriter` — Used to represent screen interaction, as in

```
ls -l /bin/cp
-rwxr-xr-x  1 root    wheel    12104 Sep 25 15:53 /bin/cp
```

Also used for code examples, whether it is “C” code, a shell script, or something else, and to display general files, such as configuration files. When necessary for clarity’s sake, these examples or figures will be enclosed in thin boxes.

Key — Represents a key to press. You will often see it in this form:

Press return to continue.

◇ A diamond in the margin, like a black diamond on a ski hill, marks “danger” or “caution.” Read paragraphs marked this way carefully.

0.2 Assumptions

To read *The Linux Kernel Hackers' Guide*, you should have a reasonably good understanding of C. That is, you should be able to read C code without having to look up everything. You should be able to write simple C programs, and understand `struct`'s, pointers, macros, and ANSI C prototyping. You do not have to have a thorough knowledge of the standard I/O library, because the standard libraries are not available in the kernel. Some of the more often used standard I/O functions have been rewritten for use within the kernel, but these are explained in this book where necessary.

You should be able to use a good text editor, recompile the Linux kernel, and do basic system administration tasks, such as making new device entries in /dev/.

You should also be able to read, as I do not offer support for this book…

> “*Hello, sir, I'm having some problems with this book you wrote.*”
> “Yes?”
> “*I can't read it.*”
> “Is it plugged in?”
> “*Yes. I also tried a lamp in that socket, so I know it is getting power. But I really don't think that's the problem.*”
> “Why not?”
> “*I can't read.*”
> “ Oh. Well, let's start here. See this? Repeat after me: *The cat sat on the rat…* ”

0.3 Hacking Wisdom

This is a collection of little things that you need to know before you start hacking. It is rather rambling, and almost resembles a glossary in form, but it is not a reference, but rather a hacker's narative, a short course in kernel hacking.

Static variables

Always initialize static variables. I cannot overemphasize this. Many seemingly random bugs have been caused by not initializing static variables. Because the kernel is not really a standard executable, the **bss** segment may or may not be zeroed, depending on the method used for booting.

`libc` unavailable

Much of `libc` is unavailable. That is, all of `libc` is unavailable, but many of the most common functions are duplicated. See the section [**not here yet**] for simple documentation of these functions. Most of the documentation for these are the section 3 and section 9 man pages.

Linux is not UNIX™

However, it is close. It is not plan 9, nor is it Mach. It is not primarily intended to be a great commercial success. People will not look kindly upon suggestions to change it fundamentally to attain any of these goals. It has been suggested that part of the reason that the quality of the Linux kernel is so high is the unbending devotion of the Linux kernel hackers to having fun playing with their new kernel.

Useful references

You will encounter certain references that you will need to understand. For instance, "Stevens" and "Bach". Read the annotated bibliography (Appendix A) for a list of books that you should at least recognize references to, even if you have not read them.

Read the FAQ

Chapter 1

Device Drivers

1.1 What is a Device Driver?

Making hardware work is tedious. To write to a hard disk, for example, requires that you write magic numbers in magic places, wait for the hard drive to say that it is ready to receive data, and then feed it the data it wants, very carefully. To write to a floppy disk is even harder, and requires that the program supervise the floppy disk drive almost constantly while it is running.

Instead of putting code in each application you write to control each device, you share the code between applications. To make sure that that code is not compromised, you protect it from users and normal programs that use it. If you do it right, you will be able to add and remove devices from your system without changing your applications at all. Furthermore, you need to be able to load your program into memory and run it, which the operating system also does. So, an operating system is essentially a preiviledged, general, sharable library or low-level hardware and memory and process control functions and routines.

All versions of UN⋆X have an abstract way of reading and writing devices. By making the devices act as much as possible like regular files, the same calls (`read()`, `write()`, etc.) can be used for devices and files. Within the kernel, there are a set of functions, registered with the filesystem, which are called to handle requests to do I/O on “device special files,” which are those which represent devices.[1]

All devices controlled by the same device driver are given the same **major number,** and of those with the same major number, different devices are distinguished by different **minor numbers.**[2]

[1] See `mknod(1,2)` for an explanation of how to make these files.

[2] This is not strictly true, but is close enough. If you understand where it is not true, you don’t

This chapter explains how to write any type of Linux device driver that you might need to, including character, block, SCSI, and network drivers. [**Well, it will when it is done**. . .] It explains what functions you need to write, how to initialize your drivers and obtain memory for them efficiently, and what function are built in to Linux to make your job easier.

Creating device drivers for Linux is easier than you might think. It merely involves writing a few functions and registering them with the Virtual Filesystem Switch (VFS), so that when the proper device special files are accessed, the VFS can call your functions.

However, a word of warning is due here: Writing a device driver **is** writing a part of the Linux kernel. This means that your driver runs with kernel permissions, and can do anything it wants to: write to any memory, reformat your hard drive, damage your monitor or video card, or even break your dishes, if your dishwasher is controlled by your computer. Be careful.

Also, your driver will run in kernel mode, and the Linux kernel, like most UN⋆X kernels, is non-pre-emptible. This means that if you driver takes a long time to work without giving other programs a chance to work, your computer will appear to "freeze" when your driver is running. Normal user-mode pre-emptive scheduling does not apply to your driver.

If you choose to write a device driver, you must take everything written here as a guide, and no more. I cannot guarantee that this chapter will be free of errors, and I cannot guarantee that you will not damage your computer, even if you follow these instructions exactly. It is highly unlikely that you will damage it, but I cannot guarantee against it. There is only one "infallible" direction I can give you: **Back up!** Back up before you test your new device driver, or you may regret it later.

1.2 User-space device drivers

It is not always necessary to write a device driver for a device, especially in applications where no two applications will compete for the device. The most useful example of this is a memory-mapped device, but you can also do this with devices in I/O space (devices accessed with `inb()` and `outb()`, etc.). If your process is running as superuser (root), you can use the `mmap()` call to map some of your process memory to actual memory locations, by `mmap()`'ing a section of /dev/mem. When you have done this mapping, it is pretty easy to write and read from real memory addresses just as you would read and write any variables.

need to read this section, and if you don't but want to learn, read the code for the tty devices, which uses up 2 major numbers, and may use a third and possibly fourth by the time you read this.

If your driver needs to respond to interrupts, then you really need to be working in kernel space, and need to write a real device driver, as there is no good way at this time to deliver interrupts to user processes. Although the DOSEMU project has created something called the SIG (Silly Interrupt Generator) which allows interrupts to be posted to user processes (I believe through the use of signals), the SIG is not particularly fast, and should be thought of as a last resort for things like DOSEMU.

An interrupt (for those who don't know) is an asyncronous notification posted by the hardware to alert the device driver of some condition. You have likely dealt with 'IRQ's when setting up your hardware; an IRQ is an "Interrupt ReQuest line," which is triggered when the device wants to talk to the driver. This may be because it has data to give to the drive, or because it is now ready to receive data, or because of some other "exceptional condition" that the driver needs to know about. It is similar to user-level processes receiving a **signal,** so similar that the same `sigaction` structure is used in the kernel to deal with interrupts as is used in user-level programs to deal with signals. Where the user-level has its signals delivered to it by the kernel, the kernel has interrupt delivered to it by hardware.

If your driver must be accessible to multiple processes at once, and/or manage contention for a resource, then you also need to write a real device driver at the kernel level, and a user-space device driver will not be sufficient or even possible.

1.2.1 Example: `vgalib`

A good example of a user-space driver is the `vgalib` library. The standard `read()` and `write()` calls are really inadequate for writing a really fast graphics driver, and so instead there is a library which acts conceptually like a device driver, but runs in user space. Any processes which use it **must** run setuid root, because it uses the `ioperm()` system call. It is possible for a process that is not setuid root to write to /dev/mem if you have a group `mem` or `kmem` which is allowed write permission to /dev/mem and the process is properly setgid, but only a process running as root can execute the `ioperm()` call.

There are several I/O ports associated with VGA graphics. `vgalib` creates symbolic names for this with `#define` statements, and then issues the `ioperm()` call like this to make it possible for the process to read and write directly from and to those ports:

```
if (ioperm(CRT_IC, 1, 1)) {
    printf("VGAlib: can't get I/O permissions \n");
    exit (-1);
}
ioperm(CRT_IM,  1, 1);
ioperm(ATT_IW, 1, 1);
```

[. . .]

It only needs to do error checking once, because the only reason for the `ioperm()` call to fail is that it is not being called by the superuser, and this status is not going to change.

◇ After making this call, the process is allowed to use `inb` and `outb` machine instructions, but only on the specified ports. These instructions can be accessed without writing directly in assembly by including `<linux/asm>`, but will only work if you compile **with optimization on,** by giving the `-O`*?* to gcc. Read `<linux/asm>` for details.

After arranging for port I/O, `vgalib` arranges for writing directly to kernel memory with the following code:

```
/* open /dev/mem */
if ((mem_fd = open("/dev/mem", O_RDWR) ) < 0) {
    printf("VGAlib: can't open /dev/mem \n");
    exit (-1);
}

/* mmap graphics memory */
if ((graph_mem = malloc(GRAPH_SIZE + (PAGE_SIZE-1))) == NULL) {
    printf("VGAlib: allocation error \n");
    exit (-1);
}
if ((unsigned long)graph_mem % PAGE_SIZE)
    graph_mem += PAGE_SIZE - ((unsigned long)graph_mem % PAGE_SIZE);
graph_mem = (unsigned char *)mmap(
    (caddr_t)graph_mem,
    GRAPH_SIZE,
    PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_FIXED,
    mem_fd,
    GRAPH_BASE
);
if ((long)graph_mem < 0) {
    printf("VGAlib: mmap error \n");
    exit (-1);
}
```

It first opens /dev/mem, then allocates memory enough so that the mapping can be done on a page (4 KB) boundary, and then attempts the map. `GRAPH_SIZE` is the size of VGA memory, and `GRAPH_BASE` is the first address of VGA memory in /dev/mem. Then by writing to the address that is returned by `mmap()`, the process is actually writing to screen memory.

1.2.2 Example: mouse conversion

If you want a driver that acts a bit more like a kernel-level driver, but does not live in kernel space, you can also make a fifo, or named pipe. This usually lives in the /dev/ directory (although it doesn't need to) and acts substantially like a device once set up. However, fifo's are one-directional only — they have one reader and one writer.

For instance, it used to be that if you had a PS/2-style mouse, and wanted to run XFree86, you had to create a fifo called /dev/mouse, and run a program called mconv which read PS/2 mouse "droppings" from /dev/psaux, and wrote the equivalent microsoft-style "droppings" to /dev/mouse. Then XFree86 would read the "droppings" from /dev/mouse, and it would be as if there were a microsoft mouse connected to /dev/mouse.[3]

1.3 Device Driver Basics

We will assume that you decide that you do not wish to write a user-space device, and would rather implement your device in the kernel. You will probably be writing writing two files, a `.c` file and a `.h` file, and possibly modifying other files as well, as will be described below. We will refer to your files as foo.c and foo.h, and your driver will be the `foo` driver.

[**Should I include at the beginning of this section an example of chargen and charsink? Many writers do, but I don't know that it is the best way. I'd like people's opinions on this.**]

1.3.1 Namespace

One of the first things you will need to do, before writing any code, is to name your device. This name should be a short (probably two or three character) string. For instance, the parallel device is the "`lp`" device, the floppies are the "`fd`" devices, and SCSI disks are the "`sd`" devices. As you write your driver, you will give your functions names prefixed with your chosen string to avoid any namespace confusion. We will call your prefix `foo,` and give your functions names like `foo_read(), foo_write(),` etc.

1.3.2 Allocating memory

Memory allocation in the kernel is a little different from memory allocation in normal user-level programs. Instead of having a `malloc()` capable of delivering almost unlimited

[3] Even though XFree86 is now able to read PS/2 style "droppings", the concepts in this example still stand. If you have a better example, I'd be glad to see it.

amounts of memory, there is a `kmalloc()` function that is a bit different:

- Memory is provided in pieces whose size is a power of 2, except that pieces larger than 128 bytes are allocated in blocks whose size is a power of 2 minus some small amount for overhead. You can request any odd size, but memory will not be used any more efficiently if you request a 31-byte piece than it will if you request a 32 byte piece. Also, there is a limit to the amount of memory that can be allocated, which is currently 131056 bytes.

- `kmalloc()` takes a second argument, the priority. This is used as an argument to the `get_free_page()` function, where it is used to determine when to return. The usual priority is `GFP_KERNEL`. If it may be called from within an interrupt, use `GFP_ATOMIC` and be truly prepared for it to fail (i.e. don't panic). This is because if you specify `GFP_KERNEL`, `kmalloc()` may sleep, which cannot be done on an interrupt. The other option is `GFP_BUFFER`, which is used only when the kernel is allocating buffer space, and never in device drivers.

To free memory allocated with `kmalloc()`, use one of two functions: `kfree()` or `kfree_s()`. These differ from `free()` in a few ways as well:

- `kfree()` is a macro which calls `kfree_s()` and acts like the standard `free()` outside the kernel.

- If you know what size object you are freeing, you can speed things up by calling `kfree_s()` directly. It takes two arguments: the first is the pointer that you are freeing, as in the single argument to `kfree()`, and the second is the size of the object being freed.

See section 1.6 for more information on `kmalloc()`, `kfree()`, and other useful functions.

The other way to acquire memory is to allocate it at initialization time. Your initialization function, `foo_init()`, takes one argument, a pointer to the current end of memory. It can take as much memory as it wants to, save a pointer or pointers to that memory, and return a pointer to the new end of memory. The advantage of this over statically allocating large buffers (`char bar[20000]`) is that if the foo driver detects that the foo device is not attached to the computer, the memory is not wasted. The `init()` function is discussed in Section 1.3.6.

Be gentle when you use kmalloc. Use only what you have to. Remember that kernel memory is unswappable, and thus allocating extra memory in the kernel is a far worse thing to do in the kernel than in a user-level program. Take only what you need, and free it when you are done, unless you are going to use it right away again.

[I believe that it is possible to allocate swappable memory with the vmalloc function, but that will be documented in the VMM section when it gets written. In the meantime, enterprising hackers are encouraged to look it up themselves.]

1.3.3 Character vs. block devices

There are two main types of devices under all UN⋆X systems, character and block devices. Character devices are those for which no buffering is performed, and block devices are those which are accessed through a cache. Block devices must be random access, but character devices are not required to be, though some are. Filesystems can only be mounted if they are on block devices.

Character devices are read from and written to with two function: `foo_read()` and `foo_write()`. The `read()` and `write()` calls do not return until the operation is complete. By contrast, block devices do not even implement the `read()` and `write()` functions, and instead have a function which has historically been called the "strategy routine." Reads and writes are done through the buffer cache mechanism by the generic functions `bread()`, `breada()`, and `bwrite()`. These functions go through the buffer cache, and so may or may not actually call the strategy routine, depending on whether or not the block requested is in the buffer cache (for reads) or on whether or not the buffer cache is full (for writes). A request may be asyncronous: `breada()` can request the strategy routine to schedule reads that have not been asked for, and to do it asyncronously, in the background, in the hopes that they will be needed later. A more complete explanation of the buffer cache is presented below in Section **??** **[When that section is written. . .]**

The sources for character devices are kept in . . . /kernel/chr_drv/, and the sources for block devices are kept in . . . /kernel/blk_drv/. They have similar interfaces, and are very much alike, except for reading and writing. Because of the difference in reading and writing, initialization is different, as block devices have to register a strategy routine, which is registered in a different way than the `foo_read()` and `foo_write()` routines of a character device driver. Specifics are dealt with in Section 1.4.1 and Section 1.5.1

1.3.4 Interrupts vs. Polling

Hardware is slow. That is, in the time it takes to get information from your average device, the CPU could be off doing something far more useful than waiting for a busy but slow device. So to keep from having to **busy-wait** all the time, **interrupts** are provided which can interrupt whatever is happening so that the operating system can do some task and return to what it was doing without losing information. In an ideal world, all devices would probably work by using interrupts. However, on a PC or clone, there are only a few interrupts available for use by your peripherals, so some drivers have to poll the hardware: ask the hardware if it is ready to transfer data yet. This unfortunately wastes time, but it sometimes needs to be done.

Also, some hardware (like memory-mapped displays) is as fast as the rest of the machine, and does not generate output asyncronously, so an interrupt-driven driver would be rather silly, even if interrupts were provided.

In Linux, many of the drivers are interrupt-driven, but some are not, and at least one can be either, and can be switched back and forth at runtime. For instance, the `lp` device (the parallel port driver) normally polls the printer to see if the printer is ready to accept output, and if the printer stays in a not ready phase for too long, the driver will sleep for a while, and try again later. This improves system performance. However, if you have a parallel card that supplies an interrupt, the driver will utilize that, which will usually make performance even better.

There are some important programming differences between interrupt-driven drivers and polling drivers. To understand this difference, you have to understand a little bit of how system calls work under UN⋆X. The kernel is not a separate task under UN⋆X. Rather, it is as if each process has a copy of the kernel. When a process executes a system call, it does not transfer control to another process, but rather, the process changes execution modes, and is said to be "in kernel mode." In this mode, it executes kernel code which is trusted to be safe.

In kernel mode, the process can still access the user-space memory that it was previously executing in, which is done through a set of macros: `get_fs_*()` and `memcpy_fromfs()` read user-space memory, and `put_fs_*()` and `memcpy_tofs()` write to user-space memory. Because the process is still running, but in a different mode, there is no question of where in memory to put the data, or where to get it from. However, when an interrupt occurs, any process might currently be running, so these macros cannot be used — if they are, they will either write over random memory space of the running process or cause the kernel to panic.

[**Explain how to use `verify_area()`, which is only used on cpu's that don't provide write protection while operating in kernel mode, to check whether the area is safe to write to.**]

Instead, when scheduling the interrupt, a driver must also provide temporary space in which to put the information, and then sleep. When the interrupt-driven part of the driver has filled up that temporary space, it wakes up the process, which copies the information from that temporary space into the process' user space and returns. In a block device driver, this temporary space is automatically provided by the buffer cache mechanism, but in a character device driver, the driver is responsible for allocating it itself.

1.3.5 The sleep-wakeup mechanism

[Begin by giving a general description of how sleeping is used and what it does. This should mention things like all processes sleeping on an event are woken at once, and then they contend for the event again, etc...]

Perhaps the best way to try to understand the Linux sleep-wakeup mechanism is to read the source for the `__sleep_on()` function, used to implement both the `sleep_on()` and `interruptible_sleep_on()` calls.

```
static inline void __sleep_on(struct wait_queue **p, int state)
{
    unsigned long flags;
    struct wait_queue wait = { current, NULL };

    if (!p)
        return;
    if (current == task[0])
        panic("task[0] trying to sleep");
    current->state = state;
    add_wait_queue(p, &wait);
    save_flags(flags);
    sti();
    schedule();
    remove_wait_queue(p, &wait);
    restore_flags(flags);
}
```

A `wait_queue` is a circular list of pointers to task structures, defined in `<linux/wait.h>` to be

```
struct wait_queue {
    struct task_struct * task;
    struct wait_queue * next;
};
```

`state` is either `TASK_INTERRUPTIBLE` or `TASK_UNINTERUPTIBLE`, depending on whether or not the sleep should be interruptable by such things as system calls. In general, the sleep should be interruptible if the device is a slow one; one which can block indefinitely, including terminals and network devices or pseudodevices.

`add_wait_queue()` turns off interrupts, if they were enabled, and adds the new `struct wait_queue` declared at the beginning of the function to the list `p`. It then recovers the original interrupt state (enabled or disabled), and returns.

`save_flags()` is a macro which saves the process flags in its argument. This is done to preserve the previous state of the interrupt enable flag. This way, the `restore_flags()` later can restore the interrupt state, whether it was enabled or disabled. `sti()` then allows interrupts to occur, and `schedule()` finds a new process to run, and switches to it. Schedule will not choose this process to run again until the state is changed to `TASK_RUNNING` by `wake_up()` called on the same wait queue, `p`, or conceivably by something else.

The process then removes itself from the `wait_queue`, restores the orginal interrupt condition with `restore_flags()`, and returns.

Whenever contention for a resource might occur, there needs to be a pointer to a `wait_queue` associated with that resource. Then, whenever contention does occur, each process that finds itself locked out of access to the resource sleeps on that resource's `wait_queue`. When any process is finished using a resource for which there is a `wait_queue`, it should wake up and processes that might be sleeping on that `wait_queue`, probably by calling `wake_up()`, or possibly `wake_up_interruptible()`.

If you don't understand why a process might want to sleep, or want more details on when and how to structure this sleeping, I urge you to buy one of the operating systems textbooks listed in Appendix A and look up **mutual exclusion** and **deadlock.**

[**This is a cop-out. I should take the time to explain and give examples, but I am not trying to write an OS text, and I want to keep this under 1000 pages**...]

1.3.5.1 More advanced sleeping

If the `sleep_on()`/`wake_up()` mechanism in Linux does not satisfy your device driver needs, you can code your own versions of `sleep_on()` and `wake_up()` that fit your needs. For an example of this, look at the serial device driver (.../kernel/chr_drv/serial.c) in function `block_til_ready()`, where quite a bit has to be done between the `add_wait_queue()` and the `schedule()`.

1.3.6 The VFS

The Virtual Filesystem Switch, or **VFS**, is the mechanism which allows Linux to mount many different filesystems at the same time. In the first versions of Linux, all filesystem access went straight into routines which understood the `minix` filesystem. To make it possible for other filesystems to be written, filesystem calls had to pass through a layer of indirection which would switch the call to the routine for the correct filesystem. This was done by some generic code which can handle generic cases and a structure of pointers to functions which handle specific cases. One structure is of interest to the device driver writer; the `file_operations` structure.

From /usr/include/linux/fs.h:

```
struct file_operations {
    int  (*lseek)   (struct inode *, struct file *, off_t, int);
    int  (*read)    (struct inode *, struct file *, char *, int);
    int  (*write)   (struct inode *, struct file *, char *, int);
    int  (*readdir) (struct inode *, struct file *, struct dirent *,
                     int count);
    int  (*select)  (struct inode *, struct file *, int,
                     select_table *);
    int  (*ioctl)   (struct inode *, struct file *, unsigned int,
                     unsigned int);
    int  (*mmap)    (struct inode *, struct file *, unsigned long,
                     size_t, int, unsigned long);
    int  (*open)    (struct inode *, struct file *);
    void (*release) (struct inode *, struct file *);
};
```

Essentially, this structure constitutes a parital list of the functions that you may have to write to create your driver.

This section details the actions and requirements of the functions in the `file_operations` structure. It documents all the arguments that these functions take. [**It should also detail all the defaults, and cover more carefully the possible return values.**

1.3.6.1 The `lseek()` function

This function is called when the system call `lseek()` is called on the device special file representing your device. An understanding of what the system call `lseek()` does should be sufficient to explain this function, which moves to the desired offset. It takes these four arguments:

`struct inode * inode`
: Pointer to the inode structure for this device.

`struct file * file`
: Pointer to the file structure for this device.

`off_t offset`
: Offset **from origin** to move to.

`int origin` 0 = take the offset from absolute offset 0 (the beginning).
1 = take the offset from the current position.
2 = take the offset from the end.

`lseek()` returns `-errno` on error, or ≥ 0 the absolute position after the lseek.

If there is no `lseek()`, the kernel will take the default action, which is to modify the `file->f_pos` element. For an `origin` of 2, the default action is to return `-EINVAL` if `file->f_inode` is NULL, otherwise it sets `file->f_pos` to `file->f_inode->i_size` + `offset`. Because of this, if `lseek()` should return an error for your device, you must write an `lseek()` function which returns that error.

1.3.6.2 The `read()` and `write()` functions

The read and write functions read and write a character string to the device. If there is no `read()` or `write()` function in the `file_operations` structure registered with the kernel, and the device is a character device, `read()` or `write()` system calls, respectively, will return `-EINVAL`. If the device is a block device, these functions should not be implemented, as the VFS will route requests through the buffer cache, which will call your strategy routine. See Section 1.5.2 for details on how the buffer cache does this. The `read` and `write` functions take these arguments:

`struct inode * inode`
This is a pointer to the inode of the device special file which was accessed. From this, you can do several things, based on the `struct inode` declaration about 100 lines into /usr/include/linux/fs.h. For instance, you can find the minor number of the file by this construction: `unsigned int minor = MINOR(inode->i_rdev);` The definition of the `MINOR` macro is in `<linux/fs.h>`, as are many other useful definitions. Read fs.h and a few device drivers for more details, and see section 1.6 for a short description. `inode->i_mode` can be used to find the mode of the file, and there are macros available for this, as well.

`struct file * file`
Pointer to file structure for this device.

`char * buf` This is a buffer of characters to read or write. It is located in *user-space* memory, and therefore must be accessed using the `get_fs*()`, `put_fs*()`, and `memcpy*fs()` macros detailed in section 1.6. User-space memory is inaccessible during an interrupt, so if your driver is interrupt driven, you will have to copy the contents of your buffer into a queue.

`int count` This is a count of characters in `buf` to be read or written. It is the size of `buf`, and is how you know that you have reached the end of `buf`, as `buf` is not guaranteed to be null-terminated.

1.3.6.3 The `readdir()` function

This function is another artifact of `file_operations` being used for implementing filesystems as well as device drivers. Do not implement it. The kernel will return `-ENOTDIR` if the system call `readdir()` is called on your device special file.

1.3.6.4 The `select()` function

The `select()` function is generally most useful with character devices. It is usually used to multiplex reads without polling — the application calls the `select()` system call, giving it a list of file descriptors to watch, and the kernel reports back to the program on which file descriptor has woken it up. It is also used as a timer. However, the `select()` function in your device driver is not directly called by the system call `select()`, and so the `file_operations` `select()` only needs to do a few things. Its arguments are:

`struct inode * inode`
Pointer to the inode structure for this device.

`struct file * file`
Pointer to the file structure for this device.

