How Operating Systems Work

Fall 2020 edition

Peter Desnoyers
Northeastern University
Fall 2020

First printing: 2018

ISBN: 978-0-359-05752-8

Dedication

To Elizabeth, Christopher, and Madeline

Contents

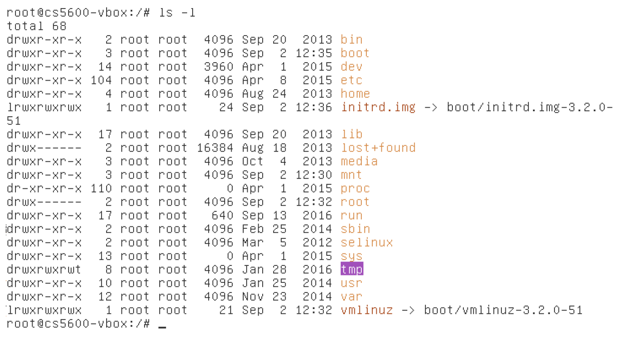
Acknowledgements

This book is based on CS 5600 at Northeastern University as I have taught it since 2008, and would not be possible without the efforts of numerous teaching assistants who have helped run this class throughout this time. In addition, Kim Prokosch of Northeastern's NUOnline was extremely helpful in getting this material on paper for the online version of this class which was taught in Fall 2013.

Chapter 1

Introduction

This textbook is for the CS5600 class in the Northeastern College of Computer and Information Science, based on the design of CS5600 starting in Fall of 2008. The goal of this class is not to teach you how to write an operating system—that is an obscure skill, practiced by far fewer people than you might think. Nor is it to learn how to use an operating system—depending on the type of use, that would be system administration, programming, or just using a computer. Instead the goal is to teach you how computers work, by describing the interacting parts underneath the user and programming interfaces.



```
root@cs5600-vbox:/# ls -l
total 68
drwxr-xr-x   2 root root  4096 Sep 20  2013 bin
drwxr-xr-x   3 root root  4096 Sep  2 12:35 boot
drwxr-xr-x  14 root root  3960 Apr  1  2015 dev
drwxr-xr-x 104 root root  4096 Apr  8  2015 etc
drwxr-xr-x   4 root root  4096 Aug 24  2013 home
lrwxrwxrwx   1 root root    24 Sep  2 12:36 initrd.img -> boot/initrd.img-3.2.0-
51
drwxr-xr-x  17 root root  4096 Sep 20  2013 lib
drwx------   2 root root 16384 Aug 18  2013 lost+found
drwxr-xr-x   3 root root  4096 Oct  4  2013 media
drwxr-xr-x   3 root root  4096 Sep  2 12:30 mnt
dr-xr-xr-x 110 root root     0 Apr  1  2015 proc
drwx------   2 root root  4096 Sep  2 12:32 root
drwxr-xr-x  17 root root   640 Sep 13  2016 run
drwxr-xr-x   2 root root  4096 Feb 25  2014 sbin
drwxr-xr-x   2 root root  4096 Mar  5  2012 selinux
drwxr-xr-x  13 root root     0 Apr  1  2015 sys
drwxrwxrwt   8 root root  4096 Jan 28  2016 tmp
drwxr-xr-x  10 root root  4096 Jan 25  2014 usr
drwxr-xr-x  12 root root  4096 Nov 23  2014 var
lrwxrwxrwx   1 root root    21 Sep  2 12:32 vmlinuz -> boot/vmlinuz-3.2.0-51
root@cs5600-vbox:/# _
```

Figure 1.1: Linux text console with simple command.

For an example of what this means, consider running a simple command such as `ls` on a Linux system. In Figure 1.1 we see the screen of a system booted in text mode, using the simple character display that the BIOS uses. In responding to the keystrokes typed by the user, we can identify not only the basic actions being performed (“run the `ls -l` command with output to the console") but a large number of interacting actions and components as well:

- the keyboard control hardware (assuming an old-fashioned PS/2 keyboard) interrupts the processor, causing it to run a portion of the keyboard input driver.
- The driver reads data from the keyboard and calls scheduling functions to wake the shell process, which was sleeping waiting for input.
- the shell process spawns a copy of itself, by invoking a system call which copies some of the shell process state and shares other parts of it between the *parent* and *child* processes using the virtual memory system.
- The new process invokes the `exec` system call, causing the operating system to map the `/bin/ls` binary into the process address space.
- As `ls` starts up, the dynamic loader loads additional shared libraries into the process address space; these as well as the `ls` code itself is loaded into memory on demand as the CPU accesses them.
- `ls` invokes system calls to read the list of files in the current directory.
- The file system code receives requests to read files containing the executable and libraries, as well as the directory listing request from the `ls` program itself, and in turn requests data from the disk (via the block device system) to fulfill these requests.
- Since the example was actually running in a virtual machine, not a physical machine[1] the hardware interactions described above were actually emulated by another software system (i.e. VirtualBox) which translated them into requests to the underlying operating system, which in turn interacted with the real keyboard and screen.

The remainder of this book, and the corresponding class, is concerned with the detailed analysis of the interactions involved in performing this simple operation. The major sections of this text concern:

OS organization: Memory organization and OS interface to decouple applications from hardware and OS details, context switching, and system calls. This section describes and uses a simple computer, described more fully in the appendices.

Synchronization: Beginning with practical problems arising from multiple simultaneous actions, we describe methods such as semaphores and monitors to control simultaneous actions, as well as methods to reason about the operation and performance of parallel operations.

[1] It makes screenshots far easier.

Virtual memory: At the hardware level, how is address translation implemented via the MMU, TLB, and page table? In the OS, how are page faults used to implement copy-on-write, demand loading, and paged virtual memory?

Block devices: These are devices such as disks, RAID arrays, and SSDs, used for storing files and similar information. Topics covered include performance and interfaces, I/O operation at a hardware level, and methods of structuring I/O systems for reliability (RAID), manageability (logical volume management) and efficiency (deduplication).

File systems: What is a file system and what are its operations? How do we implement these, and how do we lay files out on disk?

Security: What are the goals of security mechanisms in an operating system? How can we specify and implement policies to control access and operations?

The objective of the class is to be able to identify the steps involved in this and other computer operations. In learning this we will touch on hardware, device drivers, scheduling, virtual memory, and networking. We focus on *behavior*—i.e. the sequence of events which occurs in response to an input, and results in an output. This behavior cuts across layers and subsystems, as an event at the hardware level may trigger actions within a device driver, then in the core of the operating system, within a user process, etc. Rather than looking at the operating system in a structured way we are going to follow these sequences of behavior and see where they lead.

Chapter 2

Program and OS Organization

This chapter begins by defining a very simple computer, with assembly language instructions, a 16-bit address space, and memory-mapped peripherals.[1] We will use this computer as an example as we talk about the simplest operating systems.

We then examine simple methods of organizing and running a program on this computer. We extend these methods to hide hardware dependencies, insulate against changes in operating system details, and allow for program loading and execution—at this point we have achieved a simple single-user OS, similar in many ways to MSDOS 1.0.

After this we examine multi-processing and context switching, allowing multiple programs to be running simultaneously. Finally we examine what additional features are needed to protect the operating system from the user, and users from each other. At this point we have achieved a simplified version of a modern operating system; we compare it to Linux and Windows.

[1] In other words, CPU operations only read or write internal registers and external (to the CPU) memory. The memory address space is partitioned between normal random-access memory and a section devoted to I/O devices, which respond to read and write requests to particular addresses.

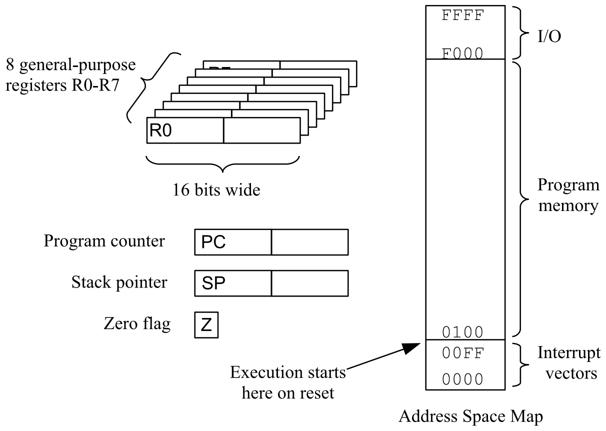


Figure 2.1: Simple computer system architecture

2.1 A Simple Computer

We use a fictional 16-bit computer, shown in Figure 2.1. It has 8 general-purpose registers, R0-R7, holding 16 bits each, as well as a stack pointer (SP) and program counter (PC), and 64 KB (2^{16}) of memory which may be accessed as 8-bit bytes or 16-bit words.

The examples below use the following instructions:

1. LOAD.B, LOAD.W - load a byte or a word from the indicated address, which may be an absolute address (i.e. a number) or contained in a register.
2. LOAD.I - load a constant value into a register. (called an "immediate" value for unknown reasons)
3. STORE.B, STORE.W - store a byte or word from a register into memory.
4. MOV - copy the contents of one register to another.
5. ADD, SUB - add or subtract one register (or a constant value) to or from another register. Sets the Z flag if the result is zero.
6. CMP - compare a register to another register or a constant value. Subtracts the second value from the register, sets the Z flag appropriately, and then throws away the result.
7. JMP - jump to the indicated address.
8. JMP_Z, JMP_NZ - jump if the Z flag is set (Z) or not set (NZ)
9. PUSH - push the 16-bit value in the indicated register onto the stack
10. POP - pop the 16-bit value top of the stack and place in the indicated

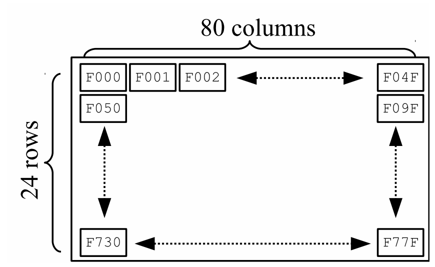


Figure 2.2: Frame buffer

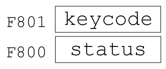


Figure 2.3: Keyboard controller

register.

11. CALL - call a subroutine by pushing the *return address* (i.e. the address of the next instruction) onto the stack and jumping to the indicated address.
12. RET - return from subroutine by popping the return address from the top of the stack and jumping to it.

In addition there are several input/output devices which are *memory-mapped*—particular memory addresses correspond to registers in these devices, rather than normal memory, and reads or writes to these addresses are used to operate the device. These devices include:

1. *frame buffer*: A region of 1920 bytes, corresponding to 24 lines of 80 characters displayed on a video display. Writing a byte to one of these locations causes the indicated character to be displayed at the corresponding location on the screen, as shown in Figure 2.2.
2. *keyboard controller*: Two registers, one indicating whether a key has been pressed, and the other the character corresponding to that key, as shown in Figure 2.3.

This description is enough for our first examples; a full specification is found in Appendix A.

Review Questions

2.1.1. I/O devices are pieces of software that are part of the operating system: *yes / no / sort of*

2.1.2. I/O devices are part of memory: *yes / no / sort of*

```
;; note - frame buffer starts at 0xF000
str:    "Hello World"

begin:  LOAD.I R1 ← &str
        LOAD.I R2 ← 11
        LOAD.I R3 ← 0xF000

loop:   LOAD.B R4 ← *(R1++)
        STORE.B R4 → *(R3++)
        SUB R2-1 → R2
        JMP_NZ loop

done:   JMP done
```

Figure 2.4: Simple 'Hello World' program. LOAD.I loads an immediate (i.e. constant) value, LOAD/STORE.B operates on a single byte instead of a 16-bit word.

```
;; keyboard status = 0xF800, keycode = 0xF801

begin:   LOAD.I R1 ← 0xF000 ;; frame buffer

loop:    LOAD.B R2 ← *(0xF800)
         TEST R2
         JMP_Z loop

         LOAD.B R2 ← *(0xF801) ;; get keystroke
         STOR.B R2 → *(R1++) ;; copy to frame buffer

         JMP loop
```

Figure 2.5: Copy keystrokes to screen

2.2 Program Organization

Our first program is seen in Figure 2.4. It performs a very simple task, copying bytes from a compiled-in string to the frame buffer to display (of course) "Hello World" and then finishing in a loop which does nothing. (Although the reader is not expected to write programs in assembly language, we assume that given the computer definition you should be able to decipher simple examples such as this.)

In Figure 2.5 we see another simple program, which performs input as well as output. In the three lines starting at the label `loop` it polls the keyboard status register, waiting for a key to be pressed. It then reads the keystroke value into R4 and stores it into the frame buffer. (Well, at least for the first 1920 keystrokes. It will advance through the frame buffer line by line, ignoring carriage returns, and eventually "fall off" the end and start scribbling over the rest of the I/O space. It is a very simple program.)

These two programs illustrate the simplest sort of software organization, consisting only of the program itself, which handles every detail including the hardware interface—not a difficult task for such a simple case. All there is here is a program and some hardware, with nothing that we can identify as an operating system; this approach might be appropriate for the smallest microcontrollers. (i.e. with a few hundred bytes of program memory and even less data memory)

2.3 A Simple Operating System Interface

> *Operating system* - software that isn't the program itself, especially that required by a user or program to interact with (i.e. *operate*) the computer.

For even slightly complex programs we are going to want to factor out the hardware interface functionality. This would e.g. allow us to use a single function for output to the frame buffer, which could be called from different places in the program. Our next program, in Figure 2.6, copies keystrokes from the keyboard to the frame buffer just like our previous one. However, in this case we have separated out the keyboard and display interface functions. With this we start to see the beginnings of an operating system.

One goal of an operating system is to provide an abstract interface to the hardware, serving several purposes. First, it allows a program developed for one computer to be used on another one without extensive modification, even if the hardware is not exactly the same. In addition, by separating program-specific and hardware-specific code, it makes it easier for each to be developed by someone who is expert in the corresponding area.[2]

Figure 2.6 might be termed a *library operating system*—it consists of a series of functions which are linked with the application, creating a single program which is loaded onto the hardware, frequently by being programmed into read-only-memory and thus being present when the computer is first turned on.

[2]Multiple levels of such separation are seen in modern computers, where BIOS and hardware drivers are written by different organizations, each knowledgeable about their own hardware, and hiding the details and complications of these devices behind an abstract interface.

Although this approach is useful for single-purpose devices, it has a key shortcoming for general-purpose computers, in that changing the program requires changing the entire contents of memory, requiring a mechanism outside of the OS and program we have described so far. In some cases, in fact, the only way to replace the program is to buy a new device—this may in fact be reasonable for sufficiently "dumb" devices (e.g. a microwave oven) but is clearly not going to be a popular way to get a new program onto a computer.