`int sel_type`
The select type to perform:

`SEL_IN`	read
`SEL_OUT`	write
`SEL_EX`	exception

`select_table * wait`
If `wait` is not NULL and there is no error condition caused by the select, `select()` should put the process to sleep, and arrange to be woken up when the device becomes ready, usually through an interrupt. If `wait` is NULL, then the driver should quickly see if the device is ready, and return even if it is not. The `select_wait()` function does this already.

If the calling program wants to wait until one of the devices upon which it is selecting becomes available for the operation it is interested in, the process will have to be put to sleep until one of those operations becomes available. This does **not** require use of a `sleep_on*()` function, however. Instead the `select_wait()` function is used. (See section 1.6 for the definition of the `select_wait()` function). The sleep state that `select_wait()` will cause is the same as that of `sleep_on_interruptible()`, and, in fact, `wake_up_interruptible()` is used to wake up the process.

However, `select_wait()` will not make the process go to sleep right away. It returns directly, and the `select()` function you wrote should then return. The process isn't put to sleep until the system call `sys_select()`, which originall called your `select()` function, uses the information given to it by the `select_wait()` function to put the process to sleep. `select_wait()` adds the process to the wait queue, but `do_select()` (called from `sys_select()`) actually puts the process to sleep by changing the process state to `TASK_INTERRUPTIBLE` and calling `schedule()`.

The first argument to `select_wait()` is the same `wait_queue` that should be used for a `sleep_on()`, and the second is the `select_table` that was passed to your `select()` function.

After having explained all this in excruciating detail, here are two rules to follow:

1. Call `select_wait()` if the device is not ready, and return 0.
2. Return 1 if the device is ready.

If you provide a `select()` function, do not provide timeouts by setting `current->timeout`, as the `select()` mechanism uses `current->timeout`, and the two methods cannot co-exist, as there is only one `timeout` for each process. Instead, consider using a timer to provide timeouts. See the description of the `add_timer()` function in section 1.6 for details.

1.3.6.5 The `ioctl()` function

The `ioctl()` function processes ioctl calls. The structure of your `ioctl()` function will be: first error checking, then one giant (possibly nested) switch statement to handle all possible ioctls. The ioctl number is passed as `cmd`, and the argument to the ioctl is passed as `arg`. It is good to have an understanding of how `ioctls` ought to work before making them up. If you are not sure about your ioctls, do not feel ashamed to ask someone knowledgeable about it, for a few reasons: you may not even need an ioctl for your purpose, and if you do need an ioctl, there may be a better way to do it than what you have thought of. Since ioctls are the least regular part of the device interface, it takes perhaps the most work to get this part right. Take the time and energy you need to get it right.

`struct inode * inode`
Pointer to the inode structure for this device.

`struct file * file`
Pointer to the file structure for this device.

`unsigned int cmd`
This is the ioctl command. It is generally used as the switch variable for a case statement.

`unsigned int arg`
This is the argument to the command. This is user defined. Since this is the same size as a `(void *)`, this can be used as a pointer to user space, accessed through the fs register as usual.

Returns: `-errno` on error
Every other return is user-defined.

If the `ioctl()` slot in the `file_operations` structure is not filled in, the VFS will return `-EINVAL`. However, in all cases, if `cmd` is one of `FIOCLEX`, `FIONCLEX`, `FIONBIO`, or `FIOASYNC`, default processing will be done:

`FIOCLEX` 0x5451
Sets the close-on-exec bit.

`FIONCLEX` 0x5450
Clears the close-on-exec bit.

`FIONBIO` 0x5421
If `arg` is non-zero, set `O_NONBLOCK`, otherwise clear `O_NONBLOCK`.

`FIOASYNC` 0x5452
If `arg` is non-zero, set `O_SYNC`, otherwise clear `O_SYNC`. `O_SYNC` is not yet implemented, but it is documented here and parsed in the kernel for completeness.

Note that you have to avoid these four numbers when creating your own ioctls, since if they conflict, the VFS ioctl code will interpret them as being one of these four, and act appropriately, causing a very hard to track down bug.

1.3.6.6 The `mmap()` function

`struct inode * inode`
Pointer to inode structure for device.

`struct file * file`
Pointer to file structure for device.

`unsigned long addr`
Beginning of address in main memory to `mmap()` into.

`size_t len` Length of memory to `mmap()`.

`int prot` One of:

`PROT_READ`	region can be read.
`PROT_WRITE`	region can be written.
`PROT_EXEC`	region can be executed.
`PROT_NONE`	region cannot be accessed.

`unsigned long off`
Offset in the file to `mmap()` from. This address in the file will be mapped to address `addr`.

[Here, give a pointer to the documentation for the new vmm (Virtual Memory Mangament) interface, and show how the functions can be used by a device `mmap()` function. Krishna should have the documentation for the vmm interface in the memory management section.]

1.3.6.7 The `open()` and `release()` functions

`struct inode * inode`
Pointer to inode structure for device.

`struct file * file`
Pointer to file structure for device.

`open()` is called when a device special files is opened. It is the policy mechanism responsible for ensuring consistency. If only one process is allowed to open the device at once, `open()` should lock the device, using whatever locking mechanism is appropriate, usually setting a bit in some state variable to mark it as busy. If a process already is using the device (if the busy bit is already set) then `open()` should return `-EBUSY`. If more than one process may open the device, this function is responsible to set up any necessary queues that would not be set up in `write()`. If no such device exists, `open()` should return `-ENODEV` to indicate this. Return 0 on success.

`release()` is called only when the process closes its last open file descriptor on the files. If devices have been marked as busy, `release()` should unset the busy bits if appropriate. If you need to clean up `kmalloc()`'ed queues or reset devices to preserve their sanity, this is the place to do it. If no `release()` function is defined, none is called.

1.3.6.8 The `init()` function

This function is not actually included in the `file_operations` structure, but you are required to implement it, because it is this function that registers the `file_operations` structure with the VFS in the first place — without this function, the VFS could not route any requests to the driver. This function is called when the kernel first boots and is configuring itself. `init()` is passed a variable holding the address of the current end of used memory. The init function then detects all devices, allocates any memory it will want based on how many devices exist (this is often used to hold such things as queues, for interrupt driven devices), and then, saving the addresses it needs, it returns the new end of memory. You will have to call your `init()` function from the correct place: for a character device, this is `chr_dev_init()` in .../kernel/chr_dev/mem.c. In general, you will only pass the `memory_start` variable to your `init()` function.

While the `init()` function runs, it registers your driver by calling the proper registration function. For character devices, this is `register_chrdev()`.[4] `register_chrdev()` takes three arguments: the major device number (an int), the "name" of the device (a string), and the address of the *device*`_fops` `file_operations` structure.

When this is done, and a character or block special file is accessed, the VFS filesystem switch automagically routes the call, whatever it is, to the proper function, if a function exists. If the function does not exist, the VFS routines take some default action.

The `init()` function usually displays some information about the driver, and usually reports all hardware found. All reporting is done via the `printk()` function.

1.4 Character Device Drivers

[Write appropriate blurb here]

[4] See section??

1.4.1 Initialization

Besides functions defined by the `file_operations` structure, there is at least one other function that you will have to write, the `foo_init()` function. You will have to change `chr_dev_init()` in chr_drv/mem.c to call your `foo_init()` function. `foo_init()` will take one argument, `long mem_start`, which will be the address of the current end of allocated memory. If your driver needs to allocate more than 4K of contiguous space at runtime, here is the place. Simply save `mem_start` in an appropriate variable, add however much space you need to `mem_start`, and return the new value. Your driver will now have exclusive access to the memory between the old and new values of `mem_start`.

`foo_init()` should first call `register_chrdev()` to register itself and avoid device number contention. `register_chrdev()` takes three arguments:

`int major` This is the major number which the driver wishes to allocate.

`char *name` This is the symbolic name of the driver. It is currently not used for anything, but this may change in the future.

`struct file_operations *f_ops`
This is the address of your `file_operations` structure defined in Section **??**

Returns: 0 if no other character device has registered with the same major number.
non-0 if the call fails, presumably because another character device has already allocated that major number.

Generally, the `foo_init()` routine will then attempt to detect the hardware that it is supposed to be driving. It should make sure that all necessary data structures are filled out for all present hardware, and have some way of ensuring that non-present hardware does not get accessed. [**detail different ways of doing this.**]

1.4.2 Interrupts vs. Polling

In a polling driver, the `foo_read()` and `foo_write()` functions are pretty easy to write. Here is an example of `foo_write()`:

```
static int foo_write(struct inode * inode, struct file * file,
                     char * buf, int count)
{
    unsigned int minor = MINOR(inode->i_rdev);
    char ret;
    while (count > 0) {
        ret = foo_write_byte(minor);
if (ret < 0) {
            foo_handle_error(WRITE, ret, minor);
            continue;
        }
        buf++ = ret; count--
    }
    return count;
}
```

foo_write_byte() and foo_handle_error() are either functions defined elsewhere in foo.c or pseudocode. WRITE would be a constant or #define.

It should be clear from this example how to code the foo_read() function as well.

Interrupt-driven drivers are a little more difficult. Here is an example of a foo_write() that is interrupt-driven:

```
static int foo_write(struct inode * inode, struct file * file,
                     char * buf, int count)
{
    unsigned int minor = MINOR(inode->i_rdev);
    unsigned long copy_size;
    unsigned long total_bytes_written = 0;
    unsigned long bytes_written;
    struct foo_struct *foo = &foo_table[minor];

    do {
        copy_size = (count <= FOO_BUFFER_SIZE ? count : FOO_BUFFER_SIZE);
        memcpy_fromfs(foo->foo_buffer, buf, copy_size);

        while (copy_size) {
            /* initiate interrupts */

            if (some_error_has_occured)  {
                /* handle error condition */
            }

            current->timeout = jiffies + FOO_INTERRUPT_TIMEOUT;
                /* set timeout in case an interrupt has been missed */
            interruptible_sleep_on(&foo->foo_wait_queue);
            bytes_written = foo->bytes_xfered;
            foo->bytes_written = 0;
            if (current->signal & ~current->blocked) {
                if (total_bytes_written + bytes_written)
                    return total_bytes_written + bytes_written;
                else
                    return -EINTR; /* nothing was written, system
                                      call was interrupted, try again */
            }
        }

        total_bytes_written += bytes_written;
        buf += bytes_written;
        count -= bytes_written;

    } while (count > 0);

    return total_bytes_written;
}

static void foo_interrupt(int irq)
{
    struct foo_struct *foo = &foo_table[foo_irq[irq]];

    /* Here, do whatever actions ought to be taken on an interrupt.
       Look at a flag in foo_table to know whether you ought to be
       reading or writing. */

    /* Increment foo->bytes_xfered by however many characters were
       read or written */

    if (buffer too full/empty)
        wake_up_interruptible(&foo->foo_wait_queue);
}
```

Again, a foo_read() function is written analagously. foo_table[] is an array of structures, each of which has several members, some of which are foo_wait_queue and bytes_xfered, which can be used for both reading and writing. foo_irq[] is an array of 16 integers, and is used for looking up which entry in foo_table[] is associated with the irq generated and reported to the foo_interrupt() function.

To tell the interrupt-handling code to call foo_interrupt(), you need to use either request_irq() or irqaction(). This is either done when foo_open() is called, or if you want to keep things simple, when foo_init() is called. request_irq() is the simpler of the two, and works rather like an old-style signal handler. It takes two arguments: the first is the number of the `irq` you are requesting, and the second is a pointer to your interrupt handler, which must take an integer argument (the irq that was generated) and have a return type of `void`. `request_irq()` returns `-EINVAL` if `irq` $>$ 15 or if the pointer to the interrupt handler is `NULL`, `-EBUSY` if that interrupt has already been taken, or 0 on success.

`irqaction()` works rather like the user-level `sigaction()`, and in fact reuses the `sigaction` structure. The `sa_restorer()` field of the sigaction structure is not used, but everything else is the same. See the entry for `irqaction()` in Section 1.6, **Supporting Functions**, for further information about `irqaction()`.

1.4.3 TTY drivers

[**The reasons that this section has not been written are that I don't know enough about TTY stuff yet. Ted re-wrote the tty devices for the 1.1 series, but I haven't studied them yet.**]

1.5 Block Device Drivers

To mount a filesystem on a device, it must be a block device driven by a block device driver. This means that the device must be a random access device, not a stream device. In other words, you must be able to seek to any location on the physical device at any time.

You do not provide `read()` and `write()` routines for a block device. Instead, your driver uses `block_read()` and `block_write()`, which are generic functions, provided by the VFS, which will call the **strategy** routine, or `request()` function, which you write in place of `read()` and `write()` for your driver. This strategy routine is also called by the **buffer cache** (See section ??), which is called by the VFS routines (See chapter ??) which is how normal files on normal filesystems are read and written.

Requests for I/O are given by the buffer cache to a routine called `ll_rw_block()`, which constructs lists of requests ordered by an **elevator algorithm,** which sorts the lists to make accesses faster and more efficient. It, in turn, calls your `request()` function to actually do the I/O.

Note that although SCSI disks and CDROMs are considered block devices, they are handled specially (as are all SCSI devices). Refer to section 1.7, Writing a SCSI Driver, for details.[5]

[5] Although SCSI didsks and CDROMs are block devices, SCSI tapes, like other tapes, are generally used as character devices.

1.5.1 Initialization

Initialization of block devices is a bit more complex than initialization of character devices, especially as some "initialization" has to be done at compile time. There is also a `register_blkdev()` call that corresponds to the character device `register_chrdev()` call, which the driver must call to say that it is present, working, and active.

1.5.1.1 The file blk.h

At the top of your driver code, after all other included header files, you need to write two lines of code:

```
#define MAJOR_NR DEVICE_MAJOR
#include "blk.h"
```

where *DEVICE*`_MAJOR` is the major number of your device. drivers/block/blk.h requires the use of the `MAJOR_NR` define to set up many other defines and macros for your driver.

Now you need to edit blk.h. Under `#ifdef MAJOR_NR`, there is a section of defines that are conditionally included for certain major numbers, protected by `#elif (MAJOR_NR ==` *DEVICE*`_MAJOR)`. At the end of this list, you will add another section for your driver. In that section, the following lines are required:

```
#define DEVICE_NAME "device"
#define DEVICE_REQUEST do_dev_request
#define DEVICE_ON(device) /* usually blank, see below */
#define DEVICE_OFF(device) /* usually blank, see below */
#define DEVICE_NR(device) (MINOR(device))
```

`DEVICE_NAME` is simply the device name. See the other entries in blk.h for examples.

`DEVICE_REQUEST` is your strategy routine, which will do all the I/O on the device. See section 1.5.3 for more details on the strategy routine.

`DEVICE_ON` and `DEVICE_OFF` are for devices that need to be turned on and off, like floppies. In fact, the floppy driver is currently the only device driver which uses these defines.

`DEVICE_NR(device)` is used to determine the number of the physical device from the minor device number. For instance, in the `hd` driver, since the second hard drive starts at minor 64, `DEVICE_NR(device)` is defined to be `(MINOR(device)>>6)`.

If your driver is interrupt-driven, you will also set

```
#define DEVICE_INTR do_dev
```

which will become a variable automatically defined and used by the remainder of blk.h, specifically by the `SET_INTR()` and `CLEAR_INTR` macros.

You might also consider setting these defines:

```
#define DEVICE_TIMEOUT DEV_TIMER
#define TIMEOUT_VALUE n
```

where *n* is the number of jiffies (clock ticks; hundredths of a second on Linux/386) to time out after if no interrupt is received. These are used if your device can become “stuck”: a condition where the driver waits indefinitely for an interrupt that will never arrive. If you define these, they will automatically be used in `SET_INTR` to make your driver time out. Of course, your driver will have to be able to handle the possibility of being timed out by a timer. See section ?? for an explanation of how to do this.

1.5.1.2 Recognizing PC standard partitions

[Inspect the routines in genhd.c and include detailed, correct instructions on how to use them to allow your device to use the standard dos partitioning scheme.]

1.5.2 The Buffer Cache

[Here, it should be explained briefly how `ll_rw_block()` is called, about getblk() and bread() and breada() and bwrite(), etc. A real explanation of the buffer cache is reserved for the VFS reference section, where something on the complexity order of Bach’s treatment of the buffer cache should exist.

For now, we assume that the reader understands the concepts behind the buffer cache. If you are a reader and don’t, please email me and I’ll help you, which will also help me put my thoughts together for that section.]

1.5.3 The Strategy Routine

All reading and writing of blocks is done through the **strategy routine**. This routine takes no arguments and returns nothing, but it knows where to find a list of requests for I/O (`CURRENT`, defined by default as `blk_dev[MAJOR_NR].current_request`), and knows how to get data from the device into the blocks. It is called with interrupts **disabled** so as to avoid race conditions, and is responsible for turning on interrupts with a call to `sti()` before returning.

The strategy routine first calls the `INIT_REQUEST` macro, which makes sure that requests are really on the request list and does some other sanity checking. `add_request()` will have already sorted the requests in the proper order according to the elevator algorithm (using an insertion sort, as it is called once for every request), so the strategy routine "merely" has to satisfy the request, call `end_request(1)`, which will take the request off the list, and then if there is still another request on the list, satisfy it and call `end_request(1)`, until there are no more requests on the list, at which time it returns.

If the driver is interrupt-driven, the strategy routine need only schedule the first request to occur, and have the interrupt-handler call `end_request(1)` and the call the strategy routine again, in order to schedule the next request. If the driver is not interrupt-driven, the strategy routine may not return until all I/O is complete.

If for some reason I/O fails permanently on the current request, `end_request(0)` must be called to destroy the request.

A request may be for a read or write. The driver determines whether a request is for a read or write by examining `CURRENT->cmd`. If `CURRENT->cmd == READ`, the request is for a read, and if `CURRENT->cmd == WRITE`, the request is for a write. If the device has seperate interrupt routines for handling reads and writes, `SET_INTR(`*n*`)` must be called to assure that the proper interrupt routine will be called.

[Here I need to include samples of both a polled strategy routine and an interrupt-driven one. The interrupt-driven one should provide seperate read and write interrupt routines to show the use of `SET_INTR`.]

1.5.4 Example Drivers

[I'm not sure this belongs here — we'll see. I'll leave the stub here for now.]

1.6 Supporting Functions

Here is a list of many of the most common supporting functions available to the device driver writer. If you find other supporting functions that are useful, please point them out to me. I know this is not a complete list, but I hope it is a helpful one.

`add_request()`

```
static void add_request(struct blk_dev_struct *dev,
                        struct request * req)
```

This is a static function in ll_rw_block.c, and cannot be called by other code. However, an understanding of this function, as well as an understanding of `ll_rw_block()`, may help you understand the strategy routine.

If the device that the request is for has an empty request queue, the request is put on the queue and the strategy routine is called. Otherwise, the proper place in the queue is chosen and the request is inserted in the queue, maintaining proper order by insertion sort.

Proper order (the elevator algorithm) is defined as:

a. Reads come before writes.

b. Lower minor numbers come before higher minor numbers.

c. Lower block numbers come before higher block numbers.

The elevator algorithm is implemented by the macro `IN_ORDER()`, which is defined in drivers/block/blk.h

Defined in: drivers/block/ll_rw_block.c
See also: `make_request()`, `ll_rw_block()`.

`add_timer()` `void add_timer(struct timer_list * timer)`

`#include <linux/timer.h>`

Installs the timer structures in the list `timer` in the timer list.

The `timer_list` structure is defined by:

```
struct timer_list {
        struct timer_list *next;
        struct timer_list *prev;
        unsigned long expires;
        unsigned long data;
        void (*function)(unsigned long);
};
```

In order to call `add_timer()`, you need to allocate a `timer_list` structure, and then call `init_timer()`, passing it a pointer to your `timer_list`. It will nullify the `next` and `prev` elements, which is the correct initialization. If necessary, you can allocate multiple `timer_list` structures, and link them into a list. Do make sure that you properly initialize all the unused pointers to `NULL`, or the timer code may get very confused.

For each struct in your list, you set three variables:

`expires` The number of jiffies (100ths of a second in Linux/86) after which to time out.

`function` Kernel-space function to run after timeout has occured.

`data` Passed as the argument to `function` when `function` is called.

Having created this list, you give a pointer to the first (usually the only) element of the list as the argument to `add_timer()`. Having passed that pointer, keep a copy of the pointer handy, because you will need to use it to modify the elements of the list (to set a new timeout when you need a function called again, to change the function to be called, or to change the data that is passed to the function) and to delete the timer, if necessary.

Note: This is *not* process-specific. Therefore, if you want to wake a certain process at a timeout, you will have to use the sleep and wake primitives. The functions that you install through this mechanism will run in the same context that interrupt handlers run in.

Defined in: kernel/sched.c
See also: `timer_table` in include/linux/timer.h, `init_timer()`, `del_timer()`.

`cli()` `#define cli() __asm__ __volatile__ ("cli"::)`

`#include <asm/system.h>`

Prevents interrupts from being acknowledged. `cli` stands for "CLear Interrupt enable".

See also: `sti()`

`del_timer` `void del_timer(struct timer_list * timer)`

`#include <linux/timer.h>`

Deletes the timer structures in the list `timer` in the timer list.

The timer list that you delete must be the address of a timer list you have earlier installed with `add_timer()`. Once you have called `del_timer()` to delete the timer from the kernel timer list, you may deallocate the memory used in the `timer_list` structures, as it is no longer referenced by the kernel timer list.

Defined in: kernel/sched.c
See also: `timer_table` in include/linux/timer.h, `init_timer()`, `add_timer()`.

`end_request()`

```
static void end_request(int uptodate)
```

```
#include "blk.h"
```

Called when a request has been satisfied or aborted. Takes one argument:

`uptodate`	If not equal to 0, means that the request has been satisfied. If equal to 0, means that the request has not been satisfied.

If the request was satisfied (`uptodate != 0`), `end_request()` maintains the request list, unlocks the buffer, and may arrange for the scheduler to be run at the next convenient time (`need_resched = 1`; this is implicit in `wake_up()`, and is not explicitly part of `end_request()`), before waking up all processes sleeping on the `wait_for_request` event, which is slept on in `make_request()`, `ll_rw_page()`, and `ll_rw_swap_file()`.

Note: This function is a static function, defined in drivers/block/blk.h for every non-SCSI device that includes blk.h. (SCSI devices do this differently; the high-level SCSI code itself provides this functionality to the low-level device-specific SCSI device drivers.) It includes several defines dependent on static device information, such as the device number. This is marginally faster than a more generic normal C function.

Defined in: kernel/blk_drv/blk.h
See also: `ll_rw_block()`, `add_request()`, `make_request()`.

`free_irq()`

```
void free_irq(unsigned int irq)
```

```
#include <linux/sched.h>
```

Frees an irq previously aquired with `request_irq()` or `irqaction()`. Takes one argument:

`irq`	interrupt level to free.

Defined in: kernel/irq.c
See also: `request_irq()`, `irqaction()`.

`get_user*()`

```
inline unsigned char get_user_byte(const char * addr)
inline unsigned short get_user_word(const short * addr)
inline unsigned long get_user_long(const int *addr)
```

`#include <asm/segment.h>`

Allows a driver to access data in user space, which is in a different segment than the kernel.

◇ **Note:** these functions may cause implicit I/O, if the memory being accessed has been swapped out, and therefore pre-emption may occur at this point. Do not include these functions in critical sections of your code even if the critical sections are protected by `cli()`/`sti()` pairs, because that implicit I/O will violate the integrity of your `cli()`/`sti()` pair. If you need to get at user-space memory, copy it to kernel-space memory *before* you enter your critical section.

These functions take one argument:

`addr` Address to get data from.

Returns: Data at that offset in user space.

Defined in: include/asm/segment.h
See also: `memcpy_*fs()`, `put_user*()`, `cli()`, `sti()`.

`inb(), inb_p()`

```
inline unsigned int inb(unsigned short port)
inline unsigned int inb_p(unsigned short port)
```

`#include <asm/io.h>`

Reads a byte from a port. `inb()` goes as fast as it can, while `inb_p()` pauses before returning. Some devices are happier if you don't read from them as fast as possible. Both functions take one argument:

`port` Port to read byte from.

Returns: The byte is returned in the low byte of the 32-bit integer, and the 3 high bytes are unused, and may be garbage.

Defined in: include/asm/io.h
See also: `outb()`, `outb_p()`.

`init_timer()`

Inline function for initializing `timer_list` structures for use with `add_timer()`.

Defined in: include/linux/timer.h
See also: `add_timer()`.

`irqaction()` `int irqaction(unsigned int irq, struct sigaction *new)`

`#include <linux/sched.h>`

Hardware interrupts are really a lot like signals. Therefore, it makes sense to be able to register an interrupt like a signal. The `sa_restorer()` field of the `struct sigaction` is not used, but otherwise it is the same. The int argument to the `sa.handler()` function may mean different things, depending on whether or not the IRQ is installed with the `SA_INTERRUPT` flag. If it is not installed with the `SA_INTERRUPT` flag, then the argument passed to the handler is a pointer to a register structure, and if it is installed with the `SA_INTERRUPT` flag, then the argument passed is the number of the IRQ. For an example of handler set to use the `SA_INTERRUPT` flag, look at how `rs_interrupt()` is installed in .../kernel/chr_drv/serial.c

The `SA_INTERRUPT` flag is used to determine whether or not the interrupt should be a "fast" interrupt. Normally, upon return from the interrupt, `need_resched`, a global flag, is checked. If it is set ($\neq 0$), then `schedule()` is run, which may schedule another process to run. They are also run with all other interrupts still enabled. However, by setting the `sigaction` structure member `sa_flags` to `SA_INTERRUPT`, "fast" interrupts are chosen, which leave out some processing, and very specifically do not call `schedule()`.

`irqaction()` takes two arguments:

`irq`	The number of the IRQ the driver wishes to acquire.
`new`	A pointer to a sigaction struct.
Returns:	`-EBUSY` if the interrupt has already been acquired, `-EINVAL` if `sa.handler()` is NULL, 0 on success.

Defined in: kernel/irq.c
See also: `request_irq()`, `free_irq()`

`IS_*(inode)`

```
IS_RDONLY(inode) ((inode)->i_flags & MS_RDONLY)
IS_NOSUID(inode) ((inode)->i_flags & MS_NOSUID)
IS_NODEV(inode) ((inode)->i_flags & MS_NODEV)
IS_NOEXEC(inode) ((inode)->i_flags & MS_NOEXEC)
IS_SYNC(inode) ((inode)->i_flags & MS_SYNC)
```

```
#include <linux/fs.h>
```

These five test to see if the inode is on a filesystem mounted the corresponding flag.

`kfree*()`

```
#define kfree(x) kfree_s((x), 0)
void kfree_s(void * obj, int size)
```

```
#include <linux/malloc.h>
```

Free memory previously allocated with `kmalloc()`. There are two possible arguments:

`obj` Pointer to kernel memory to free.

`size` To speed this up, if you know the size, use `kfree_s()` and provide the correct size. This way, the kernel memory allocator knows which bucket cache the object belongs to, and doesn't have to search all of the buckets. (For more details on this terminology, read mm/kmalloc.c.)

Defined in: mm/kmalloc.c, include/linux/malloc.h
See also: `kmalloc()`.

`kmalloc()`

```
void * kmalloc(unsigned int len, int priority)
```

```
#include <linux/kernel.h>
```

`kmalloc()` used to be limited to 4096 bytes. It is now limited to 131056 bytes $((32*4096)-16)$. Buckets, which used to be all exact powers of 2, are now a power of 2 minus some small number, except for numbers less than or equal to 128. For more details, see the implementation in mm/kmalloc.c.

`kmalloc()` takes two arguments:

`len` Length of memory to allocate. If the maximum is exceeded, kmalloc will log an error message of “`kmalloc of too large a block (%d bytes).`” and return `NULL`.

`priority` `GFP_KERNEL` or `GFP_ATOMIC`. If `GFP_KERNEL` is chosen, `kmalloc()` may sleep, allowing pre-emption to occur. This is the normal way of calling `kmalloc()`. However, there are cases where it is better to return immediately if no pages are available, without attempting to sleep to find one. One of the places in which this is true is in the swapping code, because it could cause race conditions, and another in the networking code, where things can happen at much faster speed that things could be handled by swapping to disk to make space for giving the networking code more memory. The most important reason for using `GFP_ATOMIC` is if it is being called from an interrupt, when you cannot sleep, and cannot receive other interrupts.

Returns: `NULL` on failure.
Pointer to allocated memory on success.

Defined in: mm/kmalloc.c
See also: `kfree()`

`ll_rw_block()`

```
void ll_rw_block(int rw, int nr, struct buffer_head *bh[])
```

```
#include <linux/fs.h>
```

No device driver will ever call this code: it is called only through the buffer cache. However, an understanding of this function may help you understand the function of the strategy routine.

After sanity checking, if there are no pending requests on the device's request queue, `ll_rw_block()` "plugs" the queue so that the requests don't go out until all the requests are in the queue, sorted by the elevator algorithm. `make_request()` is then called for each request. If the queue had to be plugged, then the strategy routine for that device is not active, and it is called, **with interrupts disabled. It is the responsibility of the strategy routine to re-enable interrupts.**

Defined in: devices/block/ll_rw_block.c
See also: `make_request()`, `add_request()`.

`MAJOR()`

```
#define MAJOR(a) (((unsigned)(a))>>8)
```

```
#include <linux/fs.h>
```

This takes a 16 bit device number and gives the associated major number by shifting off the minor number.

See also: `MINOR()`.

`make_request()`

```
static void make_request(int major, int rw, struct buffer_head
*bh)
```

This is a static function in ll_rw_block.c, and cannot be called by other code. However, an understanding of this function, as well as an understanding of `ll_rw_block()`, may help you understand the strategy routine.

`make_request()` first checks to see if the request is readahead or writeahead and the buffer is locked. If so, it simply ignores the request and returns. Otherwise, it locks the buffer and, except for SCSI devices, checks to make sure that write requests don't fill the queue, as read requests should take precedence.

If no spaces are available in the queue, and the request is neither readahead nor writeahead, `make_request()` sleeps on the event `wait_for_request`, and tries again when woken. When a space in the queue is found, the request information is filled in and `add_request()` is called to actually add the request to the queue.

Defined in: devices/block/ll_rw_block.c
See also: `add_request()`, `ll_rw_block()`.

`MINOR()`

```
#define MINOR(a) ((a)&0xff)
```

```
#include <linux/fs.h>
```

This takes a 16 bit device number and gives the associated minor number by masking off the major number.

See also: `MAJOR()`.

`memcpy_*fs()`

```
inline void memcpy_tofs(void * to, const void * from,
                        unsigned long n)
inline void memcpy_fromfs(void * to, const void * from,
                          unsigned long n)
```

```
#include <asm/segment.h>
```

Copies memory between user space and kernel space in chunks larger than one byte, word, or long. Be very careful to get the order of the arguments right!

◇ **Note:** these functions may cause implicit I/O, if the memory being accessed has been swapped out, and therefore pre-emption may occur at this point. Do not include these functions in critical sections of your code, even if the critical sections are protected by `cli()`/`sti()` pairs, because implicit I/O will violate the `cli()` protection. If you need to get at user-space memory, copy it to kernel-space memory *before* you enter your critical section.

These functions take three arguments:

`to`	Address to copy data to.
`from`	Address to copy data from.
`n`	Number of bytes to copy.

Defined in: include/asm/segment.h
See also: `get_user*()`, `put_user*()`, `cli()`, `sti()`.

`outb(), outb_p()`

```
inline void outb(char value, unsigned short port)
inline void outb_p(char value, unsigned short port)
```

```
#include <asm/io.h>
```

Writes a byte to a port. `outb()` goes as fast as it can, while `outb_p()` pauses before returning. Some devices are happier if you don't write to them as fast as possible. Both functions take two arguments:

`value`	The byte to write.
`port`	Port to write byte to.

Defined in: include/asm/io.h
See also: `inb()`, `inb_p()`.

`printk()`

```
int printk(const char* fmt, ...)
```

```
#include <linux/kernel.h>
```

`printk()` is a version of `printf()` for the kernel, with some restrictions. It cannot handle floats, and has a few other limitations, which are documented in kernel/vsprintf.c. It takes a variable number of arguments:

`fmt` Format string, `printf()` style.

`...` The rest of the arguments, `printf()` style.

Returns: Number of bytes written.

◇ **Note:** `printk()` may cause implicit I/O, if the memory being accessed has been swapped out, and therefore pre-emption may occur at this point. Also, `printk()` will set the interrupt enable flag, so **never use it in code protected by `cli()`.** Because it causes I/O, it is not safe to use in protected code anyway, even it if didn't set the interrupt enable flag.

Defined in: kernel/printk.c.

`put_user*()`

```
inline void put_user_byte(char val, char *addr)
inline void put_user_word(short val, short *addr)
inline void put_user_long(unsigned long val, unsigned long
*addr)
```

```
#include <asm/segment.h>
```

Allows a driver to write data in user space, which is in a different segment than the kernel. When entering the kernel through a system call, a selector for the current user space segment is put in the fs segment register, thus the names.

◇ **Note:** these functions may cause implicit I/O, if the memory being accessed has been swapped out, and therefore pre-emption may occur at this point. Do not include these functions in critical sections of your code even if the critical sections are protected by `cli()`/`sti()` pairs, because that implicit I/O will violate the integrity of your `cli()`/`sti()` pair. If you need to get at user-space memory, copy it to kernel-space memory *before* you enter your critical section.

These functions take two arguments:

`val` Value to write

`addr` Address to write data to.

Defined in: asm/segment.h
See also: `memcpy_*fs()`, `get_user*()`, `cli()`, `sti()`.

register_*dev()

```
int register_chrdev(unsigned int major, const char *name,
                    struct file_operations *fops)
int register_blkdev(unsigned int major, const char *name,
                    struct file_operations *fops)
```

```
#include <linux/fs.h>
#include <linux/errno.h>
```

Registers a device with the kernel, letting the kernel check to make sure that no other driver has already grabbed the same major number. Takes three arguments:

major	Major number of device being registered.
name	Unique string identifying driver. Used in the output for the /proc/devices file.
fops	Pointer to a file_operations structure for that device. This must **not** be NULL, or the kernel will panic later.

Returns: -EINVAL if major is ≥ MAX_CHRDEV or MAX_BLKDEV (defined in <linux/fs.h>), for character or block devices, respectively.
-EBUSY if major device number has already been allocated.
0 on success.

Defined in: fs/devices.c
See also: unregister_*dev()

request_irq()

```
int request_irq(unsigned int irq, void (*handler)(int),
                unsigned long flags, const char *device)
```

```
#include <linux/sched.h>
#include <linux/errno.h>
```

Request an IRQ from the kernel, and install an IRQ interrupt handler if successful. Takes four arguments:

irq	The IRQ being requested.

`handler` The handler to be called when the IRQ occurs. The argument to the handler function will be the number of the IRQ that it was invoked to handle.

`flags` Set to `SA_INTERRUPT` to request a "fast" interrupt or 0 to request a normal, "slow" one.

`device` A string containing the name of the device driver, *device*.

Returns: `-EINVAL` if `irq` > 15 or `handler` = `NULL`.
`-EBUSY` if `irq` is already allocated.
0 on success.

If you need more functionality in your interrupt handling, use the `irqaction()` function. This uses most of the capabilities of the `sigaction` structure to provide interrupt services similar to to the signal services provided by `sigaction()` to user-level programs.

Defined in: kernel/irq.c
See also: `free_irq()`, `irqaction()`.

`select_wait()`

```
inline void select_wait(struct wait_queue **wait_address,
                        select_table *p)
```

```
#include <linux/sched.h>
```

Add a process to the proper `select_wait` queue. This function takes two arguments:

`wait_address`
Address of a `wait_queue` pointer to add to the circular list of waits.

`p` If `p` is `NULL`, `select_wait` does nothing, otherwise the current process is put to sleep. This should be the `select_table *wait` variable that was passed to your `select()` function.

Defined in: linux/sched.h
See also: `*sleep_on(), wake_up*()`

`*sleep_on()`

```
void sleep_on(struct wait_queue ** p)
void interruptible_sleep_on(struct wait_queue ** p)
```

```
#include <linux/sched.h>
```

Sleep on an event, putting a `wait_queue` entry in the list so that the process can be woken on that event. `sleep_on()` goes into an uninteruptible sleep: The only way the process can run is to be woken by `wake_up()`. `interruptible_sleep_on()` goes into an interruptible sleep that can be woken by signals and process timeouts will cause the process to wake up. A call to `wake_up_interruptible()` is necessary to wake up the process and allow it to continue running where it left off. Both take one argument:

`p` Pointer to a proper `wait_queue` structure that records the information needed to wake the process.

Defined in: kernel/sched.c
See also: `select_wait()`, `wake_up*()`.

`sti()`

```
#define sti() __asm__ __volatile__ ("sti"::)
```

```
#include <asm/system.h>
```

Allows interrupts to be acknowledged. `sti` stands for “SeT Interrupt enable”.

Defined in: asm/system.h
See also: `cli()`.

`sys_get*()`

```
int sys_getpid(void)
int sys_getuid(void)
int sys_getgid(void)
int sys_geteuid(void)
int sys_getegid(void)
int sys_getppid(void)
int sys_getpgrp(void)
```

These system calls may be used to get the information described in the table below, or the information can be extracted directly from the process table, like this:
foo = `current->`*pid*;

pid	Process ID
uid	User ID
gid	Group ID
euid	Effective user ID
egid	Effective group ID
ppid	Process ID of process' parent process
pgid	Group ID of process' parent process

The system calls should not be used because they are slower *and* take more space. Because of this, they are no longer exported as symbols throughout the whole kernel.

Defined in: kernel/sched.c

`unregister_*dev()`

```
int unregister_chrdev(unsigned int major, const char *name)
int unregister_blkdev(unsigned int major, const char *name)
```

```
#include <linux/fs.h>
#include <linux/errno.h>
```

Removes the registration for a device device with the kernel, letting the kernel give the major number to some other device. Takes two arguments:

`major`	Major number of device being registered. Must be the same number given to `register_*dev()`.
`name`	Unique string identifying driver. Must be the same number given to `register_*dev()`.

Returns: `-EINVAL` if major is ≥ `MAX_CHRDEV` or `MAX_BLKDEV` (defined in `<linux/fs.h>`), for character or block devices, respectively, or if there have not been file operations registered for major device `major`, or if `name` is not the same name that the device was registered with.
0 on success.

Defined in: fs/devices.c
See also: `register_*dev()`

wake_up*() void wake_up(struct wait_queue ** p)
void wake_up_interruptible(struct wait_queue ** p)

#include <linux/sched.h>

Wakes up a process that has been put to sleep by the matching *sleep_on() function. wake_up() can be used to wake up tasks in a queue where the tasks may be in a TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE state, while wake_up_interruptible() will only wake up tasks in a TASK_INTERRUPTIBLE state, and will be insignificantly faster than wake_up() on queues that have only interruptible tasks. These take one argument:

q Pointer to the wait_queue structure of the process to be woken.

Note that wake_up() does not switch tasks, it only makes processes that are woken up runnable, so that the next time schedule() is called, they will be candidates to run.

Defined in: kernel/sched.c
See also: select_wait(), *sleep_on()

1.7 Writing a SCSI Device Driver

This is (with the author's explicit permission) a modified copy of the original document. If you wish to reproduce just this section, you are advised to get the original version by ftp from ftp.cs.unc.edu:/pub/faith/papers/scsi.paper.tar.gz

1.7.1 Why You Want to Write a SCSI Driver

Currently, the Linux kernel contains drivers for the following SCSI host adapters: Adaptec 1542, Adaptec 1740, Future Domain TMC-1660/TMC-1680, Seagate ST-01/ST-02, UltraStor 14F, and Western Digital WD-7000. You may want to write your own driver for an unsupported host adapter. You may also want to re-write or update one of the existing drivers.

1.7.2 What is SCSI?

The foreword to the SCSI-2 standard draft [ANS] gives a succinct definition of the Small Computer System Interface and briefly explains how SCSI-2 is related to SCSI-1 and CCS:

> The SCSI protocol is designed to provide an efficient peer-to-peer I/O bus with up to 8 devices, including one or more hosts. Data may be transferred asynchronously at rates that only depend on device implementation and cable length. Synchronous data transfers are supported at rates up to 10 mega-transfers per second. With the 32 bit wide data transfer option, data rates of up to 40 megabytes per second are possible.
>
> SCSI-2 includes command sets for magnetic and optical disks, tapes, printers, processors, CD-ROMs, scanners, medium changers, and communications devices.
>
> In 1985, when the first SCSI standard was being finalized as an American National Standard, several manufacturers approached the X3T9.2 Task Group. They wanted to increase the mandatory requirements of SCSI and to define further features for direct-access devices. Rather than delay the SCSI standard, X3T9.2 formed an ad hoc group to develop a working paper that was eventually called the Common Command Set (CCS). Many disk products were designed using this working paper in conjunction with the SCSI standard.
>
> In parallel with the development of the CCS working paper, X3T9.2 began work on an enhanced SCSI standard which was named SCSI-2. SCSI-2 included the results of the CCS working paper and extended them to all device types. It also added caching commands, performance enhancement features, and other functions that X3T9.2 deemed worthwhile. While SCSI-2 has gone well beyond the original SCSI standard (now referred to as SCSI-1), it retains a high degree of compatibility with SCSI-1 devices.

1.7.2.1 SCSI phases

The “SCSI bus” transfers data and state information between interconnected SCSI devices. A single transaction between an “initiator” and a “target” can involve up to 8 distinct “phases.” These phases are almost entirely determined by the target (e.g., the hard disk drive). The current phase can be determined from an examination of five SCSI bus signals, as shown in Table 1.1 [LXT91, p. 57].

Some controllers (notably the inexpensive Seagate controller) require direct manipulation of the SCSI bus—other controllers automatically handle these low-level details. Each of the eight phases will be described in detail.

-SEL	-BSY	-MSG	-C/D	-I/O	PHASE
HI	HI	?	?	?	BUS FREE
HI	LO	?	?	?	ARBITRATION
I	I&T	?	?	?	SELECTION
T	I&T	?	?	?	RESELECTION
HI	LO	HI	HI	HI	DATA OUT
HI	LO	HI	HI	LO	DATA IN
HI	LO	HI	LO	HI	COMMAND
HI	LO	HI	LO	LO	STATUS
HI	LO	LO	LO	HI	MESSAGE OUT
HI	LO	LO	LO	LO	MESSAGE IN

I = Initiator Asserts, T = Target Asserts, ? = HI or LO

Table 1.1: SCSI Bus Phase Determination

BUS FREE Phase

The BUS FREE phase indicates that the SCSI bus is idle and is not currently being used.

ARBITRATION Phase

The ARBITRATION phase is entered when a SCSI device attempts to gain control of the SCSI bus. Arbitration can start only if the bus was previously in the BUS FREE phase. During arbitration, the arbitrating device asserts its SCSI ID on the DATA BUS. For example, if the arbitrating device's SCSI ID is 2, then the device will assert `0x04`. If multiple devices attempt simultaneous arbitration, the device with the highest SCSI ID will win. Although ARBITRATION is optional in the SCSI-1 standard, it is a required phase in the SCSI-2 standard.

SELECTION Phase

After ARBITRATION, the arbitrating device (now called the initiator) asserts the SCSI ID of the target on the DATA BUS. The target, if present, will acknowledge the selection by raising the -BSY line. This line remains active as long as the target is connected to the initiator.

RESELECTION Phase

The SCSI protocol allows a device to disconnect from the bus while processing a request. When the device is ready, it reconnects to the host adapter. The RESELECTION phase is identical to the SELECTION phase, with the exception that it is used by the disconnected target to reconnect to the original initiator. Drivers which do not currently support RESELECTION do not allow the SCSI target to disconnect. RESELECTION should be supported by all drivers, however, so that multiple SCSI devices can simultaneously process commands. This allows dramatically increased throughput due to interleaved I/O requests.

COMMAND Phase

During this phase, 6, 10, or 12 bytes of command information are transferred from the initiator to the target.

DATA OUT and DATA IN Phases

During these phases, data are transferred between the initiator and the target. For example, the DATA OUT phase transfers data from the host adapter to the disk drive. The DATA IN phase transfers data from the disk drive to the host adapter. If the SCSI command does not require data transfer, then neither phase is entered.

STATUS Phase

This phase is entered after completion of all commands, and allows the target to send a status byte to the initiator. There are nine valid status bytes, as shown in Table 1.2 [ANS, p. 77]. Note that since bits[6] 1–5 are used for the status code (the other bits are reserved), the status byte should be masked with `0x3e` before being examined.

The meanings of the three most important status codes are outlined below:

GOOD

The operation completed successfully.

CHECK CONDITION

An error occurred. The REQUEST SENSE command should be used to find out more information about the error (see section 1.7.3).

BUSY

The device was unable to accept a command. This may occur during a self-test or shortly after power-up.

MESSAGE OUT and MESSAGE IN Phases

Additional information is transferred between the target and the initiator. This information may regard the status of an outstanding command, or

[6] Bit 0 is the least significant bit.

Value[†]	Status
0x00	GOOD
0x02	CHECK CONDITION
0x04	CONDITION MET
0x08	BUSY
0x10	INTERMEDIATE
0x14	INTERMEDIATE-CONDITION MET
0x18	RESERVATION CONFLICT
0x22	COMMAND TERMINATED
0x28	QUEUE FULL

[†] After masking with 0x3e

Table 1.2: SCSI Status Codes

may be a request for a change of protocol. Multiple MESSAGE IN and MESSAGE OUT phases may occur during a single SCSI transaction. If RESELECTION is supported, the driver must be able to correctly process the SAVE DATA POINTERS, RESTORE POINTERS, and DISCONNECT messages. Although required by the SCSI-2 standard, some devices do not automatically send a SAVE DATA POINTERS message prior to a DISCONNECT message.

1.7.3 SCSI Commands

Each SCSI command is 6, 10, or 12 bytes long. The following commands must be well understood by a SCSI driver developer.

REQUEST SENSE

Whenever a command returns a CHECK CONDITION status, the high-level Linux SCSI code automatically obtains more information about the error by executing the REQUEST SENSE. This command returns a sense key and a sense code (called the “additional sense code,” or ASC, in the SCSI-2 standard [ANS]). Some SCSI devices may also report an “additional sense code qualifier” (ASCQ). The 16 possible sense keys are described in Table 1.3. For information on the ASC and ASCQ, please refer to the SCSI standard [ANS] or to a SCSI device technical manual.

Sense Key	Description
`0x00`	NO SENSE
`0x01`	RECOVERED ERROR
`0x02`	NOT READY
`0x03`	MEDIUM ERROR
`0x04`	HARDWARE ERROR
`0x05`	ILLEGAL REQUEST
`0x06`	UNIT ATTENTION
`0x07`	DATA PROTECT
`0x08`	BLANK CHECK
`0x09`	(Vendor specific error)
`0x0a`	COPY ABORTED
`0x0b`	ABORTED COMMAND
`0x0c`	EQUAL
`0x0d`	VOLUME OVERFLOW
`0x0e`	MISCOMPARE
`0x0f`	RESERVED

Table 1.3: Sense Key Descriptions

TEST UNIT READY
This command is used to test the target's status. If the target can accept a medium-access command (e.g., a READ or a WRITE), the command returns with a GOOD status. Otherwise, the command returns with a CHECK CONDITION status and a sense key of NOT READY. This response usually indicates that the target is completing power-on self-tests.

INQUIRY This command returns the target's make, model, and device type. The high-level Linux code uses this command to differentiate among magnetic disks, optical disks, and tape drives (the high-level code currently does not support printers, processors, or juke boxes).

READ and WRITE
These commands are used to transfer data from and to the target. You should be sure your driver can support simpler commands, such as TEST UNIT READY and INQUIRY, before attempting to use the READ and WRITE commands.

1.7.4 Getting Started

The author of a low-level device driver will need to have an understanding of how interruptions are handled by the kernel. At minimum, the kernel functions that disable (`cli()`) and enable (`sti()`) interruptions should be understood. The scheduling functions (e.g., `schedule()`, `sleepon()`, and `wakeup()`) may also be needed by some drivers. A detailed explanation of these functions can be found in section 1.6.

1.7.5 Before You Begin: Gathering Tools

Before you begin to write a SCSI driver for Linux, you will need to obtain several resources.

The most important is a bootable Linux system—preferably one which boots from an IDE, RLL, or MFM hard disk. During the development of your new SCSI driver, you will rebuild the kernel and reboot your system many times. Programming errors may result in the destruction of data on your SCSI drive *and* on your non-SCSI drive. *Back up your system before you begin.*

The installed Linux system can be quite minimal: the GCC compiler distribution (including libraries and the binary utilities), an editor, and the kernel source are all you need. Additional tools like `od`, `hexdump`, and `less` will be quite helpful. All of these tools will fit on an inexpensive 20-30 MB hard disk.[7].

Documentation is essential. At minimum, you will need a technical manual for your host adapter. Since Linux is freely distributable, and since you (ideally) want to distribute your source code freely, avoid non-disclosure agreements (NDA). Most NDA's will prohibit you from releasing your source code—you might be allowed to release an object file containing your driver, but this is simply not acceptable in the Linux community at this time.

A manual that explains the SCSI standard will be helpful. Usually the technical manual for your disk drive will be sufficient, but a copy of the SCSI standard will often be helpful.[8]

Before you start, make hard copies of `hosts.h`, `scsi.h`, and one of the existing drivers in the Linux kernel. These will prove to be useful references while you write your driver.

1.7.6 The Linux SCSI Interface

The high-level SCSI interface in the Linux kernel manages all of the interaction between the kernel and the low-level SCSI device driver. Because of this layered design, a low-level SCSI driver need only provide a few basic services to the high-level code. The author of a low-level driver does not need to understand the intricacies of the kernel I/O system and, hence, can write a low-level driver in a relatively short amount of time.

Two main structures (`Scsi_Host` and `Scsi_Cmnd`) are used to communicate between the high-level code and the low-level code. The next two sections provide detailed information about these structures and the requirements of the low-level driver.

1.7.7 The `Scsi_Host` Structure

The `Scsi_Host` structure serves to describe the low-level driver to the high-level code. Usually, this description is placed in the device driver's header file in a C preprocessor definition, as shown in Figure 1.1.

The `Scsi_Host` structure is presented in Figure 1.2. Each of the fields will be explained in detail later in this section.

[7] A used 20 MB MFM hard disk and controller should cost less than US$100.

[8] The October 17, 1991, draft of the SCSI-2 standard document is available via anonymous ftp from `sunsite.unc.edu` in `/pub/Linux/development/scsi-2.tar.Z`, and is available for purchase from Global Engineering Documents (2805 McGaw, Irvine, CA 92714), (800)-854-7179 or (714)-261-1455. Please refer to document X3.131-199X. In early 1993, the manual cost US$60–70.

```
#define FDOMAIN_16X0  { "Future Domain TMC-16x0",          \
                         fdomain_16x0_detect,              \
                         fdomain_16x0_info,                \
                         fdomain_16x0_command,             \
                         fdomain_16x0_queue,               \
                         fdomain_16x0_abort,               \
                         fdomain_16x0_reset,               \
                         NULL,                             \
                         fdomain_16x0_biosparam,           \
                         1, 6, 64, 1 ,0, 0}
#endif
```

Figure 1.1: Device Driver Header File

```
typedef struct
{
  char               *name;
  int                (* detect)(int);
  const char         *(* info)(void);
  int                (* queuecommand)(Scsi_Cmnd *,
                      void (*done)(Scsi_Cmnd *));
  int                (* command)(Scsi_Cmnd *);
  int                (* abort)(Scsi_Cmnd *, int);
  int                (* reset)(void);
  int                (* slave_attach)(int, int);
  int                (* bios_param)(int, int, int []);
  int                can_queue;
  int                this_id;
  short unsigned int sg_tablesize;
  short              cmd_per_lun;
  unsigned           present:1;
  unsigned           unchecked_isa_dma:1;
} Scsi_Host;
```

Figure 1.2: The Scsi_Host Structure

1.7.7.1 Variables in the `Scsi_Host` structure

In general, the variables in the `Scsi_Host` structure are not used until after the `detect()` function (see section 1.7.7.2.1) is called. Therefore, any variables which cannot be assigned before host adapter detection should be assigned during detection. This situation might occur, for example, if a single driver provided support for several host adapters with very similar characteristics. Some of the parameters in the `Scsi_Host` structure might then depend on the specific host adapter detected.