2.4 Program Loading

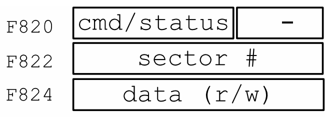


Figure 2.7: Simple disk controller

In order to load programs we need a device to load them from—in this case a disk drive, which (unlike memory) maintains its data while powered off, and is typically much larger than memory, allowing it to hold multiple programs. Data on a disk drive is organized in 512-byte blocks, which are identified by block number, starting with 0. In Figure 2.7 we see an extremely simple disk controller, which allows a single block to be read from or written to the disk[3]. Operation is

```
loop:     CALL getkey     ;; return value in R0
          PUSH R0         ;; push argument
          CALL putchar
          POP R0          ;; to balance stack
          JMP loop

getkey:   LOAD.B R4 ← *(0xF800) ;; key ready reg.
          CMP   R4, 0
          JMP_Z getkey
          LOAD.B R0 ← *(0xF801) ;; key code reg.
          RET

putchar: LOAD.B R0 ← *(SP+2) ;; fetch arg into R0
          LOAD.W R1 ← *(bufptr)
          STOR.B R0 → *(R1)  ;; *bufptr = R0
          ADD   R1+1 → R1
          STOR.W R1 → bufptr ;; bufptr++
          RET

bufptr:   word 0xF000     ;; frame buffer pointer
```

Figure 2.6: Copy keystrokes with factored input/output

[3]For more information on disk drives, see Section 5.3 in Chapter 5.

as follows:

To write 512 bytes to block B:

1. Write 256 16-byte words (e.g. copying from a buffer), one word at a time, to the disk controller data register (0xF824)
2. Write block address (B) to block address register (0xF822)
3. Write command byte (2=WRITE) to cmd/status register (address 0xF820)
4. Poll cmd/status register; its value will change from 2 to 0 to indicate transfer is complete.

To read from block B:

1. Write block address (B) to block address register (0xF822)
2. Write command byte (1=READ) to cmd/status register (0xF820)
3. Poll cmd/status register; value changes from 1 to 0 to indicate data is ready to read
4. Read 256 16-bit words from data register (0xF824), typically into a buffer in memory.

Now that we have a device to load programs from, the next step is to reserve separate portions of the address space for the OS and program, as shown in Figure 2.8, so that we have a place in memory to load those programs into. The program links against the OS as before, but this time the OS is located in a separate memory region, so different programs (each compiled and linked against this same instance of the OS) may be loaded and run at different times.

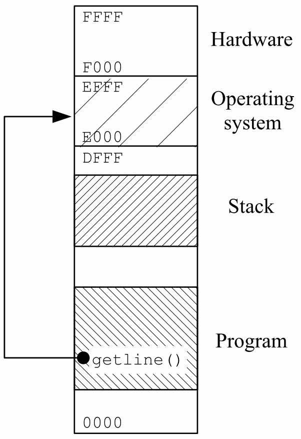


Figure 2.8: Split OS/program memory map

In Figure 2.9 we see pseudo-code[4] for a simple and user-hostile command-line interface for this OS. The user specifies a disk address and length; the OS loads a program from the specified disk location into a standard address in memory and transfers control to that address. When the program is finished it returns control to the OS command line loop, which is then able to load and run a different program.

[4]A generic term for anything that isn't real program code, but which you are supposed to understand anyway.

```
CMD_LOOP:
      line = GET_LINE()
      if line starts with "load":
         blk,count = parse(line)
         load_disk_sectors(_PROGRAM_BASE, blk, count)
      if line starts with "go":
         call _PROGRAM_BASE
      jmp CMD_LOOP
```

Figure 2.9: Simple command line and program loader. Commands are "load <start blk#> <count>" and "go"

There are a number of limitations to this operating system:

1. It's not robust: if it doesn't find the program you specified, it crashes.
2. If the program crashes, the entire system has to be reset (or power cycled) before another program can be loaded.
3. The program may not run on another machine, or on the same machine after an OS upgrade.

Problem 1 can be fixed fairly easily; for instance if we have a simple file system, and specify the file by name, then if the file isn't found the OS can print an error message and ask for another command. Problem 2 may be annoying, but it didn't prevent MS-DOS from being the most widely-used operating system for many years[5]. Problem 3 is an issue, though, although first we have to describe why it is the case.

In particular, this operating system requires a certain amount of coordination between the OS and the program: (a) The OS must know at what address the program expects to begin execution—e.g. the `main()` function in a C program or its equivalent. This isn't too much of an issue, as the OS authors can just tell the application (and compiler) writers what to do. (e.g. in our case execution begins at the very beginning of the program in memory) And (b) the program, in turn, must have the correct addresses for any of the OS functions (e.g. `getkey` in 2.6) which it invokes.

This is where the problem lies. The location of these entry points may vary from machine to machine due to e.g. different memory sizes, and will almost certainly change across versions of the OS as code is added (or occasionally removed) from some of its functions.

To work around this we typically define a standard set of entry points into the OS, or *system calls*, access these entry points via a table which

[5]In that case it typically wasn't necessary to turn off the power - the low-level keyboard driver would reset the machine when it saw CTL-ALT-DEL pressed at the same time.

is placed in a fixed location in memory (e.g. at address 0), and give each system call a specific place in this table.

One way of implementing this is for the program to access this table directly; thus if `getkey` is entry 2, programs could invoke it via the call `syscall_table[2](args)`. Alternately, many CPUs define a TRAP or INT[6] instruction which may be used for this purpose. In this case, the table will be located in a location known to the CPU (either fixed, as in the original 8088 where the table began at address 0, or identified by a control register) and TRAP N will cause the CPU to perform a function call to the N^{th} entry of this table.

We now have an interface which allows the OS to provide services to a program via a fixed interface, allowing for binary compatibility across different hardware platforms and OS versions. If we use a TRAP instruction for this interface, we have a system similar to MS-DOS, where OS and application were each given separate parts of a single address space, and access to generic as well as hardware-specific OS functions was performed via the x86 INT instruction.

Review Questions

2.4.1. Does an operating system handle hardware details for a program?
yes/no/maybe

2.4.2. Does an operating system have a graphical user interface?
yes / no / maybe

2.4.3. Does an operating system allow the user to load and run programs?
yes / no / maybe

2.4.4. Does the system call table change every time a program is compiled?
yes / no

[6] the x86 “interrupt” instruction.

Comparison to MS-DOS 1.0

This simple OS is very similar to the first version of MS-DOS. In MS-DOS 1.0, as seen in Figure 2.10, the operating system is split into 4 parts: a hardware-specific I/O system (BIOS), MS-DOS itself, the resident part of the command line interpreter, and additional "transient" parts of the command interpreter which could be over-written by larger programs (especially on machines with 16KB RAM) and re-loaded from floppy disk after the program exited.

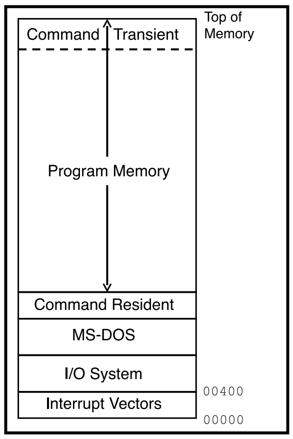


Figure adapted from *An Inside Look at MS-DOS*, Tim Paterson, Byte Magazine, 1983

Figure 2.10: MS-DOS layout

Similarities with the simple OS include:

1. separate OS and program memory regions
2. a system call table accessed via INT instruction
3. a command line which is part of the OS
4. a keyboard controller, frame buffer, and disk controller which are much like the CPU-5600 versions

2.5 Device Virtualization

The `GET_LINE` and `getkey` operations just discussed are simple examples of a powerful operating system concept—*device virtualization*. Rather than requiring the programmer to write code specific to a particular hardware implementation of a keyboard controller, the operating system provides simple "virtual devices" to the program, while the hardware details are handled within the operating system. In particular, if these virtual devices are sufficiently generic (e.g. supporting only `read` and `write` operations) then the same program can read from the physical keyboard, from a window system which sends keyboard data to the currently active window, from a file, or from a network connection like `ssh`.

Implementing a generic I/O system like this is fairly straightforward, as the set of I/O operations (open, close, read, write, etc.) is basically an interface, while each particular device (e.g. keyboard, disk file, etc.) can be thought of as a class implementing that interface. In practice this is done by providing the program with a *handle* or *descriptor* which maps to the actual I/O object within the OS, and then implementing system calls

```
        struct f_op {
            size_t (*read) (struct file *, char *, size_t);
            size_t (*write) (struct file *, char *, size_t);
            ...
        };

        /* 'current' points to current process structure
         */
        size_t sys_read(int fd, char *buf, size_t count) {
            struct file *file = current->files[fd];
            return file->f_op->read(file, buf, count);
        }
```

Figure 2.11: Simplified code for `read` system call in Linux

such as `read` and `write` by mapping the handle to the object, and then invoking the appropriate method.