1.7.7.1.1 `name`

`name` holds a pointer to a short description of the SCSI host adapter.

1.7.7.1.2 `can_queue`

`can_queue` holds the number of outstanding commands the host adapter can process. Unless RESELECTION is supported by the driver and the driver is interrupt-driven,[9] this variable should be set to 1.

1.7.7.1.3 `this_id`

Most host adapters have a specific SCSI ID assigned to them. This SCSI ID, usually 6 or 7, is used for RESELECTION. The `this_id` variable holds the host adapter's SCSI ID. If the host adapter does not have an assigned SCSI ID, this variable should be set to -1 (in this case, RESELECTION cannot be supported).

1.7.7.1.4 `sg_tablesize`

The high-level code supports "scatter-gather," a method of increasing SCSI throughput by combining many small SCSI requests into a few large SCSI requests. Since most SCSI disk drives are formatted with 1:1 interleave,[10] the time required to perform the SCSI ARBITRATION and SELECTION phases is longer than the rotational latency time between

[9] Some of the early Linux drivers were not interrupt driven and, consequently, had very poor performance.

[10] "1:1 interleave" means that all of the sectors in a single track appear consecutively on the disk surface.

sectors.[11] Therefore, only one SCSI request can be processed per disk revolution, resulting in a throughput of about 50 kilobytes per second. When scatter-gather is supported, however, average throughput is usually over 500 kilobytes per second.

The `sg_tablesize` variable holds the maximum allowable number of requests in the scatter-gather list. If the driver does not support scatter-gather, this variable should be set to `SG_NONE`. If the driver can support an unlimited number of grouped requests, this variable should be set to `SG_ALL`. Some drivers will use the host adapter to manage the scatter-gather list and may need to limit `sg_tablesize` to the number that the host adapter hardware supports. For example, some Adaptec host adapters require a limit of 16.

1.7.7.1.5 `cmd_per_lun`

The SCSI standard supports the notion of "linked commands." Linked commands allow several commands to be queued consecutively to a single SCSI device. The `cmd_per_lun` variable specifies the number of linked commands allowed. This variable should be set to 1 if command linking is not supported. At this time, however, the high-level SCSI code will not take advantage of this feature.

Linked commands are fundamentally different from multiple outstanding commands (as described by the `can_queue` variable). Linked commands always go to the same SCSI target and do not necessarily involve a RESELECTION phase. Further, linked commands eliminate the ARBITRATION, SELECTION, and MESSAGE OUT phases on all commands after the first one in the set. In contrast, multiple outstanding commands may be sent to an arbitrary SCSI target, and *require* the ARBITRATION, SELECTION, MESSAGE OUT, and RESELECTION phases.

1.7.7.1.6 `present`

The `present` bit is set (by the high-level code) if the host adapter is detected.

1.7.7.1.7 `unchecked_isa_dma`

Some host adapters use Direct Memory Access (DMA) to read and write blocks of data directly from or to the computer's main memory. Linux is a virtual memory operating

[11] This may be an over-simplification. On older devices, the actual command processing can be significant. Further, there is a great deal of layered overhead in the kernel: the high-level SCSI code, the buffering code, and the file-system code all contribute to poor SCSI performance.

system that can use more than 16 MB of physical memory. Unfortunately, on machines using the ISA bus[12], DMA is limited to the low 16 MB of physical memory.

If the `unchecked_isa_dma` bit is set, the high-level code will provide data buffers which are guaranteed to be in the low 16 MB of the physical address space. Drivers written for host adapters that do not use DMA should set this bit to zero. Drivers specific to EISA bus[13] machines should also set this bit to zero, since EISA bus machines allow unrestricted DMA access.

1.7.7.2 Functions in the `Scsi_Host` Structure

1.7.7.2.1 `detect()`

The `detect()` function's only argument is the "host number," an index into the `scsi_hosts` variable (an array of type `struct Scsi_Host`). The `detect()` function should return a non-zero value if the host adapter is detected, and should return zero otherwise.

Host adapter detection must be done carefully. Usually the process begins by looking in the ROM area for the "BIOS signature" of the host adapter. On PC/AT-compatible computers, the use of the address space between `0xc0000` and `0xfffff` is fairly well defined. For example, the video BIOS on most machines starts at `0xc0000` and the hard disk BIOS, if present, starts at `0xc8000`. When a PC/AT-compatible computer boots, every 2-kilobyte block from `0xc0000` to `0xf8000` is examined for the 2-byte signature (`0x55aa`) which indicates that a valid BIOS extension is present [Nor85].

The BIOS signature usually consists of a series of bytes that uniquely identifies the BIOS. For example, one Future Domain BIOS signature is the string

```
FUTURE DOMAIN CORP. (C) 1986-1990 1800-V2.07/28/89
```

found exactly five bytes from the start of the BIOS block.

After the BIOS signature is found, it is safe to test for the presence of a functioning host adapter in more specific ways. Since the BIOS signatures are hard-coded in the kernel, the release of a new BIOS can cause the driver to mysteriously fail. Further, people who use the SCSI adapter exclusively for Linux may want to disable the BIOS to speed boot time.

[12] The so-called "Industry Standard Architecture" bus was introduced with the IBM PC/XT and IBM PC/AT computers.

[13] The "Extended Industry Standard Architecture" bus is a non-proprietary 32-bit bus for 386 and i486 machines.

For these reasons, if the adapter can be detected safely without examining the BIOS, then that alternative method should be used.

Usually, each host adapter has a series of I/O port addresses which are used for communications. Sometimes these addresses will be hard coded into the driver, forcing all Linux users who have this host adapter to use a specific set of I/O port addresses. Other drivers are more flexible, and find the current I/O port address by scanning all possible port addresses. Usually each host adapter will allow 3 or 4 sets of addresses, which are selectable via hardware jumpers on the host adapter card.

After the I/O port addresses are found, the host adapter can be interrogated to confirm that it is, indeed, the expected host adapter. These tests are host adapter specific, but commonly include methods to determine the BIOS base address (which can then be compared to the BIOS address found during the BIOS signature search) or to verify a unique identification number associated with the board. For MCA bus[14] machines, each type of board is given a unique identification number which no other manufacturer can use—several Future Domain host adapters, for example, also use this number as a unique identifier on ISA bus machines. Other methods of verifying the host adapter existence and function will be available to the programmer.

1.7.7.2.1.1 Requesting the IRQ

After detection, the `detect()` routine must request any needed interrupt or DMA channels from the kernel. There are 16 interrupt channels, labeled IRQ 0 through IRQ 15. The kernel provides two methods for setting up an IRQ handler: `irqaction()` and `request_irq()`.

The `request_irq()` function takes two parameters, the IRQ number and a pointer to the handler routine. It then sets up a default `sigaction` structure and calls `irqaction()`. The code[15] for the `request_irq()` function is shown in Figure 1.3. I will limit my discussion to the more general `irqaction()` function.

The declaration[16] for the `irqaction()` function is

```
int irqaction( unsigned int irq, struct sigaction *new )
```

where the first parameter, `irq`, is the number of the IRQ that is being requested, and the

[14] The "Micro-Channel Architecture" bus is IBM's proprietary 32 bit bus for 386 and i486 machines.

[15] Linux 0.99.7 kernel source code, `linux/kernel/irq.c`

[16] Linux 0.99.5 kernel source code, `linux/kernel/irq.c`

```
int request_irq( unsigned int irq, void (*handler)( int ) )
{
  struct sigaction sa;

  sa.sa_handler  = handler;
  sa.sa_flags    = 0;
  sa.sa_mask     = 0;
  sa.sa_restorer = NULL;
  return irqaction( irq, &sa );
}
```

Figure 1.3: The request_irq() Function

second parameter, new, is a structure with the definition[17] shown in Figure 1.4.

```
struct sigaction
{
  __sighandler_t sa_handler;
  sigset_t       sa_mask;
  int            sa_flags;
  void           (*sa_restorer)(void);
};
```

Figure 1.4: The sigaction Structure

In this structure, sa_handler should point to your interrupt handler routine, which should have a definition similar to the following:

```
void fdomain_16x0_intr( int irq )
```

where irq will be the number of the IRQ which caused the interrupt handler routine to be invoked.

[17] Linux 0.99.5 kernel source code, linux/include/linux/signal.h

The sa_mask variable is used as an internal flag by the irqaction() routine. Traditionally, this variable is set to zero prior to calling irqaction().

The sa_flags variable can be set to zero or to SA_INTERRUPT. If zero is selected, the interrupt handler will run with other interrupts enabled, and will return via the signal-handling return functions. This option is recommended for relatively slow IRQ's, such as those associated with the keyboard and timer interrupts. If SA_INTERRUPT is selected, the handler will be called with interrupts disabled and return will avoid the signal-handling return functions. SA_INTERRUPT selects "fast" IRQ handler invocation routines, and is recommended for interrupt driven hard disk routines. The interrupt handler should turn interrupts on as soon as possible, however, so that other interrupts can be processed.

The sa_restorer variable is not currently used, and is traditionally set to NULL.

The request_irq() and irqaction() functions will return zero if the IRQ was successfully assigned to the specified interrupt handler routine. Non-zero result codes may be interpreted as follows:

-EINVAL Either the IRQ requested was larger than 15, or a NULL pointer was passed instead of a valid pointer to the interrupt handler routine.

-EBUSY The IRQ requested has already been allocated to another interrupt handler. This situation should never occur, and is reasonable cause for a call to panic().

The kernel uses an Intel "interrupt gate" to set up IRQ handler routines requested via the irqaction() function. The Intel i486 manual [Int90, p. 9-11] explains the interrupt gate as follows:

> Interrupts using...interrupt gates...cause the TF flag [trap flag] to be cleared after its current value is saved on the stack as part of the saved contents of the EFLAGS register. In so doing, the processor prevents instruction tracing from affecting interrupt response. A subsequent IRET [interrupt return] instruction restores the TF flag to the value in the saved contents of the EFLAGS register on the stack.
>
> ... An interrupt which uses an interrupt gate clears the IF flag [interrupt-enable flag], which prevents other interrupts from interfering with the current interrupt handler. A subsequent IRET instruction restores the IF flag to the value in the saved contents of the EFLAGS register on the stack.

1.7.7.2.2 Requesting the DMA channel

Some SCSI host adapters use DMA to access large blocks of data in memory. Since the CPU does not have to deal with the individual DMA requests, data transfers are faster than CPU-mediated transfers and allow the CPU to do other useful work during a block transfer (assuming interrupts are enabled).

The host adapter will use a specific DMA channel. This DMA channel will be determined by the `detect()` function and requested from the kernel with the `request_dma()` function. This function takes the DMA channel number as its only parameter and returns zero if the DMA channel was successfully allocated. Non-zero results may be interpreted as follows:

`-EINVAL` The DMA channel number requested was larger than 7.

`-EBUSY` The requested DMA channel has already been allocated. This is a very serious situation, and will probably cause any SCSI requests to fail. It is worthy of a call to `panic()`.

1.7.7.2.3 `info()`

The `info()` function merely returns a pointer to a static area containing a brief description of the low-level driver. This description, which is similar to that pointed to by the `name` variable, will be printed at boot time.

1.7.7.2.4 `queuecommand()`

The `queuecommand()` function sets up the host adapter for processing a SCSI command and then returns. When the command is finished, the `done()` function is called with the `Scsi_Cmnd` structure pointer as a parameter. This allows the SCSI command to be executed in an interrupt-driven fashion. Before returning, the `queuecommand()` function must do several things:

1. Save the pointer to the `Scsi_Cmnd` structure.

2. Save the pointer to the `done()` function in the `scsi_done()` function pointer in the `Scsi_Cmnd` structure. See section 1.7.7.2.5 for more information.

3. Set up the special `Scsi_Cmnd` variables required by the driver. See section 1.7.8 for detailed information on the `Scsi_Cmnd` structure.

4. Start the SCSI command. For an advanced host adapter, this may be as simple as sending the command to a host adapter "mailbox." For less advanced host adapters, the ARBITRATION phase is manually started.

The `queuecommand()` function is called *only* if the `can_queue` variable (see section 1.7.7.1.2) is non-zero. Otherwise the `command()` function is used for all SCSI requests. The `queuecommand()` function should return zero on success (the current high-level SCSI code presently ignores the return value).

1.7.7.2.5 done()

The `done()` function is called after the SCSI command completes. The single parameter that this command requires is a pointer to the same `Scsi_Cmnd` structure that was previously passed to the `queuecommand()` function. Before the `done()` function is called, the `result` variable must be set correctly. The `result` variable is a 32 bit integer, each byte of which has specific meaning:

Byte 0 (LSB) This byte contains the SCSI STATUS code for the command, as described in section 1.7.2.1.

Byte 1 This byte contains the SCSI MESSAGE, as described in section 1.7.2.1.

Byte 2 This byte holds the host adapter's return code. The valid codes for this byte are given in `scsi.h` and are described below:

`DID_OK` No error.

`DID_NO_CONNECT`
SCSI SELECTION failed because there was no device at the address specified.

`DID_BUS_BUSY`
SCSI ARBITRATION failed.

`DID_TIME_OUT`
A time-out occurred for some unknown reason, probably during SELECTION or while waiting for RESELECTION.

DID_BAD_TARGET
The SCSI ID of the target was the same as the SCSI ID of the host adapter.

DID_ABORT The high-level code called the low-level abort() function (see section 1.7.7.2.7).

DID_PARITY A SCSI PARITY error was detected.

DID_ERROR An error occurred which lacks a more appropriate error code (for example, an internal host adapter error).

DID_RESET The high-level code called the low-level reset() function (see section 1.7.7.2.8).

DID_BAD_INTR
An unexpected interrupt occurred *and* there is no appropriate way to handle this interrupt.

Note that returning DID_BUS_BUSY will force the command to be retried, whereas returning DID_NO_CONNECT will abort the command.

Byte 3 (MSB)
This byte is for a high-level return code, and should be left as zero by the low-level code.

Current low-level drivers do not uniformly (or correctly) implement error reporting, so it may be better to consult scsi.c to determine exactly how errors should be reported, rather than exploring existing drivers.

1.7.7.2.6 command()

The command() function processes a SCSI command and returns when the command is finished. When the original SCSI code was written, interrupt-driven drivers were not supported. The old drivers are much less efficient (in terms of response time and latency) than the current interrupt-driven drivers, but are also much easier to write. For new drivers, this command can be replaced with a call to the queuecommand() function, as demonstrated in Figure 1.5.[18]

[18] Linux 0.99.5 kernel, linux/kernel/blk_drv/scsi/aha1542.c, written by Tommy Thorn.

```
static volatile int internal_done_flag    = 0;
static volatile int internal_done_errcode = 0;
static void         internal_done( Scsi_Cmnd *SCpnt )
{
  internal_done_errcode = SCpnt->result;
  ++internal_done_flag;
}

int aha1542_command( Scsi_Cmnd *SCpnt )
{
  aha1542_queuecommand( SCpnt, internal_done );

  while (!internal_done_flag);
  internal_done_flag = 0;
  return internal_done_errcode;
}
```

Figure 1.5: Example `command()` Function

The return value is the same as the `result` variable in the `Scsi_Cmnd` structure. Please see sections 1.7.7.2.5 and 1.7.8 for more details.

1.7.7.2.7 `abort()`

The high-level SCSI code handles all timeouts. This frees the low-level driver from having to do timing, and permits different timeout periods to be used for different devices (e.g., the timeout for a SCSI tape drive is nearly infinite, whereas the timeout for a SCSI disk drive is relatively short).

The `abort()` function is used to request that the currently outstanding SCSI command, indicated by the `Scsi_Cmnd` pointer, be aborted. After setting the `result` variable in the `Scsi_Cmnd` structure, the `abort()` function returns zero. If `code`, the second parameter to the `abort()` function, is zero, then `result` should be set to `DID_ABORT`. Otherwise, `result` shoudl be set equal to `code`. If `code` is not zero, it is usually `DID_TIME_OUT` or `DID_RESET`.

Currently, none of the low-level drivers is able to correctly abort a SCSI command. The initiator should request (by asserting the `-ATN` line) that the target enter a MESSAGE OUT phase. Then, the initiator should send an ABORT message to the target.

1.7.7.2.8 `reset()`

The `reset()` function is used to reset the SCSI bus. After a SCSI bus reset, any executing command should fail with a `DID_RESET` result code (see section 1.7.7.2.5).

Currently, none of the low-level drivers handles resets correctly. To correctly reset a SCSI command, the initiator should request (by asserting the `-ATN` line) that the target enter a MESSAGE OUT phase. Then, the initiator should send a BUS DEVICE RESET message to the target. It may also be necessary to initiate a SCSI RESET by asserting the `-RST` line, which will cause all target devices to be reset. After a reset, it may be necessary to renegotiate a synchronous communications protocol with the targets.

1.7.7.2.9 `slave_attach()`

The `slave_attach()` function is *not* currently implemented. This function would be used to negotiate synchronous communications between the host adapter and the target drive. This negotiation requires an exchange of a pair of SYNCHRONOUS DATA TRANSFER REQUEST messages between the initiator and the target. This exchange should occur under the following conditions [LXT91]:

A SCSI device that supports synchronous data transfer recognizes it has not communicated with the other SCSI device since receiving the last "hard" RESET.

A SCSI device that supports synchronous data transfer recognizes it has not communicated with the other SCSI device since receiving a BUS DEVICE RESET message.

1.7.7.2.10 `bios_param()`

Linux supports the MS-DOS[19] hard disk partitioning system. Each disk contains a "partition table" which defines how the disk is divided into logical sections. Interpretation of this partition table requires information about the size of the disk in terms of cylinders, heads, and sectors per cylinder. SCSI disks, however, hide their physical geometry and are accessed logically as a contiguous list of sectors. Therefore, in order to be compatible with MS-DOS, the SCSI host adapter will "lie" about its geometry. The physical geometry of the SCSI disk, while available, is seldom used as the "logical geometry." (The reasons for this involve archaic and arbitrary limitations imposed by MS-DOS.)

Linux needs to determine the "logical geometry" so that it can correctly modify and interpret the partition table. Unfortunately, there is no standard method for converting between physical and logical geometry. Hence, the `bios_param()` function was introduced in an attempt to provide access to the host adapter geometry information.

The `size` parameter is the size of the disk in sectors. Some host adapters use a deterministic formula based on this number to calculate the logical geometry of the drive. Other host adapters store geometry information in tables which the driver can access. To facilitate this access, the `dev` parameter contains the drive's device number. Two macros are defined in `linux/fs.h` which will help to interpret this value: `MAJOR(dev)` is the device's major number, and `MINOR(dev)` is the device's minor number. These are the same major and minor device numbers used by the standard Linux `mknod` command to create the device in the `/dev` directory. The `info` parameter points to an array of three integers that the `bios_param()` function will fill in before returning:

`info[0]`	Number of heads
`info[1]`	Number of sectors per cylinder
`info[2]`	Number of cylinders

The information in `info` is *not* the physical geometry of the drive, but only a *logical*

[19] MS-DOS is a registered trademark of Microsoft Corporation.

geometry that is identical to the *logical* geometry used by MS-DOS to access the drive. The distinction between physical and logical geometry cannot be overstressed.

1.7.8 The `Scsi_Cmnd` Structure

The `Scsi_Cmnd` structure,[20] as shown in Figure 1.6, is used by the high-level code to specify a SCSI command for execution by the low-level code. Many variables in the `Scsi_Cmnd` structure can be ignored by the low-level device driver—other variables, however, are extremely important.

1.7.8.1 Reserved Areas

1.7.8.1.1 Informative Variables

`host` is an index into the `scsi_hosts` array.

`target` stores the SCSI ID of the target of the SCSI command. This information is important if multiple outstanding commands or multiple commands per target are supported.

`cmnd` is an array of bytes which hold the actual SCSI command. These bytes should be sent to the SCSI target during the COMMAND phase. `cmnd[0]` is the SCSI command code. The `COMMAND_SIZE` macro, defined in `scsi.h`, can be used to determine the length of the current SCSI command.

`result` is used to store the result code from the SCSI request. Please see section 1.7.7.2.5 for more information about this variable. This variable *must* be correctly set before the low-level routines return.

1.7.8.1.2 The Scatter-Gather List

`use_sg` contains a count of the number of pieces in the scatter-gather chain. If `use_sg` is zero, then `request_buffer` points to the data buffer for the SCSI command, and `request_bufflen` is the length of this buffer in bytes. Otherwise, `request_buffer` points to an array of `scatterlist` structures, and `use_sg` will indicate how many such structures are in the array. The use of `request_buffer` is non-intuitive and confusing.

Each element of the `scatterlist` array contains an `address` and a `length` component. If the `unchecked_isa_dma` flag in the `Scsi_Host` structure is set to 1 (see section 1.7.7.1.7 for more information on DMA transfers), the address is guaranteed to be within the first 16 MB of physical memory. Large amounts of data will be processed by a single SCSI

[20] Linux 0.99.7 kernel, `linux/kernel/blk_drv/scsi/scsi.h`

```
typedef struct scsi_cmnd
{
  int               host;
  unsigned char     target,
                    lun,
                    index;
  struct scsi_cmnd *next,
                    *prev;

  unsigned char     cmnd[10];
  unsigned          request_bufflen;
  void              *request_buffer;

  unsigned char     data_cmnd[10];
  unsigned short    use_sg;
  unsigned short    sglist_len;
  unsigned          bufflen;
  void              *buffer;

  struct request    request;
  unsigned char     sense_buffer[16];
  int               retries;
  int               allowed;
  int               timeout_per_command,
                    timeout_total,
                    timeout;
  unsigned char     internal_timeout;
  unsigned          flags;

  void (*scsi_done)(struct scsi_cmnd *);
  void (*done)(struct scsi_cmnd *);

  Scsi_Pointer      SCp;
  unsigned char     *host_scribble;
  int               result;

} Scsi_Cmnd;
```

Figure 1.6: The Scsi_Cmnd Structure

command. The length of these data will be equal to the sum of the lengths of all the buffers pointed to by the `scatterlist` array.

1.7.8.2 Scratch Areas

Depending on the capabilities and requirements of the host adapter, the scatter-gather list can be handled in a variety of ways. To support multiple methods, several scratch areas are provided for the exclusive use of the low-level driver.

1.7.8.2.1 The `scsi_done()` Pointer

This pointer should be set to the `done()` function pointer in the `queuecommand()` function (see section 1.7.7.2.4 for more information). There are no other uses for this pointer.

1.7.8.2.2 The `host_scribble` Pointer

The high-level code supplies a pair of memory allocation functions, `scsi_malloc()` and `scsi_free()`, which are guaranteed to return memory in the first 16 MB of physical memory. This memory is, therefore, suitable for use with DMA. The amount of memory allocated per request *must* be a multiple of 512 bytes, and *must* be less than or equal to 4096 bytes. The total amount of memory available via `scsi_malloc()` is a complex function of the `Scsi_Host` structure variables `sg_tablesize`, `cmd_per_lun`, and `unchecked_isa_dma`.

The `host_scribble` pointer is available to point to a region of memory allocated with `scsi_malloc()`. The low-level SCSI driver is responsible for managing this pointer and its associated memory, and should free the area when it is no longer needed.

1.7.8.2.3 The `Scsi_Pointer` Structure

The `SCp` variable, a structure of type `Scsi_Pointer`, is described in Figure 1.7. The variables in this structure can be used in *any* way necessary in the low-level driver. Typically, `buffer` points to the current entry in the `scatterlist`, `buffers_residual` counts the number of entries remaining in the `scatterlist`, `ptr` is used as a pointer into the buffer, and `this_residual` counts the characters remaining in the transfer. Some host adapters require support of this detail of interaction—others can completely ignore this structure.

The second set of variables provide convenient locations to store SCSI status information and various pointers and flags.

```
typedef struct scsi_pointer
{
  char                 *ptr;             /* data pointer */
  int                  this_residual;    /* left in this buffer */
  struct scatterlist *buffer;            /* which buffer */
  int                  buffers_residual; /* how many buffers left */

  volatile int         Status;
  volatile int         Message;
  volatile int         have_data_in;
  volatile int         sent_command;
  volatile int         phase;
} Scsi_Pointer;
```

Figure 1.7: The `Scsi_Pointer` Structure

1.8 Acknowledgements

Thanks to Drew Eckhardt, Michael K. Johnson, Karin Boes, Devesh Bhatnagar, and Doug Hoffman for reading early versions of this paper and for providing many helpful comments. Special thanks to my official COMP-291 (Professional Writing in Computer Science) "readers," Professors Peter Calingaert and Raj Kumar Singh.

1.9 Network Device Drivers

[I have not written this section because I don't know anything about it. I would appreciate help with this.

Note that Donald Becker has written an excellent skeleton device driver which is a very good start towards writing a network device driver.]