In Linux a file descriptor is an integer, used to index into a table of files opened by the current process; a simplified version of the read system call is seen in the example in Listing 2.11.[7] The listing is somewhat simplified—the actual code performs a few levels of indirection, some locking, and a bounds check while looking up the 'struct file' corresponding to 'fd', and also handles the offset within the file. The actual code is not that complex, however, as the complicated parts are all in the file system or device-specific `read` methods.

2.6 Address Space and Program Loading

Typically program address space is divided into the following parts: *code* or machine-language instructions (for some reason typically called "text"), *initialized data*, consisting of read-only and read-write initialized data, *initialized-zero data*, called "BSS" for obscure historical reasons, *heap* or dynamically allocated memory, and *stack*.

In Figure 2.12 we see the address space organization which has evolved for arranging these areas for CPUs on which the stack grows "down"—i.e. more recently pushed data is stored in lower-numbered addresses. (this is by far the most common arrangement) In this arrangement the fixed-sized portions of the address space are at the bottom, and the heap grows "up" from there, while the stack grows "down" from the highest available

[7]Like many other operating systems, Linux is written in C, which lacks direct support for abstract interfaces and data types; the actual implementation relies on a system of structures of function pointers which is similar to how the compiler implements virtual methods in C++.

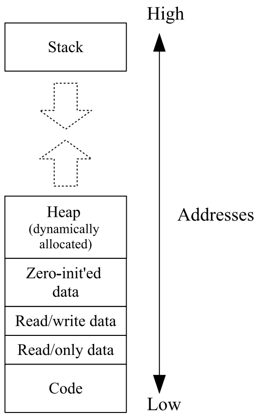


Figure 2.12: Typical process memory map: code, data, and heap at bottom; stack at top.

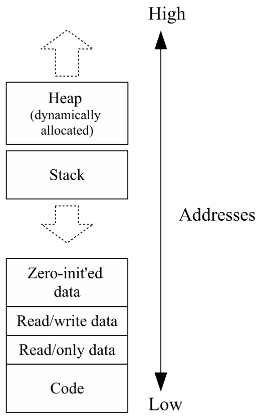


Figure 2.13: Awkward process memory map, with fixed-sized stack allocation.

address. Assuming that the memory available is contiguous, this gives the program maximum flexibility—it can use most of the memory for dynamically-allocated heap, or for the stack, as it chooses. In contrast, an organization such as Figure 2.13 would require a fixed allocation of the two regions to be made when the program is loaded by the OS, adding complexity while reducing flexibility. (Note that since the heap is software-managed it can grow in whatever direction we want; however on most CPUs the direction of stack growth is fixed.)

An additional goal of an address layout is to be able to accomodate different amounts of available memory. As an example, early microcomputers like the first IBM PCs might have between 16 KB and 64 KB of memory; we would like the same program to be able to run on machines with more or less memory, with the additional memory on the larger machine available for heap or stack. This was typically done by starting memory at address 0, so that a 16 KB machine would have available memory address 0x0000 through 0x3FFF, while a 32 K machine would be able to use 0x0000 through 0x7FFF. Code and fixed data would be located starting at a pre-defined offset near address 0, with stack and heap located above these sections, at addresses which might vary from machine to machine and program to program. This would ensure that small programs would be placed in low addresses, so that they would be guaranteed to run on low-memory machines, while the variability of stack and heap addresses was not a significant issue because the compiler does not need to generate

Index	Description	DOS name
0	divide by zero	
1	single step	
2	non-maskable	
3	debug break	
4	debug break on overflow	
5	-unused-	
6	invalid instr.	
7	-unused-	
8	system timer	IRQ0
9	keyboard input	IRQ1
10	line printer 2	IRQ2, LPT2
11	serial port 2	IRQ3, COM2
12	serial port 1	IRQ4, COM1
13	hard disk	IRQ5
14	floppy disk	IRQ6
15	line printer 1	IRQ7, LPT1
16-255	software-defined interrupts	

Table 2.1: 8086/8088 interrupts as defined by the IBM PC hardware.

direct references to them.

2.7 Interrupts

So far all the code that we have looked at has been *synchronous*, proceeding as a series of function calls reachable from some original point at which execution started. This is a good model for programs, but not always for operating systems, which may need to react to arbitrary asynchronous events. (Consider for instance trying to stop a program with control-C, if this only took effect when the program stopped and checked for it.)

To handle asynchronous I/O events, CPUs provide an *interrupt* mechanism. In response to a signal from an I/O device the CPU executes an *interrupt handler* function, returning to its current execution when the handler is done. The CPU essentially performs a forced function call, saving the address of the next instruction on the stack and jumping to the interrupt handler; the difference is that instead of doing this in response to a CALL instruction, it does it at some arbitrary time (but *between* two instructions) when the interrupt signal is asserted[8].

[8]This makes programming interrupt handlers quite tricky. Normally the compiler saves many register values before calling a function, and restores them afterwards; however an interrupt can occur anytime, and if it accidentally forgets to save a register and then modifies it, it will appear to the main program as if the register value changed spontaneously. This isn't good.

Most CPUs have several interrupt inputs; these correspond to an *interrupt vector table* in memory, either at a fixed location or identified by a special register, giving the addresses of the corresponding interrupt handlers. As an example, in Table 2.1 we see the corresponding table for an 8088 CPU as found in the original IBM PC, which provides handler addresses for external hardware interrupts as well as *exceptions* which halt normal program execution, such as dividing by zero or attempting to execute an illegal instruction.

The simplest interrupt-generating device is a *timer*, which does nothing except generate an interrupt at a periodic interval. In Listing 2.14 we see why it is called a timer—one of its most common uses is to keep track of time.

```
extern int time_in_ticks;
timer_interrupt_handler() {
    time_in_ticks++;
}
```

Figure 2.14: Simple timer interrupt handler

Another simple use for interrupts is for notification of keyboard input. Besides being useful for a “cancel” command like control-C, this is also very useful for *type-ahead*. On slower computers (e.g. the original IBM PC executed less than half a million instructions per second) a fast typist can hit multiple keys while a program is busy. A simple keyboard interface only holds one keystroke, causing additional ones to be lost. By using the keyboard interrupt, as shown in Figure 2.15, the operating system can read these keystrokes and save them, making them available to the program the next time it checks for input.

Review Questions

2.7.1. Hardware interrupts occur when particular instructions are executed: *yes / no*

A question for the reader - how would you change the one-key type-ahead buffer in Figure 2.15 to buffer a larger number of keystrokes?

```
int lastkey = -1; /* invalid keystroke */
kbd_interrupt() {
    lastkey = kbd_code;
}
int getkey() {
    while (lastkey == -1) {
        /* loop */
    }
    int tmp = lastkey;
    lastkey = -1;
    return tmp;
}
```

Figure 2.15: Single-key keyboard type-ahead buffer

2.7.2. A device (e.g. the keyboard controller) uses interrupts to send data to the CPU: *yes / no*

2.7.3. Interrupts allow a program to do multiple things at once: *yes / sort of / no*

2.8 Context Switching

Interrupt-driven type-ahead, as described above, represents a simple form of multi-processing, or handling multiple parallel operations on the same CPU. Full multi-processing, however, as found on modern operating systems, involves parallel execution of full programs, rather than merely interleaving a single program with specific bits of operating system functionality.

Our simple OS cannot do this, nor can MS-DOS (which it closely resembles), but it is a straightforward extension to do so even on limited hardware. To do this on a single CPU machine we need a mechanism for saving the state of a *process*—a running program—and restoring it after another process has taken its turn.