Chapter 2

The /proc filesystem

The proc filesystem is an interface to several kernel data structures which behaves remarkably like a filesystem. Instead of having to read /dev/kmem and have some way of knowing where things are,[1] all an application has to do is read files and directories in /proc. This way, all the addresses of the kernel data structures are compiled into the proc filesystem at kernel compile time, and programs which use the /proc interface need not be recompiled or updated when the kernel is recompiled. It is possible to mount the proc filesystem somewhere other than /proc, but that destroys the nice predictablity of the proc filesystem, so we will conveniently ignore that option. The information should somewhat resemble Linux 1.0.

2.1 `/proc` Directories and Files

[**This section should be severly cut, and the full version put in the LPG when that is available. In the mean time, better here than nowhere.**]

In /proc, there is a subdirectory for every running process, named by the number of the process's `pid`. These directories will be explained below. There are also several other files and directories. These are:

`self` This refers to the process accessing the proc filesystem, and is identical to the directory named by the process id of the process doing the look-up.

`kmsg` This file can be used instead of the `syslog()` system call to log kernel messages. A process must have superuser priviledges to read this file, and only one process should read this file. This file should **not** be read if a syslog

[1] Usually a file called a namelist file, often /etc/psdatabase.

process is running which uses the `syslog()` system call facility to log kernel messages.

`loadavg` This file gives an output like this:

```
0.13 0.14 0.05
```

These numbers are the numbers normally given by `uptime` and other commands as the load average—they are the average number of processes trying to run at once in the last minute, in the last five minutes and in the last fifteen minutes, more or less.

`meminfo` This file is a condensed version of the output from the `free` program. Its output looks like this:

```
         total:   used:    free:   shared:  buffers:
Mem:   7528448  7344128   184320  2637824  1949696
Swap:  8024064  1474560  6549504
```

Notice that the numbers are in bytes, not KB. Linus wrote a version of free which reads this file and can return either bytes (`-b`) or KB (`-k`, the default). This is included with the `procps` package at `tsx-11.mit.edu` and other places. Also notice that there is **not** a seperate entry for each swap file: the `Swap:` line sumarizes all the swap space available to the kernel.

`uptime` This file contains two things: the time that the system has been up, and the amount of time it has spent in the idle process. Both numbers are given as decimal quantities, in seconds and hundreths of a second. The two decimal digits of precision are not guaranteed on all architectures, but are currently accurate on all working implementations of Linux, due to the convenient 100 Hz clock. This file looks like this:

```
604.33 205.45
```

In this case, the system has been running for 604.33 seconds, and of that time, 205.45 seconds have been spent in the idle task.

`kcore` This is a file which represents the physical memory in the current system, in the same format as a ‘core file’. This can be used with a debugger to examine variables in the kernel. The total length of the file is the physical memory plus a 4KB header to make it look like a core file.

`stat` The `stat` file return various statistics about the system in ascii format. An example of `stat` file is the following:

```
cpu  5470 0 3764 193792
disk 0 0 0 0
page 11584 937
swap 255 618
intr 239978
ctxt 20932
btime 767808289
```

The meaning of the lines being

`cpu` The four numbers represent the number of jiffies the system spent in user mode, in user mode with low priority (nice), in system mode, and in the idle task. The last value should be 100 times the second entry in the `uptime` file.

`disk` The four `dk_drive` entries in the `kernel_stat` structure are currently unused.

`page` This is the number of pages the system brought in from the disk and out to the disk.

`swap` Is the number of swap pages the system brought in and out.

`intr` The number of interrupts received from system boot. [**The format of this line has changed in more recent kernels.**]

`ctxt` The number of context switches the system underwent.

`btime` The boot time, in seconds since the epoch.

`modules` This return the list of kernel modules, in ascii form. The format is not well defined at this point, as it has changed from version to version. This will stabilize with later versions of Linux as the modules interface stabilizes.

`malloc` This file is present only if `CONFIG_DEBUG_MALLOC` was defined during kernel compilation.

`version` This file contains a string identifying the version of Linux that is currently running. An example is:

```
Linux version 1.1.40 (johnsonm@nigel) (gcc version 2.5.8) #3 Sat Aug 6 14:2
```

Note that this contains the version of Linux, the username and hostname of the user that compiled it, the version of gcc, the "iteration" of the compilation (each new compile will increase the number), and the output of the 'date' command as of the start of the compilation.

`net` This is a directory containing three files, all of which give the status of some part of the Linux networking layer. These files contain binary structures, and are therefore not readable with cat. However, the standard netstat suite uses these files. The binary structures read from these files are defined in `<linux/if*.h>` The files are:

`unix` **[I do not yet have details on the `unix` interface. These details will be added later.]**

`arp` **[I do not yet have details on the `arp` interface. These details will be added later.]**

`route` **[I do not yet have details on the `route` interface. These details will be added later.]**

`dev` **[I do not yet have details on the `dev` interface. These details will be added later.]**

`raw` **[I do not yet have details on the `raw` interface. These details will be added later.]**

`tcp` **[I do not yet have details on the `tcp` interface. These details will be added later.]**

`udp` **[I do not yet have details on the `udp` interface. These details will be added later.]**

Each of the process subdirectories (those with numerical names and the self directory) have several files and subdirectories, as well. The files are:

`cmdline` This holds the complete command line for the process, **unless the whole process has been swapped out,** or unless the process is a zombie. In either of these later cases, there is nothing in this file: i.e. a read on this file will return as having read 0 characters. This file is null-terminated, but **not** newline-terminated.

`cwd` A link to the current working directory of that process. To find out the cwd of process 20, say, you can do this:

```
(cd /proc/20/cwd; pwd)
```

`environ` This file contains the environment for the process. There are no newlines in this file: the entries are seperated by null characters, and there is a null character at the end. Thus, to print out the environment of process 10, you would do:

```
cat /proc/10/environ | tr "\000" "\n"
```

This file is also null-terminated and not newline terminated..

`exe` This is a link to the executable. You can type

```
/proc/10/exe
```

to run another copy of whatever process 10 is.

`fd` This is a subdirectory containing one entry for each file which the process has open, named by its file descripter, and which is a link to the actual file. Programs that will take a filename, but will not take the standard input, and which write to a file, but will not send their output to standard output, can be effectively foiled this way, assuming that `-i` is the flag designating an input file and `-o` is the flag designating an output file:

```
... | foobar -i /proc/self/fd/0 -o /proc/self/fd/1 | ...
```

Voilá. Instant filter! Note that this will not work for programs that seek on their files, as the files in the `fd` directory are not seekable.

`maps` This is a file which contains a listing of all the memory mappings that the process is using. The shared libraries are mapped in this way, so there should be one entry for each shared library in use, and some processes use memory maps for other purposes as well. Here is an example:

```
00000000-00013000 r-xs 00000400 03:03 12164
00013000-00014000 rwxp 00013400 03:03 12164
00014000-0001c000 rwxp 00000000 00:00 0
bffff000-c0000000 rwxp 00000000 00:00 0
```

The first field is a number defining the start of the mapped range.
the second field is a number defining the end of the mapped range.
The third field gives the flags:
`r` means readable, `-` means not.
`w` means writeable, `-` means not.
`x` means executable, `-` means not.

`s` means shared, `p` means private.
The fourth field is the offset at which it is mapped.
The fifth field indicates the *major:minor* device number of the file being mapped.
The sixth field indicates the inode number of the file being mapped.

`mem` This is **not** the same as the mem (1,1) device, despite the fact that it has the same device numbers. The /dev/mem device is the physical memory before any address translation is done, but the mem file here is the memory of the process that accesses it. This cannot be `mmap()`ed currently, and will not be until a general `mmap()` is added to the kernel.

`root` This is a pointer to the root directory of the process. This is useful for programs that call `chroot()`, such as ftpd.

`stat` This file contains a lot of status information about the process. The fields, in order, with their proper `scanf()` format specifiers, are:

pid `%d` The process id.

comm `(%s)` The filename of the executable, in parentheses. This is visible whether or not the executable is swapped out.

state `%c` One character from the string “RSDZT” where R is running, S is sleeping in an interruptable wait, D is sleeping in an uninterruptable wait or swapping, Z is zombie, and T is traced or stopped (on a signal).

ppid `%d` The pid of the parent.

pgrp `%d` The pgrp of the process.

session `%d` The session id of the process.

tty `%d` The tty the process uses.

tpgid `%d` The pgrp of the process which currently owns the tty that the process is connected to.

flags `%u` The flags of the process. Currently, every flag has the math bit set, because crt0.s checks for math emulation, so this is not included in the output. This is probably a bug, as not *every* process is a compiled c program. The math bit should be a decimal 4, and the traced bit is decimal 10.

min_flt %u The number of minor faults the process has made, those which have not required loading a memory page from disk.

cmin_flt %u The number of minor faults that the process and its children have made.

maj_flt %u The number of major faults the process has made, those which have required loading a memory page from disk.

cmaj_flt %u The number of major faults that the process and its children have made.

utime %d The number of jiffies that this process has been scheduled in user mode.

stime %d The number of jiffies that this process has been scheduled in kernel mode.

cutime %d The number of jiffies that this proces and its children have been scheduled in user mode.

cstime %d The number of jiffies that this proces and its children have been scheduled in kernel mode.

counter %d The current maximum size in jiffies of the process's next timeslice, of what is currently left of its current timeslice, if it is the currently running process.

priority %d The standard UN⋆X nice value, plus fifteen. The value is never negative in the kernel.

timeout %u The time in jiffies of the process's next timeout.

it_real_value %u

The time in jiffies before the interval timer mechanism causes a SIGALRM to be sent to the process.

start_time %d Time the process started in jiffies after system boot.

vsize %u Virtual memory size

rss %u Resident Set Size: number of pages the process has in **real** memory, minus 3 for administrative purposes. This is just the pages which count towards text, data, or stack space. This does **not** include pages which have not been demand-loaded in, or which are swapped out.

rlim %u Current limit on the size of the process, 2GB by default.

start_code %u The address above which program text can run.

end_code %u The address below which program text can run.

start_stack %u

The address of the start of the stack.

kstk_esp %u The current value of esp (32 bit stack pointer), as found in the kernel stack page for the process.

kstk_eip %u The current value of eip (32 bit instruction pointer), as found in the kernel stack page for the process.

signal %d The bitmap of pending signals (usually 0).

blocked %d The bitmap of blocked signals (usually 0, 2 for shells).

sigignore %d The bitmap of ignored signals.

sigcatch %d The bitmap of catched signals.

wchan %u This is the "channel" in which the process is waiting. This is the address of a system call, and can be looked up in a namelist if you need a textual name.

statm This file contains special status information that takes a bit longer to cook than the information in stat, and is needed rarely enough that it has been relegated to a seperate file. For each field in this file, the proc filesystem has to look at each of the 0x300 entries in the page directory, and count what they are doing. Here is a description of these fields:

size %d The total number of pages that the process has mapped in the virtual memory space, whether they are in physical memory or not.

resident %d The total number of pages that the process has in physical memory. This should equal the rss field from the stat file, but is calculated rather than read from the process structure.

shared `%d` The total number of pages that the process has that are shared with at least one other process.

trs `%d` Text Resident Size: the total number of text (code) pages belonging to the process that are present in physical memory. Does **not** include shared library pages.

lrs `%d` Library Resident Size: the total number of library pages used by the process that are present in physical memory.

drs `%d` Data Resident Size: the total number of data pages belonging to the process that are present in physical memory. Include dirty library pages and stack.

dt `%d` The number of library pages which have been accessed (i.e., are dirty).

2.2 Structure of the /proc filesystem

The proc filesystem is rather interesting, because none of the files exist in any real directory structure. Rather, the proper vfs structures are filled in with functions which do gigantic case statements, and in the case of reading a file, get a page, fill it in, and put the result in user memory space.

One of the most interesting parts of the proc filesystem is the way that the individual process directories are implemented. Essentially, every process directory has the inode number of its PID shifted left 16 bits into a 32 bit number greater than `0x0000ffff`. Within the process directories, inode numbers are reused, because the upper 16 bits of the inode number have been masked off after choosing the right directory.

Another interesting feature is that unlike in a “real” filesystem, where there is one `file_operations` structure for the whole filesystem, as file lookup is done, different `file_operations` structures are assigned to the `f_ops` member of the file structure passed to those functions, dynamically changing which functions will be called for directory lookup and file reading.

[Expand on this section later — right now it is mostly here to remind me to finish it. . .]

2.3 Programming the /proc filesystem

◇ **Note:** the code fragments in this section won't match the sources for your own kernel exactly, as the `/proc` filesystem has been expanded since this was originally written, and is being expanding still more. For instance, the `root_dir` structure has nearly doubled in size from the one quoted here below.

Unlike in most filesystems, not all inode numbers in the proc filesystem are unique. Some files are declared in structures like

```
static struct proc_dir_entry root_dir[] = {
        { 1,1,"." },
        { 1,2,".." },
        { 2,7,"loadavg" },
        { 3,6,"uptime" },
        { 4,7,"meminfo" },
        { 5,4,"kmsg" },
        { 6,7,"version" },
        { 7,4,"self" }  /* will change inode # */
        { 8,4,"net" }
};
```

and some files are dynamically created as the filesystem is read. All the process directories (those with numerical names and `self`) essentially have inode numbers that are the pid shifted left 16 bits, but the files within those directories re-use low (1–10 or so) inode numbers, which are added at runtime to the pid of the process. This is done in inode.c by careful re-assignment of `inode_operation` structures.

Most of the short read-only files in the root directory and in each process subdirectory one use a simplified interface provided by the `array_inode_operations` structure, within array.c.

Other directories, such as /proc/net/, have their own inode numbers. For instance, the net directory itself has inode number 8. The files within that directory use inode numbers from the range 128–160, and those are uniquely identified in inode.c and the files given the proper permissions when looked up and read.

Adding a file is relatively simple, and is left as an exersize for the reader. Adding a new directory is a little bit harder. Assuming that it is not a dynamically allocated directory like the process directories, here are the steps:[2]

[2] Unless you are making a subdirectory of the replicating, dynamically allocated process directory, you would have to create a new filesystem type, similar to the proc filesystem in design. Subdirectories of the process directories are supported by the mechanism which dynamically creates the process

1. Choose a **unique** range of inode numbers, giving yourself a reasonable amount of room for expansion. Then, right before the line

   ```
   if (!pid) { /* not a process directory but in /proc/ */
   ```

   add a section that looks like this:

   ```
   if ((ino >= 128) && (ino <= 160)) { /* files withing /proc/net */
         inode->i_mode = S_IFREG | 0444;
         inode->i_op = &proc_net_inode_operations;
         return;
   }
   ```

   but modify it to to do what you want. For instance, perhaps you have a range of 200–256, and some files, inodes 200, 201, and 202, and some directories, which are inodes 204 and 205. You also have a file that is readable only by root, inode 206.

   Your example might look like this:

   ```
   if ((ino >= 200) && (ino <= 256)) { /* files withing /proc/foo */
         switch (ino) {
           case 204:
           case 205:
                inode->i_mode = S_IFDIR | 0555;
                inode->i_op = &proc_foo_inode_operations;
                break;
           case 206:
                inode->i_mode = S_IFREG | 0400;
                inode->i_op = &proc_foo_inode_operations;
                break;
           default:
                inode->i_mode = S_IFREG | 0444;
                inode->i_op = &proc_foo_inode_operations;
                break;
         }
         return;
   }
   ```

2. Find the definition of the files. If your files will go in a subdirectory of /proc, for instance, you will look in root.c, and find the following:

   ```
   static struct proc_dir_entry root_dir[] = {
        { 1,1,"." },
   ```

directories. I suggest going through this explanation of how to add a non-dynamically-allocated directory, understand it, and then read the code for the process subdirectories, if you wish to add subdirectories to the process subdirectories.

```
          { 1,2,".." },
          { 2,7,"loadavg" },
          { 3,6,"uptime" },
          { 4,7,"meminfo" },
          { 5,4,"kmsg" },
          { 6,7,"version" },
          { 7,4,"self" },  /* will change inode # */
          { 8,4,"net" }
     };
```

You will then add a new file to this structure, like this, using the next available inode number:

```
     [...]
          { 6,7,"version" },
          { 7,4,"self" },  /* will change inode # */
          { 8,4,"net" },
          { 9,3,"foo" }
     };
```

You will then have to provide for this new directory in inode.c, so:

```
          if (!pid) { /* not a process directory but in /proc/ */
               inode->i_mode = S_IFREG | 0444;
               inode->i_op = &proc_array_inode_operations;
               switch (ino)
                 case 5:
                      inode->i_op = &proc_kmsg_inode_operations;
                      break;
                 case 8: /* for the net directory */
                      inode->i_mode = S_IFDIR | 0555;
                      inode->i_op = &proc_net_inode_operations;
                      break;
                 default:
                      break;
                      return;
          }
```

becomes

```
          if (!pid) { /* not a process directory but in /proc/ */
               inode->i_mode = S_IFREG | 0444;
               inode->i_op = &proc_array_inode_operations;
               switch (ino)
                 case 5:
                      inode->i_op = &proc_kmsg_inode_operations;
```

```
                                break;
                        case 8: /* for the net directory */
                                inode->i_mode = S_IFDIR | 0555;
                                inode->i_op = &proc_net_inode_operations;
                                break;
                        case 9: /* for the foo directory */
                                inode->i_mode = S_IFDIR | 0555;
                                inode->i_op = &proc_foo_inode_operations;
                                break;
                        default:
                                break;
                                return;
        }
```

3. You now have to provide for the contents of the files within the foo directory. Make a file called proc/foo.c, following the following model:[3] **[The code in `proc_lookupfoo()` and `proc_readfoo()` should be abstracted, as the functionality is used in more than one place.]**

```
/*
 *  linux/fs/proc/foo.c
 *
 *  Copyright (C) 1993 Linus Torvalds, Michael K. Johnson, and Your N. Here
 *
 *  proc foo directory handling functions
 *
 *  inode numbers 200 - 256 are reserved for this directory
 *  (/proc/foo/ and its subdirectories)
 */

#include <asm/segment.h>
#include <linux/errno.h>
#include <linux/sched.h>
#include <linux/proc_fs.h>
#include <linux/stat.h>

static int proc_readfoo(struct inode *, struct file *, struct dirent *, int);
static int proc_lookupfoo(struct inode *,const char *,int,struct inode **);
static int proc_read(struct inode * inode, struct file * file,
                     char * buf, int count);

static struct file_operations proc_foo_operations = {
```

[3] This file is availabe as file proc/foo.c in the *The Linux Kernel Hackers' Guide* source mentioned on the copyright page.

```
        NULL,                  /* lseek - default */
        proc_read,             /* read */
        NULL,                  /* write - bad */
        proc_readfoo,          /* readdir */
        NULL,                  /* select - default */
        NULL,                  /* ioctl - default */
        NULL,                  /* mmap */
        NULL,                  /* no special open code */
        NULL                   /* no special release code */
};

/*
 * proc directories can do almost nothing..
 */
struct inode_operations proc_foo_inode_operations = {
        &proc_foo_operations,  /* default foo directory file-ops */
        NULL,                  /* create */
        proc_lookupfoo,        /* lookup */
        NULL,                  /* link */
        NULL,                  /* unlink */
        NULL,                  /* symlink */
        NULL,                  /* mkdir */
        NULL,                  /* rmdir */
        NULL,                  /* mknod */
        NULL,                  /* rename */
        NULL,                  /* readlink */
        NULL,                  /* follow_link */
        NULL,                  /* bmap */
        NULL,                  /* truncate */
        NULL                   /* permission */
};

static struct proc_dir_entry foo_dir[] = {
        { 1,2,".." },
        { 9,1,"." },
        { 200,3,"bar" },
        { 201,4,"suds" },
        { 202,5,"xyzzy" },
        { 203,3,"baz" },
        { 204,4,"dir1" },
        { 205,4,"dir2" },
        { 206,8,"rootfile" }
};
```

```
#define NR_FOO_DIRENTRY ((sizeof (foo_dir))/(sizeof (foo_dir[0])))

unsigned int get_bar(char * buffer);
unsigned int get_suds(char * buffer);
unsigned int get_xyzzy(char * buffer);
unsigned int get_baz(char * buffer);
unsigned int get_rootfile(char * buffer);


static int proc_read(struct inode * inode, struct file * file,
                     char * buf, int count)
{
        char * page;
        int length;
        int end;
        unsigned int ino;

        if (count < 0)
                return -EINVAL;
        page = (char *) get_free_page(GFP_KERNEL);
        if (!page)
                return -ENOMEM;
        ino = inode->i_ino;
        switch (ino) {
                case 200:
                        length = get_bar(page);
                        break;
                case 201:
                        length = get_suds(page);
                        break;
                case 202:
                        length = get_xyzzy(page);
                        break;
                case 203:
                        length = get_baz(page);
                        break;
                case 206:
                        length = get_rootfile(page);
                        break;
                default:
                        free_page((unsigned long) page);
                        return -EBADF;
        }
        if (file->f_pos >= length) {
                free_page((unsigned long) page);
                return 0;
        }
        if (count + file->f_pos > length)
                count = length - file->f_pos;
        end = count + file->f_pos;
        memcpy_tofs(buf, page + file->f_pos, count);
        free_page((unsigned long) page);
        file->f_pos = end;
        return count;

}

static int proc_lookupfoo(struct inode * dir,const char * name, int len,
        struct inode ** result)
{
        unsigned int pid, ino;
        int i;

        *result = NULL;
        if (!dir)
                return -ENOENT;
        if (!S_ISDIR(dir->i_mode)) {
                iput(dir);
                return -ENOENT;
        }
        ino = dir->i_ino;
        i = NR_FOO_DIRENTRY;
        while (i-- > 0 && !proc_match(len,name,foo_dir+i))
                /* nothing */;
        if (i < 0) {
                iput(dir);
                return -ENOENT;
        }
        if (!(*result = iget(dir->i_sb,ino))) {
                iput(dir);
                return -ENOENT;
        }
        iput(dir);
        return 0;
}

static int proc_readfoo(struct inode * inode, struct file * filp,
        struct dirent * dirent, int count)
```

```
{
        struct proc_dir_entry * de;
        unsigned int pid, ino;
        int i,j;

        if (!inode || !S_ISDIR(inode->i_mode))
                return -EBADF;
        ino = inode->i_ino;
        if (((unsigned) filp->f_pos) < NR_FOO_DIRENTRY) {
                de = foo_dir + filp->f_pos;
                filp->f_pos++;
                i = de->namelen;
                ino = de->low_ino;
                put_fs_long(ino, &dirent->d_ino);
                put_fs_word(i,&dirent->d_reclen);
                put_fs_byte(0,i+dirent->d_name);
                j = i;
                while (i--)
                        put_fs_byte(de->name[i], i+dirent->d_name);
                return j;
        }
        return 0;
}


unsigned int get_foo(char * buffer)
{

/* code to find everything goes here */

        return sprintf(buffer, "format string", variables);
}


unsigned int get_suds(char * buffer)
{

/* code to find everything goes here */

        return sprintf(buffer, "format string", variables);
}


unsigned int get_xyzzy(char * buffer)
```

```
{

/* code to find everything goes here */

        return sprintf(buffer, "format string", variables);
}


unsigned int get_baz(char * buffer)
{

/* code to find everything goes here */

        return sprintf(buffer, "format string", variables);
}


unsigned int get_rootfile(char * buffer)
{

/* code to find everything goes here */

        return sprintf(buffer, "format string", variables);
}
```

4. Filling in the directories dir1 and dir2 is left as an excersize. In most cases, such directories will not be needed. However, if they are, the steps presented here may be applied recursively to add files to a directory at another level. Notice that I saved a range of 200–256 for /proc/foo/ and all its subdirectories, so there are plenty of unused inode numbers in that range for your files in dir1 and dir2. I suggest reserving a range for each directory, in case you need to expand. Also, I suggest keeping all the extra data and functions in foo.c, rather than making yet another file, unless the files in the dir1 and dir2 directories are significantly different in concept than foo.

5. Make the appropriate changes to fs/proc/Makefile. This is also left as an excersize for the reader.

[Please note: I have made changes similar to these (I wrote the /proc/net/ support). However, this has been written from memory, and may be unintentionally incomplete. If you notice any inadequacies, please explain them to me in as complete detail as possible. My email address is `johnsonm@sunsite.unc.edu`**]**

Chapter 3

The Linux scheduler

[This is still pretty weak, but I think that I have removed most or all of the inaccuracies that were in previous versions. Jim Wisner appears to have dropped from the face of the net, so I have not been able to get his help at making this chapter more meaningful. If anyone has a copy of the paper he wrote on the scheduler, please get in touch with me, as he promised me a copy, and I'd at least like to see what he had to say about the scheduler.]

[I'm not going to spend any further time on this until the new scheduler is added to Linux. The current one doesn't handle lots of tasks at once very well, and some day a new one will be put in.]

The scheduler is a function, `schedule()`, which is called at various times to determine whether or not to switch tasks, and if so, which task to switch to. The scheduler works in concert with the function `do_timer()`, which is called 100 times per second (on Linux/x86), on each system timer interrupt, and with the system call handler part `ret_from_sys_call()` which is called on the return from system calls.

When
a system call completes, or when a "slow" interrupt completes, `ret_from_sys_call()` is called. It does a bit of work, but all we care about are two lines:

```
cmpl $0,_need_resched
jne reschedule
```

These lines check the `need_resched` flag, and if it is set, `schedule()` is called, choosing a new process, and then after `schedule()` has chosen the new process, a little more magic in `ret_from_sys_call()` restores the environment for the chosen process (which may well be, and usually is, the process which is already running), and returns to user space. Returning to user space causes the new process which has been selected to run to be returned to.

In `sched_init()` in kernel/sched.c, `request_irq()` is used to get the timer interrupt. `request_irq()` sets up housekeeping before and after interrupts are serviced, as seen in `<asm/irq.h>`. However, interrupts that are serviced often and that must be serviced quickly, such as serial interrupts, do *not* call `ret_from_sys_call()` when done and do as little as possible, to keep the overhead down. In particular, they only save the registers that C would clobber, and assume that if the handler is going to use any others, the handler will deal with that. These "fast" interrupt handlers must be installed with the `irqaction()` function described in section 1.6.