To do this we take advantage of the way in which program state is stored on the stack. This may be seen in Figure 2.16, where we see the stack frame generated by a call to function `g()` with arguments and local variables.

By holding arguments, return addresses, and local variables, the stack essentially captures all the private state of a running computation. If we were to save the stack of a running process, go off and do something else—taking care to use a different stack—and then switch stacks again to return to the first process, no one would be the wiser except for any delay incurred.

```
f() {
   g(4, 5);
}
g(int n, m) {
   int a = 10;
   ...
}
```

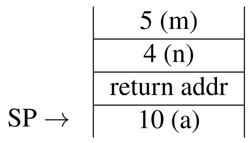


Figure 2.16: Subroutine call stack shown when in `g()`, called from `f()`, showing relationship between arguments, return address, and local variables.

```
sleep(time_t t) {
  end = now() + t;
  while (now() < end)
     do nothing;
}
```

```
sleep(time_t t) {
  ... switch() →     →
                   do something else
  ... return ←       ... for t seconds
}                     ← then return

      [process A]
                          [process B]
```

Figure 2.17: Alternate methods of implementing `sleep()`.

In fact, in Figure 2.17 we see two implementations of the `sleep()` function; the first busy-waits until the specified time has passed, while the second uses some mechanism to switch to another program for a while, and then returns when the interval is up. The particular mechanism used to switch from one process to another is simple but subtle: we save the processor registers by pushing them to the stack, and then save the value of the stack pointer into another location in memory. (This is commonly a location in a *process control block*, an object which represents the state of a process when another one is executing, and can be put on wait lists and otherwise manipulated.) We can then switch to another process by loading the stack pointer value for that second process (e.g. from its location in its process control block), restoring registers from the stack, and returning.

The flow of control involved in such a context switch is difficult to get used to, because the context switch itself *looks* like a simple function call, but *behaves* in a radically different way. In your previous classes you will have learned to think about functions as abstract operations, returning by definition to the same place where they were invoked. In a context switch, however, control enters the function from one location, and after a few simple instructions returns to an entirely different location.

We see different representations of this in Figures 2.18 and 2.19. The context switch code is shown first: it saves registers to process 1's stack and saves the value of the stack pointer, then loads process 2's stack pointer, pops saved registers, and returns. Note that the second half of the function is referring to an entirely different stack than the first half, so the registers and return address popped from the stack are different from the ones saved in the first half of the function.

In addition we see two different visualizations of the flow of control during context switch. In each case control enters `switch` via a call from one process (or *thread* of control) but exits by returning to a different process.

A context switch enters a process or thread by **returning** from a function call, and leaves the process by **calling** into the `switch` function.

This is a curious property of context switching: we can only switch *to* a process if we have switched *from* it at some point in the past. This results in a chicken-and-egg[9] sort of problem—how do we start a process in the first place? This is done via manipulating the stack "by hand" in the process creation code, making it look like a previous call was made to `switch`, with a return address pointing to the beginning of the code to be executed, forming what is called a *trampoline* which "bounces" back to the desired location.

In Figure 2.21 we see a thread being started so that it begins execution with the first instruction of function `main()`. Imagine that just before the beginning of `main()` there had been a call to `context_switch`; when that call returns execution will begin at address `main`. To start a thread

```
switch_1_2:
    PUSH R0   # save registers
    PUSH R1
    ...
    STOR SP -> proc1_sp
    LOAD SP <- proc2_sp
    ...
    POP R1
    POP R0    # restore them
    RET
```

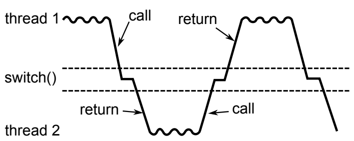


Figure 2.18: Different ways of looking at a context switch from Process 1 to Process 2.

[9]An English idiom referring to the rhetorical question "Which came first, the chicken or the egg?"

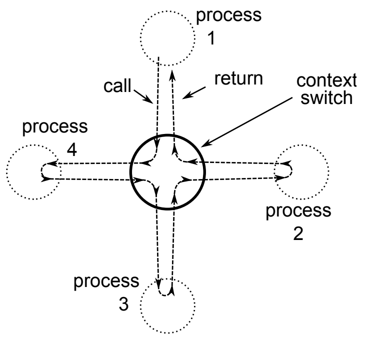


Figure 2.19: Another way of looking at context switch control flow—processes call into `switch` which then returns to another process.

```
_start() {
   /* prepare argc, argv */
   int val = main(argc, argv);
   exit(val);
   /* Not reached */
}
```

Figure 2.20: Simplified C run-time library (crt0.o) - invoke `main`, and then call `exit` to terminate process, guaranteeing no return from the true start function.

which will begin at `main`, then, we just fake this call stack; when we switch to the thread the first time, `context_switch` will then return to location `main`, where execution will begin.

A function is entered via `CALL` and exited via `RET`; similarly since we enter a process via `RET`, we exit it via `CALL`. In particular, we define a function (typically called `exit()`) which makes sure that the process will never be switched to again. (e.g. it is removed from any lists of processes to be run, its resources are freed, etc.) Note that some programming languages (e.g. C) allow process execution to be terminated by returning from the main function; this is done by calling `main` from the "real" start function, as shown in Figure 2.20.

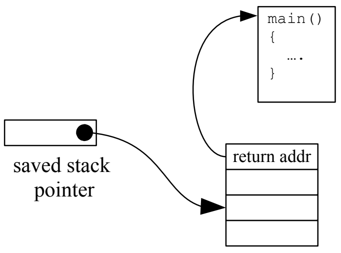


Figure 2.21: "Trampoline" return stack pointing to the beginning of the function to be executed (`main`)

Review Questions

2.8.1. Which of the following are stored on the stack?

a) Function arguments

b) Return addresses
c) Global variables
d) Local variables

2.8.2. The `RET` (return) instruction: a) Returns to the instruction immediately after `CALL` b) Returns to the address on the top of the stack.

2.8.3. When context switching from process A to process B, what CPU instruction actually jumps to code in B? (i.e. sets the PC to an address that is part of B's execution) : `CALL` / `JMP` / `RET`

2.9 Advanced Context Switching

So far we have considered the case where switching between processes is initiated by an explicit call into the OS from the currently running process. But an interrupt is essentially a function call from the current process into a part of the operating system—the interrupt handler—and we can in fact context switch to another process from within the interrupt handler function.[10] A simple example is the case of the timer interrupt, which can easily be used to implement *time slicing* between multiple processes. If the timer device was set to interrupt every e.g. 20 ms, and its interrupt handler did nothing except context switch to the next in a circular list of processes, then these processes would share the CPU in 20 ms slices.

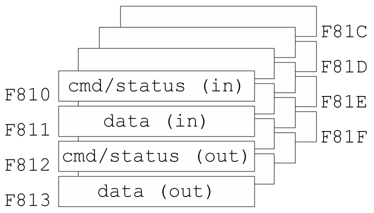


Figure 2.22: Simple memory-mapped 4-port serial interface

Scheduling

Context switching is the mechanism used by the operating system to switch from one running process to another; *scheduling* refers to the decision the operating system must make as to *which* process to switch to next. Scheduling is not covered in much detail in this version of the text.

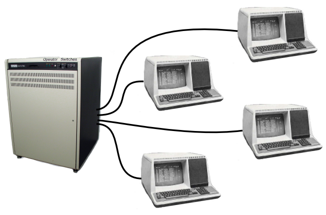


Figure 2.23: Old (c. 1975?) multi-user computer system with 4 serial terminals.