The Linux scheduler is significantly different from the schedulers in other unices, especially in its treatment of 'nice level' priorities. Instead of scheduling processes with higher priority first, Linux does round-robin scheduling, but lets higher priority processes run both sooner and longer. The standard UN⋆X scheduler instead uses queues of processes. Most implementations use two priority queues; a standard queue and a "real time" queue. Essentially, all processes on the "real time" queue get executed before processes on the standard queue, if they are not blocked, and within each queue, higher nice-level processes run before lower ones. The Linux scheduler gives much better interactive performance at the expense of some "throughput."

3.1 The code

Here is a copy of the relevant code from /usr/src/linux/kernel/sched.c, annotated and abridged.

```
void schedule(void)
{
        int i,next,c;
        struct task_struct ** p;

/* check alarm, wake up any interruptible tasks that have got a signal */

        need_resched = 0;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
```

The process table is an array of pointers to `struct task_struct` structures. See /usr/include/linux/sched.h for the definition of this structure.

```
                if (!*p || ((*p)->state != TASK_INTERRUPTIBLE))
                        continue;
                if ((*p)->timeout && (*p)->timeout < jiffies) {
```

If a process has a timeout and has reached it, then `jiffies` (the number of 100th's of a second since system start) will have passed `timeout`. `timeout` was originally set as `jiffies +` *desired_timeout*.

```
                        (*p)->timeout = 0;
                        (*p)->state = TASK_RUNNING;
                } else if ((*p)->signal & ~(*p)->blocked)
```

If the process has been sent a signal, and is no longer blocked, then let the process be allowed to run again, when its turn comes.

```
                        (*p)->state = TASK_RUNNING;
        }
```

At this point, all runnable processes have been flagged as runnable, and we are ready to choose one to run, by running through the process table. What we are looking for is the process with the largest counter. The counter for each runnable process is incremented each time the scheduler is called by an amount that is weighted by the priority, which is the kernel version of the 'nice' value. (It differs in that the priority is never negative.)

```
/* this is the scheduler proper: */

        while (1) {
                c = -1;
                next = 0;
                i = NR_TASKS;
                p = &task[NR_TASKS];
                while (--i) {
                        if (!*--p)
```

If there is no process in this slot then don't bother…

```
                                continue;
                        if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                                c = (*p)->counter, next = i;
```

If the counter is higher than any previous counter, then make the process the next process, unless, of course, an even higher one is encountered later in the loop.

```
                }
                if (c)
                        break;
                for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                        if (*p)
                                (*p)->counter = ((*p)->counter >> 1) +
                                                (*p)->priority;
```

Here is where the counter is set. It is first divided by 2, and then the priority is added. Note that this happens only if no process has been found to switch to, because of the `break;` line.

```
        }
        sti();
        switch_to(next);
}
```

`sti()` enables interrupts again, and `switch_to()` (defined in linux/sched.h) sets things up so that when we return to `ret_to_sys_call()`, we will return from `ret_to_sys_call()` into the *new* process.

I have truncated `do_timer()` extremely, only showing the pieces that relate specifically to `schedule()`. For information on the rest, see the appropriate section. For instance, for commentary on the `itimer` mechanism see the section on `itimers`. [**I suppose I need to *write* that section now... I will need to put a reference here to that section.**] I have specifically left out all the accounting stuff, all the timer stuff, and the floppy timer.

```
static void do_timer(struct pt_regs * regs)
{
        unsigned long mask;
        struct timer_struct *tp = timer_table+0;
        struct task_struct ** task_p;

        jiffies++;
```

Here is where `jiffies` is incremented. This is all-important to the rest of the kernel, because all time calculations (except for timed delay loops) are based on this variable.

```
        if (current == task[0] || (--current->counter)<=0) {
                current->counter=0;
                need_resched = 1;
                }
        }
```

Don't let task 0 run if anything else can run, because task 0 doesn't do anything. If task 0 is running, the machine is idle, but don't let it be idle if anything else is happening, so run schedudule as soon as possible. Set the `need_resched` flag if necessary so that schedule gets called again as soon as possible.

Chapter 4

How System Calls Work

[This needs to be a little re-worked and expanded upon, but I am waiting to see if the iBCS stuff makes any impact on it as I write other stuff.]

This section covers first the mechanisms provided by the 386 for handling system calls, and then shows how Linux uses those mechanisms. This is not a reference to the individual system calls: There are very many of them, new ones are added occasionally, and they are documented in man pages that should be on your Linux system.

[Ideally, this chapter should be part of another section, I think. Maybe, however, it should just be expanded. I think it belongs somewhere near the chapter on how to write a device driver, because it explains how to write a system call.]

4.1 What Does the 386 Provide?

The 386 recognizes two event classes: exceptions and interrupts. Both cause a forced context switch to new a procedure or task. Interrupts can occur at unexpected times during the execution of a program and are used to respond to signals from hardware. Exceptions are caused by the execution of instructions.

Two sources of interrupts are recognized by the 386: Maskable interrupts and Nonmaskable interrupts. Two sources of exceptions are recognized by the 386: Processor detected exceptions and programmed exceptions.

Each interrupt or exception has a number, which is referred to by the 386 literature as the vector. The NMI interrupt and the processor detected exceptions have been assigned vectors in the range 0 through 31, inclusive. The vectors for maskable interrupts are determined by the hardware. External interrupt controllers put the vector on the bus during the interrupt-acknowledge cycle. Any vector in the range 32 through 255, inclusive, can be used for maskable interrupts or programmed exceptions. See figure 4.1 for a listing of all the possible interrupts and exceptions. [**Check all this out to make sure that it is right.**]

0	divide error
1	debug exception
2	NMI interrupt
3	Breakpoint
4	INTO-detected Overflow
5	BOUND range exceeded
6	Invalid opcode
7	coprocessor not available
8	double fault
9	coprocessor segment overrun
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	reserved
16	coprocessor error
17–31	reserved
32–255	maskable interrupts

Figure 4.1: Interrupt and Exception Assignments

HIGHEST	Faults except debug faults
	Trap instructions INTO, INT n, INT 3
	Debug traps for this instruction
	Debug traps for next instruction
	NMI interrupt
LOWEST	INTR interrupt

Figure 4.2: Priority of simultaneous interrupts and exceptions

4.2 How Linux Uses Interrupts and Exceptions

Under Linux the execution of a system call is invoked by a maskable interrupt or **exception** class transfer, caused by the instruction `int 0x80`. We use vector 0x80 to transfer control to the kernel. This interrupt vector is initialized during system startup, along with other important vectors like the system clock vector.

As of version 0.99.2 of Linux, there are 116 system calls. Documentation for these can be found in the man (2) pages. When a user invokes a system call, execution flow is as follows:

- Each call is vectored through a stub in libc. Each call within the libc library is generally a `syscall`*X*`()` macro, where *X* is the number of parameters used by the actual routine. Some system calls are more complex then others because of variable length argument lists, but even these complex system calls must use the same entry point: they just have more parameter setup overhead. Examples of a complex system call include `open()` and `ioctl()`.

- Each syscall macro expands to an assembly routine which sets up the calling stack frame and calls `_system_call()` through an interrupt, via the instruction `int $0x80`

  For example, the setuid system call is coded as

  ```
  _syscall1(int,setuid,uid_t,uid);
  ```

  Which will expand to:

  ```
  _setuid:
    subl $4,%exp
    pushl %ebx
    movzwl 12(%esp),%eax
    movl %eax,4(%esp)
    movl $23,%eax
    movl 4(%esp),%ebx
    int $0x80
    movl %eax,%edx
    testl %edx,%edx
    jge L2
    negl %edx
    movl %edx,_errno
    movl $-1,%eax
    popl %ebx
    addl $4,%esp
    ret
  ```

```
L2:
  movl %edx,%eax
  popl %ebx
  addl $4,%esp
  ret
```

The macro definition for the `syscall`X`()` macros can be found in /usr/include/linux/unistd.h, and the user-space system call library code can be found in /usr/src/libc/syscall/

- At this point no system code for the call has been executed. Not until the int $0x80 is executed does the call transfer to the kernel entry point `_system_call()`. This entry point is the same for all system calls. It is responsible for saving all registers, checking to make sure a valid system call was invoked and then ultimately transfering control to the actual system call code via the offsets in the `_sys_call_table`. It is also responsible for calling `_ret_from_sys_call()` when the system call has been completed, but before returning to user space.

  Actual code for `system_call` entry point can be found in /usr/src/linux/kernel/sys_call.S Actual code for many of the system calls can be found in /usr/src/linux/kernel/sys.c, and the rest are found elsewhere. `find` is your friend.

- After the system call has executed, `_ret_from_sys_call()` is called. It checks to see if the scheduler should be run, and if so, calls it.

- Upon return from the system call, the `syscall`X`()` macro code checks for a negative return value, and if there is one, puts a positive copy of the return value in the global variable `_errno`, so that it can be accessed by code like `perror()`.

4.3 How Linux Initializes the system call vectors

The `startup_32()` code found in /usr/src/linux/boot/head.S starts everything off by calling `setup_idt()`. This routine sets up an IDT (Interrupt Descriptor Table) with 256 entries. No interrupt entry points are actually loaded by this routine, as that is done only after paging has been enabled and the kernel has been moved to 0xC0000000. An IDT has 256 entries, each 4 bytes long, for a total of 1024 bytes.

When `start_kernel()` (found in /usr/src/linux/init/main.c) is called it invokes `trap_init()` (found in /usr/src/linux/kernel/traps.c). `trap_init()` sets up the IDT via the macro `set_trap_gate()` (found in /usr/include/asm/system.h). `trap_init()` initializes the interrupt descriptor table as shown in figure 4.3.

0	divide_error
1	debug
2	nmi
3	int3
4	overflow
5	bounds
6	invalid_op
7	device_not_available
8	double_fault
9	coprocessor_segment_overrun
10	invalid_TSS
11	segment_not_present
12	stack_segment
13	general_protection
14	page_fault
15	reserved
16	coprocessor_error
17	alignment_check
18–48	reserved

Figure 4.3: Initialization of interrupts

At this point the interrupt vector for the system calls is not set up. It is initialized by `sched_init()` (found in /usr/src/linux/kernel/sched.c). A call to `set_system_gate (0x80, &system_call)` sets interrupt 0x80 to be a vector to the `system_call()` entry point.

4.4 How to Add Your Own System Calls

1. Create a directory under the /usr/src/linux/ directory to hold your code.

2. Put any include files in /usr/include/sys/ and /usr/include/linux/.

3. Add the relocatable module produced by the link of your new kernel code to the `ARCHIVES` and the subdirectory to the `SUBDIRS` lines of the top level Makefile. See fs/Makefile, target fs.o for an example.

4. Add a `#define __NR_`*xx* to unistd.h to assign a call number for your system call, where *xx*, the index, is something descriptive relating to your system call. It will be used to set up the vector through `sys_call_table` to invoke you code.

5. Add an entry point for your system call to the `sys_call_table` in sys.h. It should match the index (*xx*) that you assigned in the previous step. The `NR_syscalls` variable will be recalculated automatically.
6. Modify any kernel code in kernel/fs/mm/, etc. to take into account the environment needed to support your new code.
7. Run make from the top level to produce the new kernel incorporating your new code.

At this point, you will have to either add a syscall to your libraries, or use the proper `_syscall`*n*`()` macro in your user program for your programs to access the new system call.

The *386DX Microprocessor Programmer's Reference Manual* is a helpful reference, as is James Turley's *Advanced 80386 Programming Techniques.* See the Annotated bibliography in Appendix A.

Chapter 5

Linux Memory Management

[This chapter needs to be made much friendlier. I'd hate to remove detail, but it needs to be less daunting. Many have told me that this is a daunting chapter, and it need not be. I'll re-work it later. In the meantime, please bear with me.]

5.1 Overview

The Linux memory manager implements demand paging with a copy-on-write strategy relying on the 386's paging support. A process acquires its page tables from its parent (during a `fork()`) with the entries marked as read-only or swapped. Then, if the process tries to write to that memory space, and the page is a copy-on-write page, it is copied, and the page is marked read-write. An `exec()` results in the reading in of a page or so from the executable. The process then faults in any other pages it needs.

Each process has a page directory which means it can access 1 KB of page tables pointing to 1 MB of 4 KB pages which is 4 GB of memory. A process' page directory is initialized during a fork by `copy_page_tables()`. The idle process has its page directory initialized during the initialization sequence.

Each user process has a local descriptor table that contains a code segment and data-stack segment. These user segments extend from 0 to 3 GB (0xc0000000). In user space linear addresses[1] and logical addresses[2] are identical.

[1] In the 80386, linear address run from 0GB to 4GB. A linear address points to a particular memory location within this space. A linear address is **not** a physical address — it is a virtual address.

[2] A logical address consists of a selector and an offset. The selector points to a segment and the offset tells how far into that segment the address is located.

The kernel code and data segments are priveleged segments defined in the global descriptor table and extend from 3 GB to 4 GB. The swapper page directory (`swapper_page_dir` is set up so that logical addresses and physical addresses are identical in kernel space.

The space above 3 GB appears in a process' page directory as pointers to kernel page tables. This space is invisible to the process in user mode but the mapping becomes relevant when privileged mode is entered, for example, to handle a system call.

Supervisor mode is entered within the context of the current process so address translation occurs with respect to the process' page directory but using kernel segments. This is identically the mapping produced by using the `swapper_pg_dir` and kernel segments as both page directories use the same page tables in this space. Only `task[0]` (the idle task[3] [**This should be documented earlier in this guide**...]) uses the `swapper_pg_dir` directly.

- The user process' `segment_base` = 0x00, `page_dir` private to the process.
- user process makes a system call: `segment_base`=0xc0000000 `page_dir` = same user `page_dir`.
- `swapper_pg_dir` contains a mapping for all physical pages from 0xc0000000 to 0xc0000000 + `end_mem`, so the first 768 entries in `swapper_pg_dir` are 0's, and then there are 4 or more that point to kernel page tables.
- The user page directories have the same entries as tt swapper_pg_dir above 768. The first 768 entries map the user space.

The upshot is that whenever the linear address is above 0xc0000000 everything uses the same kernel page tables.

The user stack sits at the top of the user data segment and grows down. The kernel stack is not a pretty data structure or segment that I can point to with a "yon lies the kernel stack." A `kernel_stack_frame` (a page) is associated with each newly created process and is used whenever the kernel operates within the context of that process. Bad things would happen if the kernel stack were to grow below its current stack frame. [**Where is the kernel stack put? I know that there is one for every process, but where is it stored when it's not being used?**]

User pages can be stolen or swapped. A user page is one that is mapped below 3 GB in a user page table. This region does not contain page directories or page tables. Only dirty pages are swapped.

[3] Sometimes called the swapper task, even though it has nothing to do with swapping in the Linux implementation, for historical reasons

Minor alterations are needed in some places (tests for process memory limits comes to mind) to provide support for programmer defined segments.

5.2 Physical memory

Here is a map of physical memory before any user processes are executed. The column on the left gives the **starting** address of the item, numbers in *italics* are approximate. The column in the middle names the item(s). The column on the far right gives the relevant routine or variable name or explains the entry.

0x110000	FREE	`memory_end` or `high_memory`
	`mem_map`	`mem_init()`
	`inode_table`	`inode_init()`
	device data	`device_init()`†
0x100000	more `pg_table`s	`paging_init()`
0x0A0000	RESERVED	
0x060000	FREE	
	`low_memory_start`	
0x006000	kernel code + data	
	`floppy_track_buffer`	
	`bad_pg_table` `bad_page`	used by `page_fault_handler`s to kill processes gracefully when out of memory.
0x002000	`pg0`	the first kernel page table.
0x001000	`swapper_pg_dir`	the kernel page directory.
0x000000	null page	

†device-inits that acquire memory are(main.c): `profil_buffer`, `con_init`, `psaux_init`, `rd_init`, `scsi_dev_init`. Note that all memory not marked as FREE is RESERVED (`mem_init`). RESERVED pages belong to the kernel and are **never** freed or swapped.

5.3 A user process' view of memory

0xc0000000	The invisible kernel	reserved
	initial stack	
	room for stack growth	4 pages
0x60000000	shared libraries	
`brk`	unused	
	malloc memory	
`end_data`	uninitialized data	
`end_code`	initialized data	
0x00000000	text	

Both the code segment and data segment extend all the way from 0x00 to 3 GB. Currently the page fault handler `do_wp_page` checks to ensure that a process does not write to its code space. However, by catching the `SEGV` signal, it is possible to write to code space, causing a copy-on-write to occur. The handler `do_no_page` ensures that any new pages the process acquires belong to either the executable, a shared library, the stack, or lie within the `brk` value.

A user process can reset its `brk` value by calling `sbrk()`. This is what `malloc()` does when it needs to. The text and data portions are allocated on separate pages unless one chooses the `-N` compiler option. Shared library load addresses are currently taken from the shared image itself. The address is between 1.5 GB and 3 GB, except in special cases.

User process Memory Allocation

	swappable	shareable
a few code pages	Y	Y
a few data pages	Y	N?
stack	Y	N
`pg_dir`	N	N
code/data `page_table`	N	N
stack `page_table`	N	N
`task_struct`	N	N
`kernel_stack_frame`	N	N
shlib `page_table`	N	N
a few shlib pages	Y	Y?

[**What do the question marks mean? Do they mean that they might go either way, or that you are not sure?**] The stack, shlibs and data are too far removed from each other to be spanned by one page table. All kernel `page_tables` are shared by all processes so they are not in the list. Only dirty pages are swapped. Clean pages are stolen so the process can read them back in from the executable if it likes. Mostly only clean pages are shared. A dirty page ends up shared across a fork until the parent or child chooses to write to it again.

5.4 Memory Management data in the process table

Here is a summary of some of the data kept in the process table which is used for memory managment: [**These should be much better documented. We need more details.**]

- Process memory limits: `ulong start_code, end_code, end_data, brk, start_stack;`
- Page fault counting: `ulong min_flt, maj_flt, cmin_flt, cmaj_flt`
- Local descriptor table: `struct desc_struct ldt[32]` is the local descriptor table for task.
- `rss`: number of resident pages.
- `swappable`: if 0, then process's pages will not be swapped.
- `kernel_stack_page`: pointer to page allocated in fork.
- `saved_kernel_stack`: V86 mode stuff.
- `struct tss`
  - Stack segments

    `esp0` kernel stack pointer (`kernel_stack_page`)

    `ss0` kernel stack segment (0x10)

    `esp1` = `ss1` = `esp2` = `ss2` = 0
      unused privelege levels.
  - Segment selectors: `ds` = `es` = `fs` = `gs` = `ss` = `0x17`, `cs` = `0x0f`
    All point to segments in the current `ldt[]`.
  - `cr3`: points to the page directory for this process.
  - `ldt`: `_LDT(n)` selector for current task's LDT.

5.5 Memory initialization

In `start_kernel()` (main.c) there are 3 variables related to memory initialization:

`memory_start`	starts out at 1 MB. Updated by device initialization.
`memory_end`	end of physical memory: 8 MB, 16 MB, or whatever.
`low_memory_start`	end of the kernel code and data that is loaded initially.

Each device init typically takes `memory_start` and returns an updated value if it allocates space at `memory_start` (by simply grabbing it). `paging_init()` initializes the page tables in the `swapper_pg_dir` (starting at 0xc0000000) to cover all of the physical memory from `memory_start` to `memory_end`. Actually the first 4 MB is done in `startup_32` (head.S). `memory_start` is incremented if any new `page_table`s are added. The first page is zeroed to trap null pointer references in the kernel.

In `sched_init()` the `ldt` and `tss` descriptors for `task[0]` are set in the GDT, and loaded into the TR and LDTR (the only time it's done explicitly). A trap gate (0x80) is set up for `system_call()`. The nested task flag is turned off in preparation for entering user mode. The timer is turned on. The `task_struct` for `task[0]` appears in its entirety in `<linux/sched.h>`.

`mem_map` is then constructed by `mem_init()` to reflect the current usage of physical pages. This is the state reflected in the physical memory map of the previous section.

Then Linux moves into user mode with an `iret` after pushing the current `ss`, `esp`, etc. Of course the user segments for `task[0]` are mapped right over the kernel segments so execution continues exactly where it left off.

`task[0]`:

- `pg_dir` = `swapper_pg_dir` which means the the only addresses mapped are in the range 3 GB to 3 GB + `high_memory`.
- `LDT[1]` = user code, base=0xc0000000, size = 640K
- `LDT[2]` = user data, base=0xc0000000, size = 640K

The first `exec()` sets the LDT entries for `task[1]` to the user values of base = 0x0, limit = TASK_SIZE = 0xc0000000. Thereafter, no process sees the kernel segments while in user mode.

5.5.1 Processes and the Memory Manager

Memory-related work done by `fork()`:

- Memory allocation
  - 1 page for the `task_struct`.
  - 1 page for the kernel stack.
  - 1 for the `pg_dir` and some for `pg_tables` (copy_page_tables)
- Other changes
  - `ss0` set to kernel stack segment (0x10) to be sure?
  - `esp0` set to top of the newly allocated `kernel_stack_page`
  - `cr3` set by `copy_page_tables()` to point to newly allocated page directory.
  - `ldt = _LDT(task_nr)` creates new ldt descriptor.
  - descriptors set in gdt for new tss and `ldt[]`.
  - The remaining registers are inherited from parent.

The processes end up sharing their code and data segments (although they have separate local desctriptor tables, the entries point to the same segments). The stack and data pages will be copied when the parent or child writes to them (copy-on-write).

Memory-related work done by `exec()`:

- memory allocation
  - 1 page for exec header entire file for omagic
  - 1 page or more for stack (MAX_ARG_PAGES)
- `clear_page_tables()` used to remove old pages.
- `change_ldt()` sets the descriptors in the new `LDT[]`
- `ldt[1]` = code base=0x00, limit=TASK_SIZE
- `ldt[2]` = data base=0x00, limit=TASK_SIZE
  These segments are DPL=3, P=1, S=1, G=1. type=a (code) or 2 (data)
- Up to `MAX_ARG_PAGES` dirty pages of argv and envp are allocated and stashed at the top of the data segment for the newly created user stack.
- Set the instruction pointer of the caller `eip = ex.a_entry`
- Set the stack pointer of the caller to the stack just created (esp = stack pointer) These will be popped off the stack when the caller resumes.

- update memory limits
  `end_code = ex.a_text`
  `end_data = end_code + ex.a_data`
  `brk = end_data + ex.a_bss`

Interrupts and traps are handled within the context of the current task. In particular, the page directory of the current process is used in address translation. The segments, however, are kernel segments so that all linear addresses point into kernel memory. For example, assume a user process invokes a system call and the kernel wants to access a variable at address 0x01. The linear address is 0xc0000001 (using kernel segments) and the physical address is 0x01. The later is because the process' page directory maps this range exactly as `page_pg_dir`.

The kernel space (0xc0000000 + `high_memory`) is mapped by the kernel page tables which are themselves part of the RESERVED memory. They are therefore shared by all processes. During a fork `copy_page_tables()` treats RESERVED page tables differently. It sets pointers in the process page directories to point to kernel page tables and does not actually allocate new page tables as it does normally. As an example the `kernel_stack_page` (which sits somewhere in the kernel space) does not need an associated `page_table` allocated in the process' `pg_dir` to map it.

The interrupt instruction sets the stack pointer and stack segment from the privilege 0 values saved in the tss of the current task. Note that the kernel stack is a really fragmented object — it's not a single object, but rather a bunch of stack frames each allocated when a process is created, and released when it exits. The kernel stack should never grow so rapidly within a process context that it extends below the current frame.

5.6 Acquiring and Freeing Memory: Paging Policy

When any kernel routine wants memory it ends up calling `get_free_page()`. This is at a lower level than `kmalloc()` (in fact `kmalloc()` uses `get_free_page()` when it needs more memory).

`get_free_page()` takes one parameter, a priority. Possible values are `GFP_BUFFER`, `GFP_KERNEL`, and `GFP_ATOMIC`. It takes a page off of the `free_page_list`, updates `mem_map`, zeroes the page and returns the physical address of the page (note that `kmalloc()` returns a physical address. The logic of the mm depends on the identity map between logical and physical addresses).

That itself is simple enough. The problem, of course, is that the `free_page_list` may be empty. If you did not request an atomic operation, at this stage, you enter into the realm of page stealing which we'll go into in a moment. As a last resort (and for atomic requests) a page is torn off from the `secondary_page_list` (as you may have guessed, when pages are freed, the `secondary_page_list` gets filled up first).

The actual manipulation of the `page_lists` and `mem_map` occurs in this mysterious macro called `REMOVE_FROM_MEM_QUEUE()` which you probably never want to look into. Suffice it to say that interrupts are disabled. [**I think that this should be explained here. It is not *that* hard...**]

Now back to the page stealing bit. `get_free_page()` calls `try_to_free_page()` which repeatedly calls `shrink_buffers()` and `swap_out()` in that order until it is successful in freeing a page. The priority is increased on each successive iteration so that these two routines run through their page stealing loops more often.

Here's one run through `swap_out()`:

- Run through the process table and get a swappable task say Q.
- Find a user page table (not RESERVED) in Q's space.
- For each *page* in the table `try_to_swap_out(` *page* `)`.
- Quit when a page is freed.

Note that `swap_out()` (called by `try_to_free_page()`) maintains static variables so it may resume the search where it left off on the previous call.

`try_to_swap_out()` scans the page tables of all user processes and enforces the stealing policy:

1. Do not fiddle with RESERVED pages.
2. Age the page if it is marked accessed (1 bit).
3. Don't tamper with recently acquired pages (`last_free_pages[]`).
4. Leave dirty pages with `map_counts` > 1 alone.
5. Decrement the `map_count` of clean pages.
6. Free clean pages if they are unmapped.
7. Swap dirty pages with a `map_count` of 1.