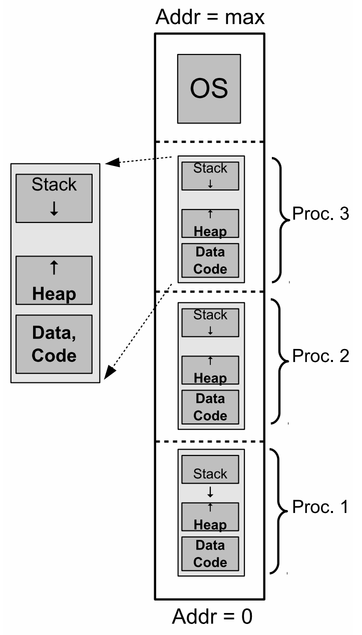


Figure 2.24: Possible memory address layout for 4 processes plus operating system.

Multi-User Computer System

We now have all the software mechanisms needed to construct a multi-user computer system.Instead of a keyboard and video display we will use *serial ports* connected to external terminals; the system is shown in Figure 2.23 and the details of the memory-mapped interface to the serial ports are shown in Figure 2.22. When the user types a character on their terminal it will be transmitted over the serial line and received by the serial port, which will set the input status to 1 and put the received character in the input register. (just like the keyboard controller)[11]

To output data to the user a character is written to the output register, which is then transmitted over the serial line and displayed to the user by

[10]Depending on the CPU there may be a few differences in stack layout between an interrupt and a function call, but these can be patched up in software.

[11]It may seem to a modern reader that such a terminal would be as complex as a computer; however the earliest terminals ("teletypewriters") were almost entirely mechanical.

the terminal. It takes some amount of time to transmit a character; during this time the output status register is set to 1, and a new character should not be written until it returns to zero. Again similar to the keyboard controller we can also perform interrupt-driven I/O; in this case one interrupt indicates when a character has been received, while a second indicates that a character has finished being transmitted and we may send the next character.

Review Questions

2.9.1. Multiple copies of the same program:

1 Can share their entire memory space, since they have the same code and variables: *yes/no*
2 Can share their program code, but not the data memory holding their variables: *yes/no*
3 Can't share their code memory, because the two processes would interfere with each other as they try to execute the same instructions: *yes/no*

I/O-driven Context Switching

Now we know *how* to switch between programs, but *when* should we do it? We see one possible answer in Figure 2.25—switching on user input. Many simple programs (e.g. the shell, editors, etc.) consist of a user input loop: the program waits for input from the user, processes it, displays any resulting output, and then waits for user input again. Most of the time the program is idle, waiting for input; we take advantage of this by modifying the OS input routine to switch to another process when there is no input ready.

The code in Figure 2.25 will not switch to another process until the current process explicitly requests more input. For input which requires very little processing (e.g. an editor updating the screen) this is fine. However, if the program were to perform large amounts of computation before its next input request, then the other users might not be able to get a response for a long period of time. We can address this problem using interrupts: (1) When data is received for a program which is waiting for input, we switch to that program, allowing it to respond immediately. (2) When the timer interrupt fires we switch from the currently running process to another running process. (A "running" process is one that is not waiting for input—i.e. one that was previously suspended by a timer interrupt.)

2.10 Address Spaces for Multiple Processes

In Figure 2.24 we see a possible address space layout for our 4-user system, with four programs—one per terminal—each receiving about a quarter of the available memory. There is one significant problem, though: How do we get programs to run in these different memory regions?

As mentioned earlier in this chapter, the location at which a program is placed in memory is important, because there are many locations in a typical program where the address of a portion of the program is needed as part of an instruction. (e.g. for a subroutine call: on many CPUs, a function call `f()` would be compiled to the instruction `CALL f`, with the address of `f` forming part of the instruction.) If a program has been compiled to start at a specific location in memory[12] then it typically will not work if loaded into a different location.

There are a number of different ways to handle this problem:

- *fixed-address compilation:* each program to be run on the system could be compiled multiple times, once for each possible starting point, and then the appropriate one loaded when a user runs a

```
terminal is {
   queue   unclaimed_keystrokes;
   process *waiting_process;
   ...
};
process *current;
queue of (process*) active;

GETKEY(terminal *term):
   if (term->unclaimed_input is empty)
      term->waiting_process = current
      switch_to(active.pop_head())
  return term->unclaimed_input.pop_head()

interrupt:
   term->unclaimed_input.push_tail(key)
   if (term->waiting_process)
      active.push_tail(term->waiting_process)
      term->waiting_process = NULL
```

Figure 2.25: Context switching on GETKEY—while a process is waiting for input we take it off of the list of active processes; when input is received we wake the process waiting for it.

[12]E.g. 32-bit Linux programs are typically compiled to start at address 0x8048000.

```
200   CALL 500
      ...
500   ...

      (a)
```

```
200  CALL PC+300
     ...
500  ...

     (b)
```

Figure 2.26: Example of absolute and PC-relative addressing, both loaded at address 200

program. This seems like a bad idea, as it is inflexible and complex in many different ways. (e.g. it fixes the locations of the partitions, regardless of the total system memory size, or the size of a program, or how many programs we might wish to run at once) The only place I've seen this approach used is in certain embedded systems, where you may have multiple separate programs running at once but they are all compiled together as part of a single firmware version.

- *position-independent code:* here we ensure that programs are compiled in a way that makes them insensitive to their starting address, by using what is called *PC-relative addressing*. This is illustrated in Figure 2.26: rather than using an absolute address (e.g. 500 in the figure) for a function call, we use an alternate instruction which indicates an offset from the current PC. Unfortunately this is frequently inefficient; for instance 32-bit Intel architecture CPUs are able to efficiently perform PC-relative CALL and JMP instructions, but require multiple instructions to perform a PC-relative data access. (this was fixed in the 64-bit extensions)
- *load-time fixup:* Here we defer the final determination of addresses until the program is actually loaded into memory. The program file, or *executable*, will thus contain not only the code and data to be loaded into memory, but a list of locations which must be modified according to the address at which the program is placed in memory. Thus in Figure 2.26, this list would indicate how the target of the CALL instruction should be calculated.[13]
- *hardware support:* By far the most popular way of sharing system memory between multiple running programs is by the use of hardware address translation; such hardware support is required to run modern general-purpose operating systems such as Linux, Mac OS X, or Windows. The basic idea is illustrated in Figure 2.27: the CPU uses *virtual addresses* for instruction fetches or data loads and stores, which are then translated by an MMU (Memory Manage-

[13]This approach is used on uClinux, a modified version of Linux which runs on low-end microcontrollers lacking virtual memory hardware.

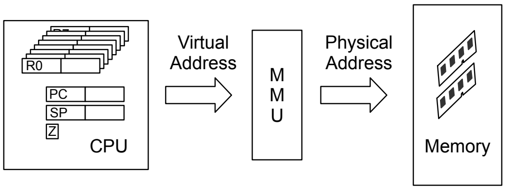


Figure 2.27: Virtual-to-physical address translation. All addresses in the CPU are virtual, and are translated to physical addresses by the MMU (Memory Management Unit) before being used to access physical memory.

ment Unit) to *physical addresses* (i.e. the actual address of a byte within a specific memory chip) for each memory operation.

2.11 Memory Protection and Translation

Hardware-supported address translation and memory protection (e.g. see Figure 2.27) is used on all well-known general-purpose operating systems today (e.g. Linux, OSX, Windows, and various server operating systems) as well as many others (e.g. the OSes used on most cell phones)[14]. Address translation is used for the following reasons:

- Flexible sharing of memory between processes. As seen above, sharing a single physical address space between a set of processes that changes over time is complicated without hardware support. Address translation allows programs to be compiled against a standard virtual address space layout, which is then mapped to available memory when the program is loaded into memory.
- Security. On a multi-user computer there are obvious reasons for preventing one user from accessing another's data; to accomplish this it is necessary to prevent "normal" processes from directly accessing memory used by another process or by the operating system. (even if the system is only used by one user at at time, the operating system must be protected if it is to be relied on to prevent access by one user to another user's files.)
- Robustness. If a program is allowed to write to any address in the system, then a bug in that program may cause the entire system to

[14]Address translation costs both money and power to add to a CPU; thus for instance the iPod Touch has a CPU with address translation, while the iPod Nano doesn't.

crash, e.g. by corrupting the operating system.[15] If a process is constrained to only modifying memory that it has been allocated, then the same bug would cause only that process to crash, after which it may be restarted.

It is possible to ensure this degree of protection with software mechanisms under certain very limited circumstances, by e.g. restricting user processes to only use Java bytecodes rather than direct program execution.[16] In the normal case however, where an application is allowed to directly execute most CPU instructions at full speed, hardware support is needed to prevent a process from making unauthorized memory reads and writes. This mechanism needs to be reconfigured by the operating system on every context switch, to apply the correct set of permissions to the running process, yet programs themselves must be prevented from modifying the configuration to bypass permission checking.

How can we allow the OS to modify memory protection, while preventing user programs from doing so and subverting memory protection? This is done by introducing a *processor state*: when the processor is running in *user* mode it is not allowed to modify memory mapping configuration, while when running in *supervisor* (also called *kernel*) mode it may do so. The code of a normal application executes in user mode, while the operating system *kernel*[17] runs in supervisor mode. We next need a mechanism for safely entering supervisor mode when either (a) an application invokes a system call, or (b) a hardware interrupt occurs, and then switching back to user mode when returning.

A question for the reader - what might happen if unprivileged programs were able to modify the exception table?

This is typically done via the interrupt or exception mechanism, which (as described earlier in this chapter) causes a forced function call in response to certain events,to an address specified in a *exception vector* or *exception table*. If we use an exception for invoking system calls, and the CPU always switches to supervisor mode when handling exceptions, then all operating system code will run in supervisor mode, and a special instruction may be used to return back to user mode when a system operation is complete. As long as the exception table is protected

[15]This happened frequently in MS-DOS, which had no memory protection.

[16]For instance, this approach is used by the Inferno operating system from Bell Labs, as well as several Java-based research operating systems.

[17]The core of the operating system, which does not run as a process—i.e. ignoring system services which run as normal processes.

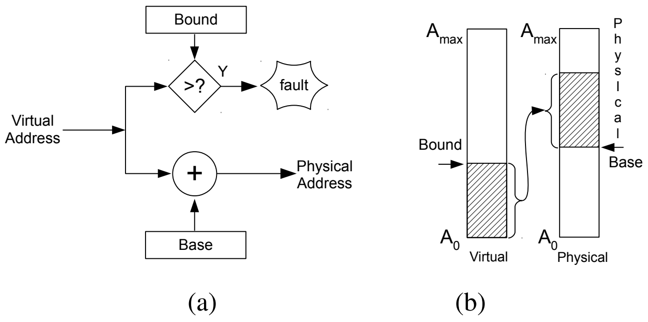


Figure 2.28: Base and bounds address translation, depicting address calculation (left) and virtual to physical memory map correspondence (right).

from user-space modification, this hardware mechanism provides the a basis on which a secure operating system may be built.

The simplest such address translation mechanism is known as *base* and *bounds* registers, as illustrated in Figure 2.28a. A virtual address is first checked to ensure that it lies between 0 and a limit specified in the *bounds* register; if this check fails, an exception is raised and the operating system can terminate the process. Otherwise an offset (from the *base* register) is added to the virtual address, giving the resulting physical address. In this way a standard virtual address space (addresses 0 through the process size) is mapped onto an arbitrary (but contiguous) range of physical memory, as shown in Figure 2.28b.

There are a few complications in getting this to work with supervisor mode, as it needs to be able to access OS data structures which are (a) inaccessible to user-space code, and (b) at the same location in memory no matter which user-space base register value is currently being used. Although several techniques have been used, the simplest one is to ignore base and bounds registers in supervisor mode, so that the operating system uses physical addresses, giving access to all of memory, while user processes execute in separate translated address spaces[18].

The switch from user to supervisor memory space (e.g. switching from translating via the base+bounds registers to using direct addressing) is

[18]This also makes it easier for the OS to change base+bounds registers when switching between processes, as it will have no effect on supervisor-mode address translation. Changing the mapping of the memory region being currently executed—something which most operating systems have to do very early in the boot process—is a very tricky thing.

done automatically by the hardware on any trap or interrupt. The operating system is then free to change the values in the (user) base and bounds registers to reflect the address space of the process it is switching to.

2.12 Putting it all together

In the introduction we saw the example of a simple command (`ls`) being executed in Linux. Many of the details of its operation were covered in this chapter.

Hardware: In our example, the keyboard controller was for an old-fashioned PS/2 keyboard, and the text display used was the simplest text mode supported by PC hardware, normally only used by some BIOSes. These are almost identical to the corresponding I/O devices in our hypothetical computer—they're located at different addresses, and support a few extra functions (e.g. flashing letters, key-up and key-down events, and keyboard *output* to e.g. turn on the caps-lock light), but otherwise are the same.

Code: To explain the operating system code we'll use the 64-bit Linux kernel version 4.6.0, because that's what I have handy. (you can browse and search the source code at `http://elixir.free-electrons.com/linux/v4.6/source`) If I use the kernel debugger to put a breakpoint on the actual TTY read function (`n_tty_read`) we get the following backtrace, which we will refer to in explaining input operation:

```
(gdb) backtrace
#0  n_tty_read (tty=0xffff88003a99fc00, file=0xffff880036b3e900,
    buf=0x7ffcff243a77 "", nr=1) at drivers/tty/n_tty.c:2123
#1  0xffffffff814d2792 in tty_read (file=0xffff880036b3e900, buf=<optimized
    out>, count=1, ppos=<optimized out>) at drivers/tty/tty_io.c:1082
#2  0xffffffff8121a197 in __vfs_read (file=0xffff88003a99fc00, buf=<optimized
    out>, count=<optimized out>, pos=0xffff88003b60bf18) at fs/read_write.c:473
#3  0xffffffff8121b236 in vfs_read (file=0xffff880036b3e900, buf=0x7ffcff243a77
    "", count=<optimized out>, pos=0xffff88003b60bf18) at fs/read_write.c:495
#4  0xffffffff8121c725 in SYSC_read (count=<optimized out>, buf=<optimized out>,
    fd=<optimized out>) at fs/read_write.c:610
#5  SyS_read (fd=<optimized out>, buf=140724589050487, count=1) at
    fs/read_write.c:603
#6  0xffffffff81798a76 in entry_SYSCALL_64 () at arch/x86/entry/entry_64.S:207
#7  0x0000000000000001 in irq_stack_union ()
#8  0x0000000000000000 in ?? ()
```

System calls: The Linux command line is a separate program, the *shell*, running in its own process, which invokes the `read` system call by executing the `INT0x80` instructure with the system call number (`SYS_READ` = 3) in the `EAX` register, the file descriptor (stdin = 0) in `EBX`, a buffer pointer in `ECX`, and the buffer length in `EDX` - see `'man 2 read'` for a full description of the system call semantics. (note that this is how it works for 32-bit mode; it's slightly different and more complicated for 64-bit.)