Of these actions, 6 and 7 will stop the process as they result in the actual freeing of a physical page. Action 5 results in one of the processes losing an unshared clean page that was not accessed recently (decrement *Q*`->rss`) which is not all that bad, but the cumulative effects of a few iterations can slow down a process considerably. At present, there are 6 iterations, so a page shared by 6 processes can get stolen if it is clean.

Page table entries are updated and the TLB invalidated. [**Wonder about the latter. It seems unnecessary since accessed pages aren't offed and there is a walk through many page tables between iterations . . . may be in case an interrupt came along and wanted the most recently axed page?**]

The actual work of freeing the page is done by `free_page()`, the complement of `get_free_page()`. It ignores RESERVED pages, updates `mem_map`, then frees the page and updates the `page_lists` if it is unmapped. For swapping (in 6 above), `write_swap_page()` gets called and does nothing remarkable from the memory management perspective.

The details of `shrink_buffers()` would take us too far afield. Essentially it looks for free buffers, then writes out dirty buffers, then goes at busy buffers and calls `free_page()` when its able to free all the buffers on a page.

Note that page directories and page tables along with RESERVED pages do not get swapped, stolen or aged. They are mapped in the process page directory through reserved page tables. They are freed only on exit from the process.

5.7 The page fault handlers

When a process is created via fork, it starts out with a page directory and a page or so of the executable. So the page fault handler is the source of most of a processes' memory.

The page fault handler `do_page_fault()` retrieves the faulting address from the register cr2. The error code (retrieved in sys_call.S) differentiates user/supervisor access and the reason for the fault — write protection or a missing page. The former is handled by `do_wp_page()` and the latter by `do_no_page()`.

If the faulting address is greater than TASK_SIZE the process receives a SIGKILL. [**Why this check? This can only happen in kernel mode because of segment level protection.**]

These routines have some subtleties as they can get called from an interrupt. You can't assume that it is the 'current' task that is executing.

`do_no_page()` handles three possible situations:

1. The page is swapped.

2. The page belongs to the executable or a shared library.
3. The page is missing — a data page that has not been allocated.

In all cases `get_empty_pgtable()` is called first to ensure the existence of a page table that covers the faulting address. In case 3 `get_empty_page()` is called to provide a page at the required address and in case of the swapped page, `swap_in()` is called.

In case 2, the handler calls `share_page()` to see if the page is shareable with some other process. If that fails it reads in the page from the executable or library (It repeats the call to `share_page()` in case another process did the same meanwhile). Any portion of the page beyond the brk value is zeroed.

A page read in from the disk is counted as a major fault (`maj_flt`). This happens with a `swap_in()` or when it is read from the executable or a library. Other cases are deemed minor faults (`min_flt`).

When a shareable page is found, it is write-protected. A process that writes to a shared page will then have to go through `do_wp_page()` which does the copy-on-write.

`do_wp_page()` does the following:

- send SIGSEGV if any user process is writing to current `code_space`.
- If the old page is not shared then just unprotect it.
  Else `get_free_page()` and `copy_page()`. The page acquires the dirty flag from the old page. Decrement the map count of the old page.

5.8 Paging

Paging is swapping on a page basis rather than by entire processes. We will use swapping here to refer to paging, since Linux only pages, and does not swap, and people are more used to the word “swap” than “page.” Kernel pages are never swapped. Clean pages are also not written to swap. They are freed and reloaded when required. The swapper maintains a single bit of aging info in the `PAGE_ACCESSED` bit of the page table entries. [**What are the maintainance details? How is it used?**]

Linux supports multiple swap files or devices which may be turned on or off by the swapon and swapoff system calls. Each swapfile or device is described by a `struct swap_info_struct` (swap.c).

```
    static struct swap_info_struct {
        unsigned long flags;
        struct inode * swap_file;
        unsigned int swap_device;
        unsigned char * swap_map;
        char * swap_lockmap;
        int lowest_bit;
        int highest_bit;
    } swap_info[MAX_SWAPFILES];
```

The flags field (SWP_USED or SWP_WRITEOK) is used to control access to the swap files. When SWP_WRITEOK is off space will not be allocated in that file. This is used by swapoff when it tries to unuse a file. When swapon adds a new swap file it sets SWP_USED. A static variable nr_swapfiles stores the number of currently active swap files. The fields lowest_bit and highest_bit bound the free region in the swap file and are used to speed up the search for free swap space.

The user program mkswap initializes a swap device or file. The first page contains a signature ('SWAP-SPACE') in the last 10 bytes, and holds a bitmap. Initially 0's in the bitmap signal bad pages. A '1' in the bitmap means the corresponding page is free. This page is never allocated so the initialization needs to be done just once.

The syscall swapon() is called by the user program swapon typically from /etc/rc. A couple of pages of memory are allocated for swap_map and swap_lockmap.

swap_map holds a byte for each page in the swapfile. It is initialized from the bitmap to contain a 0 for available pages and 128 for unusable pages. It is used to maintain a count of swap requests on each page in the swap file. swap_lockmap holds a bit for each page that is used to ensure mutual exclusion when reading or writing swap files.

When a page of memory is to be swapped out an index to the swap location is obtained by a call to get_swap_page(). This index is then stored in bits 1–31 of the page table entry so the swapped page may be located by the page fault handler, do_no_page() when needed.

The upper 7 bits of the index give the swapfile (or device) and the lower 24 bits give the page number on that device. That makes as many as 128 swapfiles, each with room for about 64 GB, but the space overhead due to the swap_map would be large. Instead the swapfile size is limited to 16 MB, because the swap_map then takes 1 page.

The function swap_duplicate() is used by copy_page_tables() to let a child process inherit swapped pages during a fork. It just increments the count maintained in swap_map for that page. Each process will swap in a separate copy of the page when it accesses it.

swap_free() decrements the count maintained in swap_map. When the count drops to 0 the page can be reallocated by get_swap_page(). It is called each time a swapped page is read into memory (`swap_in()`) or when a page is to be discarded (`free_one_table()`, etc.).

5.9 80386 Memory Mangament

A logical address specified in an instruction is first translated to a linear address by the segmenting hardware. This linear address is then translated to a physical address by the paging unit.

5.9.1 Paging on the 386

There are two levels of indirection in address translation by the paging unit. A **page directory** contains pointers to 1024 page tables. Each **page table** contains pointers to 1024 pages. The register CR3 contains the physical base address of the page directory and is stored as part of the TSS in the `task_struct` and is therefore loaded on each task switch.

A 32-bit Linear address is divided as follows:

31 22	21 12	11 0
DIR	TABLE	OFFSET

Physical address is then computed (in hardware) as:

CR3 + DIR	points to the table_base.
table_base + TABLE	points to the page_base.
physical_address =	page_base + OFFSET

Page directories (page tables) are page aligned so the lower 12 bits are used to store useful information about the page table (page) pointed to by the entry.

Format for Page directory and Page table entries:

31 12	11 9	8	7	6	5	4	3	2	1	0
ADDRESS	OS	0	0	D	A	0	0	U/S	R/W	P

D 1 means page is dirty (undefined for page directory entry).
R/W 0 means readonly for user.
U/S 1 means user page.
P 1 means page is present in memory.
A 1 means page has been accessed (set to 0 by aging).
OS bits can be used for LRU etc, and are defined by the OS.

The corresponding definitions for Linux are in `<linux/mm.h>`.

When a page is swapped, bits 1–31 of the page table entry are used to mark where a page is stored in swap (bit 0 must be 0).

Paging is enabled by setting the highest bit in CR0. **[in head.S?]** At each stage of the address translation access permissions are verified and pages not present in memory and protection violations result in page faults. The fault handler (in memory.c) then either brings in a new page or unwriteprotects a page or does whatever needs to be done.

Page Fault handling Information

- The register CR2 contains the linear address that caused the last page fault.
- Page Fault Error Code (16 bits):

bit	cleared	set
0	page not present	page level protection
1	fault due to read	fault due to write
2	supervisor mode	user mode

The rest are undefined. These are extracted in sys_call.S.

The Translation Lookaside Buffer (TLB) is a hardware cache for physical addresses of the most recently used virtual addresses. When a virtual address is translated the 386 first looks in the TLB to see if the information it needs is available. If not, it has to make a couple of memory references to get at the page directory and then the page table before it can actually get at the page. Three physical memory references for address translation for every logical memory reference would kill the system, hence the TLB.

The TLB is flushed if CR3 loaded or by task switch that changes CR0. It is explicitly flushed in Linux by calling `invalidate()` which just reloads CR3.

5.9.2 Segments in the 80386

Segment registers are used in address translation to generate a linear address from a logical (virtual) address.

```
linear_address = segment_base + logical_address
```

The linear address is then translated into a physical address by the paging hardware.

Each segment in the system is described by a 8 byte segment descriptor which contains all pertinent information (base, limit, type, privilege).

The segments are:
Regular segments

- code and data segments

System segments

- (TSS) task state segments
- (LDT) local descriptor tables

Characteristics of system segments:

- System segments are task specific.
- There is a Task State Segment (TSS) associated with each task in the system. It contains the `tss_struct` (sched.h). The size of the segment is that of the `tss_struct` excluding the `i387_union` (232 bytes). It contains all the information necessary to restart the task.
- The LDT's contain regular segment descriptors that are private to a task. In Linux there is one LDT per task. There is room for 32 descriptors in the linux `task_struct`. The normal LDT generated by Linux has a size of 24 bytes, hence room for only 3 entries as above. Its contents are:
  - LDT[0] Null (mandatory)
  - LDT[1] user code segment descriptor.
  - LDT[2] user data/stack segment descriptor.
- The user segments all have base=0x00 so that the linear address is the same as the logical address.

To keep track of all these segments, the 386 uses a global descriptor table (GDT) that is setup in memory by the system (located by the GDT register). The GDT contains a segment descriptors for each task state segment, each local descriptor tablet and also regular segments. The Linux GDT contains just two normal segment entries:

- GDT[0] is the null descriptor.
- GDT[1] is the kernel code segment descriptor.
- GDT[2] is the kernel data/stack segment descriptor.

The rest of the GDT is filled with TSS and LDT system descriptors:

- GDT[3] ???
- GDT[4] = TSS0, GDT[5] = LDT0,
- GDT[6] = TSS1, GDT[7] = LDT1
- ...etc ...

Note LDT[n] != LDTn

- LDT[n] = the nth descriptor in the LDT of the current task.
- LDTn = a descriptor in the GDT for the LDT of the nth task.

At present the GDT has a total of 256 entries or room for as many as 126 tasks. The kernel segments have base 0xc0000000 which is where the kernel lives in the linear view. Before a segment can be used, the contents of the descriptor for that segment must be loaded into the segment register. The 386 has a complex set of criteria regarding access to segments so you can't simply load a descriptor into a segment register. Also these segment registers have programmer invisible portions. The visible portion is what is usually called a segment register: cs, ds, es, fs, gs, and ss.

The programmer loads one of these registers with a 16-bit value called a selector. The selector uniquely identifies a segment descriptor in one of the tables. Access is validated and the corresponding descriptor loaded by the hardware.

Currently Linux largely ignores the (overly?) complex segment level protection afforded by the 386. It is biased towards the paging hardware and the associated page level protection. The segment level rules that apply to user processes are

1. A process cannot directly access the kernel data or code segments
2. There is always limit checking but given that every user segment goes from 0x00 to 0xc0000000 it is unlikely to apply. **[This has changed, and needs updating, please.]**

5.9.3 Selectors in the 80386

A segment selector is loaded into a segment register (cs, ds, etc.) to select one of the regular segments in the system as the one addressed via that segment register.

Segment selector Format:

15 3	2 1	0
index	TI	RPL

TI Table indicator:

0 means selector indexes into GDT

1 means selector indexes into LDT

RPL Privelege level. Linux uses only two privelege levels.

0 means kernel

3 means user

Examples:

Kernel code segment

TI=0, index=1, RPL=0, therefore selector = 0x08 (GDT[1])

User data segment

TI=1, index=2, RPL=3, therefore selector = 0x17 (LDT[2])

Selectors used in Linux:

TI	**index**	**RPL**	**selector**	**segment**	
0	1	0	0x08	kernel code	GDT[1]
0	2	0	0x10	kernel data/stack	GDT[2]
0	3	0	???	???	GDT[3]
1	1	3	0x0F	user code	LDT[1]
1	2	3	0x17	user data/stack	LDT[2]

Selectors for system segments are not to be loaded directly into segment registers. Instead one must load the TR or LDTR.

On entry into syscall:

- ds and es are set to the kernel data segment (0x10)
- fs is set to the user data segment (0x17) and is used to access data pointed to by arguments to the system call.
- The stack segment and pointer are automatically set to ss0 and esp0 by the interrupt and the old values restored when the syscall returns.

5.9.4 Segment descriptors

There is a segment descriptor used to describe each segment in the system. There are regular descriptors and system descriptors. Here's a descriptor in all its glory. The strange format is essentally to maintain compatibility with the 286. Note that it takes 8 bytes.

63–54	55	54	53	52	51–48	47	46	45	44–40	39–16	15–0
Base 31–24	G	D	R	U	Limit 19–16	P	DPL	S	TYPE	Segment Base 23–0	Segment Limit 15–0

Explanation:

R reserved (0)
DPL 0 means kernel, 3 means user
G 1 means 4K granularity (Always set in Linux)
D 1 means default operand size 32bits
U programmer definable
P 1 means present in physical memory
S 0 means system segment, 1 means normal code or data segment.
Type There are many possibilities. Interpreted differently for system and normal descriptors.

Linux system descriptors:
TSS: P=1, DPL=0, S=0, type=9, limit = 231 room for 1 `tss_struct`.
LDT: P=1, DPL=0, S=0, type=2, limit = 23 room for 3 segment descriptors.
The base is set during `fork()`. There is a TSS and LDT for each task.

Linux regular kernel descriptors: (head.S)
code: P=1, DPL=0, S=1, G=1, D=1, type=a, base=0xc0000000, limit=0x3ffff
data: P=1, DPL=0, S=1, G=1, D=1, type=2, base=0xc0000000, limit=0x3ffff

The LDT for task[0] contains: (sched.h)
code: P=1, DPL=3, S=1, G=1, D=1, type=a, base=0xc0000000, limit=0x9f
data: P=1, DPL=3, S=1, G=1, D=1, type=2, base=0xc0000000, limit=0x9f

The default LDT for the remaining tasks: (`exec()`)
code: P=1, DPL=3, S=1, G=1, D=1, type=a, base=0, limit= 0xbffff
data: P=1, DPL=3, S=1, G=1, D=1, type=2, base=0, limit= 0xbffff

The size of the kernel segments is 0x40000 pages (4KB pages since G=1 = 1 Gigabyte. The type implies that the permissions on the code segment is read-exec and on the data segment is read-write.

Registers associated with segmentation. Format of segment register: (Only the selector is programmer visible)

16-bit	32-bit	32-bit	
selector	physical base addr	segment limit	attributes

The invisible portion of the segment register is more conveniently viewed in terms of the format used in the descriptor table entries that the programmer sets up. The descriptor tables have registers associated with them that are used to locate them in memory. The GDTR (and IDTR) are initialized at startup once the tables are defined. The LDTR is loaded on each task switch.

Format of GDTR (and IDTR):

32-bits	16-bits
Linear base addr	table limit

The TR and LDTR are loaded from the GDT and so have the format of the other segment registers. The task register (TR) contains the descriptor for the currently executing task's TSS. The execution of a jump to a TSS selector causes the state to be saved in the old TSS, the TR is loaded with the new descriptor and the registers are restored from the new TSS. This is the process used by schedule to switch to various user tasks. Note that the field `tss_struct.ldt` contains a selector for the LDT of that task. It is used to load the LDTR. (sched.h)

5.9.5 Macros used in setting up descriptors

Some assembler macros are defined in sched.h and system.h to ease access and setting of descriptors. Each TSS entry and LDT entry takes 8 bytes.

Manipulating GDT system descriptors:

- `_TSS(n)`,
  `_LDT(n)` These provide the index into the GDT for the n'th task.
- `_LDT(n)` is stored in the the ldt field of the `tss_struct` by fork.
- `_set_tssldt_desc(n, addr, limit, type)`
  `ulong *n` points to the GDT entry to set (see fork.c). The segment base (TSS or LDT) is set to 0xc0000000 + `addr`. Specific instances of the above are, where ltype refers to the byte containing P, DPL, S and type:

  `set_ldt_desc(n, addr)` ltype = 0x82

  P=1, DPL=0, S=0, type=2 means LDT entry. limit = 23 =¿ room for 3 segment descriptors.

`set_tss_desc(n, addr)` ltype = 0x89
P=1, DPL=0, S=0, type = 9, means available 80386 TSS limit = 231
room for 1 tss_struct.

- `load_TR(n)`,
  `load_ldt(n)` load descriptors for task number n into the task register and ldt register.

- `ulong get_base (struct desc_struct ldt)` gets the base from a descriptor.

- `ulong get_limit (ulong segment)` gets the limit (size) from a segment selector. Returns the size of the segment in bytes.

- `set_base(struct desc_struct ldt, ulong base)`,
  `set_limit(struct desc_struct ldt, ulong limit)`
  Will set the base and limit for descriptors (4K granular segments). The limit here is actually the size in bytes of the segment.

- `_set_seg_desc(gate_addr, type, dpl, base, limit)`
  Default values 0x00408000 =¿ D=1, P=1, G=0
  Present, operation size is 32 bit and max size is 1M.
  `gate_addr` must be a `(ulong *)`

Appendix A

Bibliography

Two bibliographies for now...

A.1 Normal Bibliography

Bibliography

[ANS] *Draft Proposed American National Standard for Information Systems: Small Computer System Interface – 2 (SCSI-2).* (X3T9.2/86-109, revision 10h, October 17, 1991).

[Int90] Intel. *i486 Processor Programmer's Reference Manual.* Intel/McGraw-Hill, 1990.

[LXT91] *LXT SCSI Products: Specification and OEM Technical Manual*, 1991.

[Nor85] Peter Norton. *The Peter Norton Programmer's Guide to the IBM PC.* Bellevue, Washington: Microsoft Press, 1985.

A.2 Annotated Bibliography

This anontated bibliography covers books on operating system theory as well as different kinds of programming in a UN⋆X environment. The price marked may or may not be an exact price, but should be close enough for government work. [**If you have a book that you think should go in the bibliography, please write a short review of it and send all the necessary information (title, author, publisher, ISBN, and approximate price) and the review to** `johnsonm@sunsite.unc.edu`]

This version is slowly going away, in favor of a real bibliography.

Title: The Design of the UNIX Operating System
Author: Maurice J. Bach
Publisher: Prentice Hall, 1986
ISBN: 0-13-201799-7
Appr. Price: $65.00

This is one of the books that Linus used to design Linux. It is a description of the data structures used in the System V kernel. Many of the names of the important functions in the Linux source come from this book, and are named after the algorithms presented here. For instance, if you can't quite figure out what exactly getblk(), brelse(), bread(), breada(), and bwrite() are, chapter 3 explains very well.

While most of the algorithms are similar or the same, a few differences are worth noting:

- The Linux buffer cache is dynamically resized, so the algorithm for dealing with getting new buffers is a bit different. Therefore the above referenced explanation of getblk() is a little different than the getblk() in Linux.
- Linux does not currently use streams, and if/when streams are implemented for Linux, they are likely to have somewhat different semantics.
- The semantics and calling structure for device drivers is different. The concept is similar, and the chapter on device drivers is still worth reading, but for details on the device driver structures, the *The Linux Kernel Hackers' Guide* is the proper reference.
- The memory management algorithms are somewhat different.

There are other small differences as well, but a good understanding of this text will help you understand the Linux source.

Title:	Advanced Programming in the UNIX Environment
Author:	W. Richard Stevens
Publisher:	Addison Wesley, 1992
ISBN:	0-201-56317-7
Appr. Price:	$50.00

This excellent tome covers the stuff you *really* have to know to write *real* UN⋆X programs. It includes a discussion of the various standards for UN⋆X implementations, including POSIX, X/Open XPG3, and FIPS, and concentrates on two implementations, SVR4 and pre-release 4.4 BSD, which it refers to as 4.3+BSD. The book concentrates heavily on application and fairly complete specification, and notes which features relate to which standards and releases.

The chapters include: Unix Standardization and Implementations, File I/O, Files and Directories, Standard I/O Library, System Data Files and Information, The Environment of a Unix Process, Process Control, Process Relationships, Signals, Terminal I/O, Advanced I/O (non-blocking, streams, async, memory-mapped, etc.), Daemon Processes, Interprocess Communication, Advanced Interprocess Communication, and some example applications, including chapters on A Database Library, Commmunicating with a PostScript Printer, A Modem Dialer, and then a seemingly misplaced final chapter on Pseudo Terminals.

I have found that this book makes it possible for me to write useable programs for UN⋆X. It will help you achieve POSIX compliance in ways that won't break SVR4 or BSD, as a general rule. This book will save you ten times its cost in frustration.

Title:	Advanced 80386 Programming Techniques
Author:	James L. Turley
Publisher:	Osborne McGraw-Hill, 1988
ISBN:	0-07-881342-5
Appr. Price:	$22.95

This book covers the 80386 quite well, without touching on any other hardware. Some code samples are included. All major features are covered, as are many of the concepts needed. The chapters of this book are: Basics, Memory Segmentation, Privilege Levels, Paging, Multitasking, Communicating Among Tasks, Handling Faults and Interrupts, 80286 Emulation, 8086 Emulation, Debugging, The 80387 Numeric Processor Extension, Programming for Performance, Reset and Real Mode, Hardware, and a few appendices, including tables of the memory management structures as a handy reference.

The author has a good writing style: If you are technically minded, you will find yourself caught up just reading this book. One strong feature of this book for Linux is that the author is very careful not to explain how to do things under DOS, nor how to deal with particular hardware. In fact, the only times he mentions DOS and PC-compatible hardware are in the introduction, where he promises never to mention them again.

Title:	The C Programming Language, second edition
Author:	Brian W. Kernighan and Dennis M. Ritchie
Publisher:	Prentice Hall, 1988
ISBN:	0-13-110362-8 (paper) 0-13-110370-9 (hard)
Appr. Price:	$35.00

The C programming bible. Includes a C tutorial, UN⋆X interface reference, C reference, and standard library reference.

You program in C, you buy this book. It's that simple.

Title:	Operating Systems: Design and Implementation
Author:	Andrew S. Tanenbaum
Publisher:	Prentice Hall, 1987
ISBN:	0-13-637406-9
Appr. Price:	$50.00

This book, while a little simplistic in spots, and missing some important ideas, is a fairly clear exposition of what it takes to write an operating system. Half the book is taken up with the source code to a UN⋆X clone called Minix, which is based on a microkernel, unlike Linux, which sports a monolithic design. It has been said that Minix shows that it is possible to to write a microkernel-based UN⋆X, but does not adequately explain *why* one would do so.

Linux was originally intended to be a free Minix replacement:[1] In fact, it was originally to be binary-compatible with Minix-386. Minix-386 was the development environment under which Linux was bootstrapped. No Minix code is in Linux, but vesitiges of this heritage live on in such things as the minix filesystem in Linux. Early in Linux's existence, Andrew Tanenbaum started a flame war with Linus about OS design, which was interesting, if not enlightening…

However, this book might still prove worthwhile for those who want a basic explanation of OS concepts, as Tanenbaum's explanations of the basic concepts remain some of the clearer (and more entertaining, if you like to be entertained) available. Unfortunately, basic is the key work here, as many things such as virtual memory are not covered at all.

Title:	Modern Operating Systems
Author:	Andrew S. Tanenbaum
Publisher:	Prentice Hall, 1992
ISBN:	0-13-588187-0
Appr. Price:	$51.75

[1] Linus' Minix, Linus tells us.

The first half of this book is a rewrite of Tanenbaum's earlier *Operating Systems*, but this book covers several things that the earlier book missed, including such things as virtual memory. Minix is not included, but overviews of MS-DOS and several distributed systems are. This book is probably more useful to someone who wants to do something with his or her knowlege than Tanenbaum's earlier *Operating Systems: Design and Implementation.* Some clue as to the reason may be found in the title... However, what DOS is doing in a book on *modern* operating systems, many have failed to discover.

Title:	Operating Systems
Author:	William Stallings
Publisher:	Macmillan, 1992 (800-548-9939)
ISBN:	0-02-415481-4
Appr. Price:	$??.??

A very thorough text on operating systems, this book gives more in-depth coverage of the topics covered in Tannebaum's books, and covers more topics, in a much brisker style. This book covers all the major topics that you would need to know to build an operating system, and does so in a clear way. The author uses examples from three major systems, comparing and contrasting them: UN⋆X, OS/2, and MVS. With each topic covered, these example systems are used to clarify the points and provide an example of an implementation.

Topics covered in *Operating Systems* include threads, real-time systems, multiprocessor scheduling, distributed systems, process migration, and security, as well as the standard topics like memory management and scheduling. The section on distributed processing appears to be up-to-date, and I found it very helpful.

Title:	UNIX Network Programming
Author:	W. Richard Stevens
Publisher:	Prentice Hall, 1990
ISBN:	0-13-949876-1
Appr. Price:	$48.75

This book covers several kinds of networking under UN⋆X, and provides very thorough references to the forms of networking that it does not cover directly. It covers TCP/IP and XNS most heavily, and fairly exhaustively describes how all the calls work. It also has a description and sample code using System V's TLI, and pretty complete coverage of System V IPC. This book contains a lot of source code examples to get you started, and many useful proceedures. One example is code to provide useable semaphores, based on the partially broken implementation that System V provides.

Title:	Programming in the UNIX environment
Author:	Brian W. Kernighan and Robert Pike
Publisher:	Prentice Hall, 1984
ISBN:	0-13-937699 (hardcover) 0-13-937681-X (paperback)
Appr. Price:	$??.??

no abstract

Title:	Writing UNIX Device Drivers
Author:	George Pajari
Publisher:	Addison Wesley, 1992
ISBN:	0-201-52374-4
Appr. Price:	$32.95

This book is written by the President and founder of Driver Design Labs, a company which specializes in the development of UN⋆X device drivers. This book is an excellent introduction to the sometimes wacky world of device driver design. The four basic types of drivers (character, block, tty, STREAMS) are first discussed briefly. Many full examples of device drivers of all types are given, starting with the simplest and progressing in complexity. All examples are of drivers which deal with UN⋆X on PC-compatible hardware. **Chapters include:** Character Drivers I: A Test Data Generator Character Drivers II: An A/D Converter Character Drivers III: A Line Printer Block Drivers I: A Test Data Generator Block Drivers II: A RAM Disk Driver Block Drivers III: A SCSI Disk Driver Character Drivers IV: The Raw Disk Driver Terminal Drivers I: The COM1 Port Character Drivers V: A Tape Drive STREAMS Drivers I: A Loop-Back Driver STREAMS Drivers II: The COM1 Port (Revisited) Driver

Installation Zen and the Art of Device Driver Writing

Although many of the calls used in the book are not Linux-compatible, the general idea is there, and many of the ideas map directly into Linux.

Title:	title
Author:	author
Publisher:	pub,yr
ISBN:	isbn
Appr. Price:	$??.??

no abstract

Appendix B

Tour of the Linux kernel source

[This is an alpha release of a chapter written by Alessandro Rubini, `rubini@ipvvis.unipv.it`**. I'm including it here as it gets worked on for comments.]**

This chapter tries to explain the Linux source code in an orderly manner, trying to help the reader to achieve a good understanding of how the source code is laid out and how the most relevant unix features are implemented. The target is to help the experienced C programmer who is not accustomed to Linux in getting familiar with the overall Linux design. That's why the chosen entry point for the kernel tour is the kernel own entry point: system boot.

A good understanding of C language is required to understand this material, as well as some familiarity with both UN⋆X concepts and the PC architecture. However, no C code will appear in this chapter, but rather pointers to the actual code. The finest issues of kernel design are explained in other chapters of this guide, while this chapter tends to remain an informal overview.

Any pathname for files referenced in this chapter is referred to the main source-tree directory, usually `/usr/src/linux`.

> Most of the information reported here is taken from the source code of Linux release 1.0. Nonetheless, references to later versions are provided at times. Any paragraph within the tour shaped like this one is meant to underline changes the kernel has undergone after the 1.0 release. If no such paragraph is present, then no changes occurred up to release 1.0.9 – 1.1.76.

> Sometimes a paragraph like this occurs in the text. It is a pointer to the right sources to get more information on the subject just covered. Needless to say, *the source* is the primary source.

B.1 Booting the system

When the PC is powered up, the 80x86 processor finds itself in real mode and executes the code at address 0xFFFF0, which corresponds to a ROM-BIOS address. The PC BIOS performs some tests on the system and initializes the interrupt vector at physical address 0. After that it loads the first sector of a bootable device to 0x7C00, and jumps to it. The device is usually the floppy or the hard drive. The preceding description is quite a simplified one, but it's all that's needed to understand the kernel initial workings.

The very first part of the Linux kernel is written in 8086 assembly language (`boot/bootsect.S`). When run, it moves itself to absolute address 0x90000, loads the next 2 kBytes of code from the boot device to address 0x90200, and the rest of the kernel to address 0x10000. The message "`Loading...`" is displayed during system load. Control is then passed to the code in `boot/Setup.S`, another real-mode assembly source.

The setup portion identifies some features of the host system and the type of vga board. If requested to, it asks the user to choose the video mode for the console. It then moves the whole system from address 0x10000 to address 0x1000, enters protected mode and jumps to the rest of the system (at 0x1000).

The next step is kernel decompression. The code at 0x1000 comes from `zBoot/head.S` which initializes registers and invokes `decompress_kernel()`, which in turn is made up of `zBoot/inflate.c`, `zBoot/unzip.c` and `zBoot/misc.c`. The decompressed data goes to address 0x100000 (1 Meg), and this is the main reason why Linux can't run with less than 2 megs ram.

> Encapsulation of the kernel in a gzip file is accomplished by `Makefile` and utilities in the `zBoot` directory. They are interesting files to look at.
>
> Kernel release 1.1.75 moved the `boot` and `zBoot` directories down to `arch/i386/boot`. This change is meant to allow true kernel builds for different architectures. Nonetheless, I'll stick to i386-specific information.

Decompressed code is executed at address 0x1010000 [**Maybe I've lost track of physical addresses, here, as I don't know very well gas source code**], where all the 32-bit setup is accomplished: IDT, GDT and LDT are loaded, the processor and coprocessor are identified, and paging is setup; eventually, the routine `start_kernel` is invoked. The source for the above operations is in `boot/head.S`. It is probably the trickiest code in the whole kernel.

Note that if an error occurs during any of the preceding steps, the computer will lockup. The OS can't deal with errors when it isn't yet fully operative.

`start_kernel()` resides in `init/main.c`, and never returns. Anything from now on is coded in C language, left aside interrupt management and system call enter/leave (well, most of the macros embed assembly code, too).

B.2 Spinning the wheel

After dealing with all the tricky questions, `start_kernel()` initializes all the parts of the kernel, specifically:

- Sets the memory bounds and calls `paging_init()`.
- Initializes the traps, IRQ channels and scheduling.
- Parses the command line.
- If requested to, allocates a profiling buffer.
- Initializes all the device drivers and disk buffering, as well as other minor parts.
- Calibrates the delay loop (computes the "BogoMips" number).
- Checks if interrupt 16 works with the coprocessor.

Finally, the kernel is ready to `move_to_user_mode()`, in order to fork the `init` process, whose code is in the same source file. Process number 0 then, the so-called idle task, keeps running in an infinite idle loop.

The `init` process tries to execute `/etc/init`, or `/bin/init`, or `/sbin/init`.

If none of them succeeds, code is provided to execute "`/bin/sh /etc/rc`" and fork a root shell on the first terminal. This code dates back to Linux 0.01, when the OS was made by the kernel alone, and no `login` process was available.

After `exec()`ing the `init` program from one of the standard places (let's assume we have one of them), the kernel has no direct control on the program flow. Its role, from now on is to provide processes with system calls, as well as servicing asynchronous events (such as hardware interrupts). Multitasking has been setup, and it is now `init` who manages multiuser access by `fork()`ing system daemons and `login` processes.

Being the kernel in charge of providing services, the tour will proceed by looking at those services (the "system calls"), as well as by providing general ideas about the underlying data structures and code organization.

B.3 How the kernel sees a process

¿From the kernel point of view, a process is an entry in the process table. Nothing more.

The process table, then, is one of the most important data structures within the system, together with the memory-management tables and the buffer cache. The individual item in the process table is the `task_struct` structure, quite a huge one, defined in `include/linux/sched.h`. Within the `task_struct` both low-level and high-level information is kept— ranging from the copy of some hardware registers to the inode of the working directory for the process.

The process table is both an array and a double-linked list, as well as a tree. The physical implementation is a static array of pointers, whose length is `NR_TASKS`, a constant defined in `include/linux/tasks.h`, and each structure resides in a reserved memory page. The list structure is achieved through the pointers `next_task` and `prev_task`, while the tree structure is quite complex and will not be described here. You may wish to change `NR_TASKS` from the default vaue of 128, but be sure to have proper dependency files to force recompilation of all the source files involved.

After booting is over, the kernel is always working on behalf of one of the processes, and the global variable `current`, a pointer to a `task_struct` item, is used to record the running one. `current` is only changed by the scheduler, in `kernel/sched.c`. When, however, all procecces must be looked at, the macro `for_each_task` is used. It is conderably faster than a sequential scan of the array, when the system is lightly loaded.

A process is always running in either "user mode" or "kernel mode". The main body of a user program is executed in user mode and system calls are executed in kernel mode. The stack used by the process in the two execution modes is different—a conventional stack segment is used for user mode, while a fixed-size stack (one page, owned by the process) is used in kernel mode. The kernel stack page is never swapped out, because it must be available whenever a system call is entered.

System calls, within the kernel, exist as C language functions, their 'official' name being prefixed by '`sys_`'. A system call named, for example, *burnout* invokes the kernel function `sys_`*burnout*`()`.

> The system call mechanism is described in chapter 3 of this guide. Looking at `for_each_task` and `SET_LINKS`, in `include/linux/sched.h` can help understanding the list and tree structures in the process table.

B.4 Creating and destroying processes

A unix system creates a process though the `fork()` system call, and process termination is performed either by `exit()` or by receiving a signal. The Linux implementation for them resides in `kernel/fork.c` and `kernel/exit.c`.

Forking is easy, and `fork.c` is short and ready understandable. Its main task is filling the data structure for the new process. Relevant steps, apart from filling fields, are

- getting a free page to hold the `task_struct`
- finding an empty process slot (`find_empty_process()`)
- getting another free page for the `kernel_stack_page`
- copying the father's LDT to the child
- duplicating `mmap` information of the father

`sys_fork()` also manages file descriptors and inodes.

> The 1.0 kernel offers some vestigial support to threading, and the `fork()` system call shows some hints to that. Kernel threads is work-in-progress outside the mainstream kernel.

Exiting from a process is trickier, because the parent process must be notified about any child who exits. Moreover, a process can exit by being `kill()`ed by another process (these are UN⋆X features). The file `exit.c` is therefore the home of `sys_kill()` and the vairious flavours of `sys_wait()`, in addition to `sys_exit()`.

The code belonging to `exit.c` is not described here—it is not that interesting. It deals with a lot of details in order to leave the system in a consistent state. The POSIX standard, then, is quite demanding about signals, and it must be dealt with.

B.5 Executing programs

After `fork()`ing, two copies of the same program are running. One of them usually `exec()`s another program. The `exec()` system call must locate the binary image of the executable file, load and run it. The word 'load' doesn't necessarily mean "copy in memory the binary image", as Linux supports demand loading.

The Linux implementation of `exec()` supports different binary formats. This is accomplished through the `linux_binfmt` structure, which embeds two pointers to functions—one to load the executable and the other to load the library, each binary format representing both the executable and the library. Loading of shared libraries is implemented in the same source file as exec() is, but let's stick to exec() itself.

The UN⋆X systems provide the programmer with six flavours of the exec() function. All but one of them can be implemented as library functions, and theLinux kernel implements sys_execve() alone. It performs quite a simple task: loading the head of the executable, and trying to execute it. If the first two bytes are "#!", then the first line is parsed and an interpreter is invoked, otherwise the registered binary formats are sequentially tried.

The native Linux format is supported directly within fs/exec.c, and the relevant functions are load_aout_binary and load_aout_library. As for the binaries, the function loading an *a.out* executable ends up either in mmap()ing the disk file, or in calling read_exec(). The former way uses the Linux demand loading mechanism to fault-in program pages when they're accessed, while the latter way is used when memory mapping is not supported by the host filesystem (for example the "msdos" filesystem).

Late 1.1 kernels embed a revised msdos filesystem, which supports mmap(). Moreover, the struct linux_binfmt is a linked list rather than an array, to allow loading a new binary format as a kernel module. Finally, the structure itself has been extended to access format-related core-dump routines.

B.6 Accessing filesystems

It is well known that the filesystem is the most basic resource in a UN⋆X system, so basic and ubiquitous that it needs a more handy name — I'll stick to the standard practice of calling it simply "fs".

I'll assume the reader already knows the basic UN⋆X fs ideas — access permissions, inodes, the superblock, mounting and umounting. Those concepts are well explained by smarter authors than me within the standard UN⋆X literature, so I won't duplicate their efforts and I'll stick to Linux specific issues.

While the first Unices used to support a single fs type, whose structure was widespread in the whole kernel, today's practice is to use a standardized interface between the kernel and the fs, in order to ease data interchange across architectures. Linux itself provides a standardized layer to pass information between the kernel and each fs module. This interface layer is called VFS, for "virtual filesystem".

Filesystem code is therefore split into two layers: the upper layer is concerned with the management of kernel tables and data structures, while the lower layer is made up of the set of fs-dependent functions, and is invoked through the VFS data structures.

All the fs-independent material resides in the fs/*.c files. They address the following issues:

- Managing the buffer chache (buffer.c);
- Responding to the fcntl() and ioctl() system calls (fcntl.c and ioctl.c);
- Mapping pipes and fifos on inodes and buffers (fifo.c, pipe.c);
- Managing file- and inode-tables (file_table.c, inode.c);
- Locking and unlocking files and records (locks.c);
- Mapping names to inodes (namei.c, open.c);
- Implementing the tricky select() function (select.c);
- Providing information (stat.c);
- mounting and umounting filesystems (super.c);
- exec()ing executables and dumping cores (exec.c);
- Loading the various binary formats (bin_fmt*.c, as outlined above).

The VFS interface, then, consists of a set of relatively high-level operations which are invoked from the fs-independent code and are actually performed by each filesystem type. The most relevant structures are inode_operations and file_operations, though they're not alone: other structures exist as well. All of them are defined within include/linux/fs.h.

The kernel entry point to the actual file system is the structure file_system_type. An array of file_system_types is embodied within fs/filesystems.c and it is referenced whenever a mount is issued. The function read_super for the relevant fs type is then in charge of filling a struct super_block item, which in turn embeds a struct super_operations and a struct *type*_sb_info. The former provides pointers to generic fs operations for the current fs-type, the latter embeds specific information for the fs-type.

> The array of filesystem types has been turned in a linked list, to allow loading new fs types as kernel modules. The function (un-)register_filesystem is coded within fs/super.c.

B.7 Quick Anatomy of a Filesystem Type

The role of a filesystem type is to perform the low-level tasks used to map the relatively high level VFS operations on the physical media (disks, network or whatever). The VFS interface is flexible enough to allow support for both conventional UN⋆X filesystems and exotic situations such as the `msdos` and `umsdos` types.

Each fs-type is made up of the following items, in addition to its own directory:

- An entry in the `file_systems[]` array (`fs/filesystems.c`);
- The superblock include file (`include/linux/`*type*`_fs_sb.h`);
- The inode include file (`include/linux/`*type*`_fs_i.h`);
- The generic own include file (`include/linux/`*type*`_fs.h`);
- Two `#include` lines within `include/linux/fs.h`, as well as the entries in `struct super_block` and `struct inode`.

The own directory for the fs type contains all the real code, responsible of inode and data management.

> The chapter about `procfs` in this guide uncovers all the details about low-level code and VFS interface for that fs type. Source code in `fs/procfs` is quite understandable after reading the chapter.

We'll now look at the internal workings of the VFS mechanism, and the `minix` filesystem source is used as a working example. I chose the `minix` type because it is small but complete; moreover, any other fs type in Linux derives from the `minix` one. The `ext2` type, the de-facto standard in recent Linux installations, is much more complex than that and its exploration is left as an exercise for the smart reader.

When a minix-fs is mounted, `minix_read_super` fills the `super_block` structure with data read from the mounted device. The `s_op` field of the structure will then hold a pointer to `minix_sops`, which is used by the generic filesystem code to dispatch superblock operations.

Chaining the newly mounted fs in the global system tree relies on the following data items (assuming `sb` is the `super_block` structure and `dir_i` points to the inode for the mount point):

- `sb->s_mounted` points to the root-dir inode of the mounted filesystem (`MINIX_ROOT_INO`);
- `dir_i->i_mount` holds `sb->s_mounted`;
- `sb->s_covered` holds `dir_i`

Umounting will eventually be performed by do_umount, which in turn invokes minix_put_super.

Whenever a file is accessed, minix_read_inode comes into play; it fills the system-wide inode structure with fields coming form minix_inode. The inode->i_op field is filled according to inode->i_mode and it is responsible for any further operation on the file. The source for the minix functions just described are to be found in fs/minix/inode.c.

The inode_operations structure is used to dispatch inode operations (you guessed it) to the fs-type specific kernel functions; the first entry in the structure is a pointer to a file_operations item, which is the data-management equivalent of i_op. The minix fs-type allows three instances of inode-operation sets (for direcotries, for files and for symbolic links) and two instances of file-operation sets (symlinks don't need one).

Directory operations (minix_readdir alone) are to be found in fs/minix/dir.c; file operations (read and write) appear within fs/minix/file.c and symlink operations (reading and following the link) in fs/minix/symlink.c.

The rest of the minix directory implements the following tasks:

- bitmap.c manages allocation and freeing of inodes and blocks (the ext2 fs, otherwise, has two different source files);
- fsynk.c is responsible for the fsync() system calls — it manages direct, indirect and double indirect blocks (I assume you know about them, it's common UN⋆X knowledge);
- namei.c embeds all the name-related inode operations, such as creating and destroying nodes, renaming and linking;
- truncate.c performs truncation of files.

B.8 The console driver

Being the main I/O device on most Linux boxes, the console driver deserves some attention. The source code related to the console, as well as the other character drivers, is to be found in drivers/char, and we'll use this very directory as our referenece point when naming files.

Console initialization is performed by the function tty_init(), in tty_io.c. This function is only concerned in getting major device numbers and calling the init function for each device set. con_init(), then is the one related to the console, and resides in console.c.

Initialization of the console has changed quite a lot during 1.1 evolution. `console_init()` has been detatched from `tty_init()`, and is called directly by `../../main.c`. The virtual consoles are now dynamically allocated, and quite a good deal of code has changed. So, I'll skip the details of initialization, allocation and such.

B.8.1 How file operations are dispatched to the console

This paragraph is quite low-level, and can be happily skipped over.

Needless to say, a UN⋆X device is accessed though the filesystem. This paragraph details all steps from the device file to the actual console functions. Moreover, the following information is extracted from the 1.1.73 source code, and it may be slightly different from the 1.0 source.

When a device inode is opened, the function `chrdev_open()` (or `blkdev_open()`, but we'll stich to character devices) in ../../fs/devices.c gets executed. This function is reached by means of the structure `def_chr_fops`, which in turn is referenced by `chrdev_inode_operations`, used by all the filesystem types (see the previous section about filesystems).

`chrdev_open` takes care of specifying the device operations by substituting the device specific `file_operations` table in the current `filp` and calls the specific `open()`. Device specific tables are kept in the array `chrdevs[]`, indexed by the majour device number, and filled by the same `../../fs/devices.c`.

If the device is a tty one (aren't we aiming at the console?), we come to the tty drivers, whose functions are in `tty_io.c`, indexed by `tty_fops`. Thus, `tty_open()` calls `init_dev()`, which allocates any data structure needed by the device, based on the minor device number.

The minor number is also used to retrieve the actual driver for the device, which has been registered through `tty_register_driver()`. The driver, then, is still another structure used to dispatch computation, just like `file_ops`; it is concerned with writing and controlling the device. The last data structure used in managing a tty is the line discipline, described later. The line discipline for the console (and any other tty device) is set by `initialize_tty_struct()`, invoked by `init_dev`.

Everything we touched in this paragraph is device-independent. The only console-specific particular is that `console.c`, has registered its own driver during `con_init()`. The line discipline, on the contrary, in independent of the device.

The `tty_driver` structure is fully explained within. `<linux/tty_driver.h>`

The above information has been extracted from 1.1.73 source code. It isn't unlikely for your kernel to be somewhat different ("This information is subject to change without notice").

B.8.2 Writing to the console

When a console device is written to, the function con_write gets invoked. This function manages all the control characters and escape sequences used to provide applications with complete screen management. The escape sequences implemented are those of the vt102 terminal; This means that your environment should say TERM=vt102 when you are telnetting to a non-Linux host; the best choice for local activities, however, is TERM=console because the Linux console offers a superset of vt102 functionality.

con_write(), thus, is mostly made up of nested switch statements, used to handle a finite state automaton interpreting escape sequences one character at a time. When in normal mode, the character being printed is written directly to the video memory, using the current attr-ibute. Within console.c, all the fields of struct vc are made accessible through macros, so any reference to (for example) attr, does actually refer to the field in the structure vc_cons[currcons], as long as currcons is the number of the console being referred to.

Actually, vc_cons in newer kernels is no longer an array of structures , it now is an array of pointers whose contents are kmalloc()ed. The use of macros greatly simplified changing the approach, because much of the code didn't need to be rewritten.

Actual mapping and unmapping of the console memory to screen is performed by the functions set_scrmem() (which copies data from the console buffer to video memory) and get_scrmem (which copies back data to the console buffer). The private buffer of the current console is physically mapped on the actual video RAM, in order to minimize the number of data transfers. This means that get- and set- -_scrmem() are static to console.c and are called only during a console switch.

B.8.3 Reading the console

Reading the console is accomplished through the line-discipline. The default (and unique) line discipline in Linux is called tty_ldisc_N_TTY. The line discipline is what "disciplines input through a line". It is another function table (we're used to the approach, aren't we?), which is concerned with reading the device. With the help of termios flags, the line discipline is what controls input from the tty: raw, cbreak and cooked mode; select(); ioctl() and so on.

The read function in the line discipline is called read_chan(), which reads the tty buffer independently of whence it came from. The reason is that character arrival through a tty is managed by asynchronous hardware interrupts.

The line discipline N_TTY is to be found in the same tty_io.c, though later kernels use a different n_tty.c source file.

The lowest level of console input is part of keyboard management, and thus it is handled within `keyboard.c`, in the function `keyboard_interrupt()`.

B.8.4 Keyboard management

Keyboard management is quite a nightmare. It is confined to the file `keyboard.c`, which is full of hexadecimal numbers to represent the various keycodes appearing in keyboards of different manifacturers.

I won't dig in k̑eyboard.c, because no relevant information is there to the kernel hacker.

> For those readers who are really interested in the Linux keyboard, the best approach to `keyboard.c` is from the last line upward. Lowest level details occur mainly in the first half of the file.

B.8.5 Switching the current console

The current console is switched through invocation of the function `change_console()`, which resides in `tty_io.c` and is invoked by both `keyboard.c` and `vt.c` (the former switches console in response to keypresses, the latter when a program requests it by invoking an `ioctl()` call).

The actual switching process is performed in two steps, and the function `complete_change_console()` takes care of the second part of it. Splitting the switch is meant to complete the task after a possible handshake with the process controlling the tty we're leaving. If the console is not under process control, `change_console()` calls `complete_change_console()` by itself. Process intervertion is needed to successfully switch from a graphic console to a text one and viceversa, and the X server (for example) is the controlling process of its own graphic console.

B.8.6 The selection mechanism

"`selection`" is the cut and paste facility for the Linux text consoles. The mechanism is mainly handled by a user-level process, which can be instantiated by either `selection` or `gpm`. The user-level program uses `ioctl()` on the console to tell the kernel to highlight a region of the screen. The selected text, then, is copied to a selection buffer. The buffer is a static entity in `console.c`. Pasting text is accomplished by 'manually' pushing characters in the tty input queue. The whole selection mechanism is protected by `#ifdef` so users can disable it during kernel configuration to save a few kilobytes of ram.

Selection is a very-low-level facility, and its workings are hidden from any other kernel activity. This means that most #ifdef's simply deals with removing the highlight before the screen is modified in any way.

> Newer kernels feature improved code for selection, and the mouse pointer can be highlighted independently of the selected text (1.1.32 and later). Moreover, from 1.1.73 onward a dynamic buffer is used for selected text rather than a static one, making the kernel 4kB smaller.

B.8.7 ioctl()ling the device

The ioctl() system call is the entry point for user processes to control the behaviour of device files. Ioctl management is spawned by ../../fs/ioctl.c, where the real sys_ioctl() resides. The standard ioctl requests are performed right there, other file-related requests are processed by file_ioctl() (same source file), while any other request is dispatches to the device-specific ioctl() function.

The ioctl material for console devices resides in vt.c, because the console driver dispatches ioctl requests to vt_ioctl().

> The information above refer to 1.1.7x. The 1.0 kernel doesn't have the "driver" table, and vt_ioctl() is pointed to directly by the file_operations() table.

Ioctl material is quite confused, indeed. Some requests are related to the device, and some are related to the line discipline. I'll try to summarize things for the 1.0 and the 1.1.7x kernels. Anything happened in between.

The 1.1.7x series features the following approach: tty_ioctl.c implements only line discipline requests (namely n_tty_ioctl(), which is the only n_tty function outside of n_tty.c), while the file_operations field points to tty_ioctl() in tty_io.c. If the request number is not resolved by tty_ioctl(), it is passed along to tty->driver.ioctl or, if it fails, to tty->ldisc.ioctl. Driver-related stuff for the console it to be found in vt.c, while line discipline material is in tty_ioctl.c.

In the 1.0 kernel, tty_ioctl() is in tty_ioctl.c and is pointed to by generic tty file_operations. Unresolved requests are passed along to the specific ioctl function or to the line-discipline code, in a way similar to 1.1.7x.

Note that in both cases, the TIOCLINUX request is in the device-independent code. This implies that the console selection can be set by ioctlling any tty (set_selection() always operates on the foreground console), and this is a security hole. It is also a good reason to switch to a newer kernel, where the problem is fixed by only allowing the superuser to handle the selection.

A variety of requests can be issued to the console device, and the best way to know about them is to browse the source file vt.c.

Appendix C

The GNU General Public License

Printed below is the GNU General Public License (the *GPL* or *copyleft*), under which Linux is licensed. It is reproduced here to clear up some of the confusion about Linux's copyright status — Linux is *not* shareware, and it is *not* in the public domain. The bulk of the Linux kernel is copyright © 1993 by Linus Torvalds, and other software and parts of the kernel are copyrighted by their authors. Thus, Linux *is* copyrighted, however, you may redistribute it under the terms of the GPL printed below.

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

C.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

C.2 Terms and Conditions for Copying, Distribution, and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this

License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

    a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

    b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

    c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

    The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

    If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or

distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

C.3 Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> ⟨*one line to give the program's name and a brief idea of what it does.*⟩ Copyright © 19yy ⟨*name of author*⟩
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author Gnomovision
comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.  This is
free software, and you are welcome to redistribute it under certain
conditions; type ‘show c’ for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items — whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.
>
> ⟨*signature of Ty Coon*⟩, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.