The `entry_SYSCALL_64` function is the trap handler; it saves all sorts of registers, checks that it's a legal system call number, and then calls the appropriate entry in the system call table. (since it needs to save registers and perform other machine-level functions it is one of the few kernel functions written in machine language)

```
#6  0xffffffff81798a76 in entry_SYSCALL_64 () at arch/x86/entry/entry_64.S:207
207             call    *sys_call_table(, %rax, 8)
```

I/O virtualization: Linux file descriptors are small integers which index into a per-process array of pointers to internal kernel file structures. File descriptor 0 is standard input, and 1 is standard output. The pointer to the current process structure is called (unsurprisingly) `current`; we can look into its file table and see that entries 0 and 1 point to the same file structure (ending in 3e900) passed to `n_tty_read` in the stack trace above:

Note that the operating system kernel is almost entirely composed of exception handlers, which run in response to deliberate traps from user applications (system calls) or accidental ones (e.g. memory access faults), as well as interrupts from I/O devices and timers. This means that when a system is idle it is not actually executing code in the operating system kernel itself; instead a special *idle process* with lowest priority executes when no other work is available.

```
          (gdb) p current->files.fdtab.fd[0]@2
          $9 = {0xffff880036b3e900, 0xffff880036b3e900}
```

The `SYSC_read` function looks up this structure (returning an error for bad file descriptor numbers); `vfs_read` does a few more checks, and then calls `__vfs_read` which forwards to the "read" method from the file operations table in the file structure:

```
#2  0xffffffff8121a197 in __vfs_read (file=0xffff88003a99fc00, buf=<optimized
    out>, count=<optimized out>, pos=0xffff88003b60bf18) at fs/read_write.c:473
473                     return file->f_op->read(file, buf, count, pos);
```

When the file was originally opened, this operations table was set to point to the read and write operations for the TTY driver, which is responsible for keyboard input and text-mode screen output:

```
(gdb) p file->f_op
$13 = (const struct file_operations *) 0xffffffff81872fa0 <tty_fops>
(gdb) p *file->f_op
$14 = {owner = 0x0, llseek = 0xffffffff81219ff0 <no_llseek>,
  read = 0xffffffff814d2700 <tty_read>, write = 0xffffffff814d27f0 <tty_write>,
  ...
```

Context switching: In `n_tty_read` it adds the current process to a wait queue, then checks to see if there is any input (or error conditions or lots of other reasons why it might return early) and if none, it goes to sleep:

```
2166            add_wait_queue(&tty->read_wait, &wait);
  ...
2188                    if (!input_available_p(tty, 0)) {
  ...
2207                            timeout = wait_woken(&wait, TASK_INTERRUPTIBLE,
2208                                                 timeout);
```

Here `wait_woken` sets a few things and then calls `schedule_timeout`, which sets a timer and then calls `schedule`, the central context switch function, which picks the next runnable process and switches to it.

The interrupt which wakes it up is much more convoluted, as the actual interrupt handler schedules a “deferred work” callback which does the real work. (why? For several reasons, one of which is that you can block in a deferred work handler while interrupts have to return immediately.) Here are selected lines from the interrupt backtrace:

```
#0  tty_schedule_flip (port=<optimized out>) at drivers/tty/tty_buffer.c:406
#1  tty_flip_buffer_push (port=0xffff88003e088000)
    at drivers/tty/tty_buffer.c:558
#2  0xffffffff814dc8ae in tty_schedule_flip () at drivers/tty/tty_buffer.c:559
#3  0xffffffff814e490e in put_queue (ch=<optimized out>, vc=<optimized out>)
    at drivers/tty/vt/keyboard.c:306
 ...
#8  0xffffffff814e5c11 in kbd_keycode (hw_raw=<optimized out>, down=<optimized
    out>, keycode=<optimized out>) at drivers/tty/vt/keyboard.c:1457
#9  kbd_event (handle=<optimized out>, event_type=<optimized out>,
    event_code=<optimized out>, value=2) at drivers/tty/vt/keyboard.c:1475
 ...
#16 atkbd_interrupt (serio=0xffff88003684e800, data=<optimized out>,
    flags=<optimized out>) at drivers/input/keyboard/atkbd.c:512
#17 0xffffffff8162fdc6 in serio_interrupt (serio=0xffff88003684e800,
    data=57 '9', dfl=0) at drivers/input/serio/serio.c:1006
#18 0xffffffff81630e72 in i8042_interrupt (irq=<optimized out>,
    dev_id=<optimized out>) at drivers/input/serio/i8042.c:548
 ...
#23 handle_irq (desc=<optimized out>, regs=<optimized out>)
    at arch/x86/kernel/irq_64.c:78
#24 0xffffffff8179b22b in do_IRQ (regs=0xffffffff81c03df8
    <init_thread_union+15864>) at arch/x86/kernel/irq.c:240
```

which schedules the deferred work:

```
#1  tty_schedule_flip (port=<optimized out>) at drivers/tty/tty_buffer.c:406
400             struct tty_bufhead *buf = &port->buf;
 ...
406             queue_work(system_unbound_wq, &buf->work);
(gdb) p *buf->work
$41 = {data = {counter = 64}, entry = {next = 0xffff88003e088010,
  prev = 0xffff88003e088010}, func = 0xffffffff814dcd00 <flush_to_ldisc>}
```

If we put a breakpoint on `flush_to_ldisc` and step through it, you eventually get to the following lines:

```
        if (read_cnt(ldata)) {
  ...
                wake_up_interruptible_poll(&tty->read_wait, POLLIN);
```

which wake up the shell process that was sleeping on `tty->read_wait`, by removing it from the queue associated with `read_wait` and reinserting it into the list of runnable processes.

Process creation: The shell process executes the `ls` command by invoking `fork`, to create a subprocess, and then invoking `wait` to wait until the subprocess has finished. Within the subprocess the `exec` system call is used to load and execute the `ls` program itself; when it is done the `exit` system call frees the subprocess and causes the `wait` in the parent process to return. (process creation will be covered in more depth when we look at virtual memory)

Output: The shell and the `ls` processes send output to the screen by using the `write` system call; the text console driver is responsible for determining where the next character should be placed on the screen, handling end-of-line, and copying data to scroll displayed text upwards when it reaches the end of the buffer. (this way both processes can output to the same screen without over-writing each other)

In particular, `tty_write` eventually calls `do_con_write` in `drivers/tty/vt/vt.c`, which has a bunch of convoluted logic to handle line wrap, scrolling, cursor control commands, etc., but for the simplest case just adds on 8 bits to set the right background and foreground color, and writes into the screen buffer via a pointer:

```
#define scr_writew(val, addr) (*(addr) = (val))
...
            scr_writew((vc_attr << 8) + tc,
                       (u16 *) vc->vc_pos);
```

Answers to Review Questions

2.1.1 *yes/no/sort of*: "no". I/O devices are pieces of hardware separate from the memory and the CPU, e.g., a card that plugs into the PCI bus. Software, whether part of the operating system or a program, consists of instructions in memory that are executed by the CPU.

2.1.2 *yes/no/sort of*: "sort of". The CPU interacts with most I/O devices as if they were normal memory locations, using load and store instructions to memory addresses. However, unlike normal RAM, which just stores the value written and returns it when read, the device takes various actions when the CPU reads or writes its memory locations.

2.4.1 "yes". Although programs may occasionally interact directly with specific pieces of hardware, a primary purpose of the operating system is to provide simple and consistent interfaces to complex and varying hardware devices.

2.4.2 "maybe". Some systems don't have a display. On a system with a display, the operating system may manage that display for user programs, as it does the keyboard (e.g., in Windows). On other systems (e.g., Linux), a separate program may be responsible for the interface.

2.4.3 "maybe". The simplest operating systems support a single, preloaded program, while the whole point of general-purpose operating systems like Windows or Linux is to allow the user to load their own programs.

2.4.4 "no". That's the whole point of a system call table. The addresses of functions in a program or the operating system may change if the code is modified and recompiled, but the system call table remains constant.

2.7.1 No. Hardware interrupts are external asynchronous events, and can occur at any point during program execution. (well, almost any point. It's possible to disable interrupts while executing code which can't be interrupted.)

2.7.2 No. An interrupt tells the CPU that something happened (or one of several possible somethings, if an interrupt line is shared), but that's all. It's the job of the interrupt handler to figure out what happened and handle it (hence the name) by e.g. reading in newly available data.

2.7.3 Sort of. Interrupts can easily be used to perform brief tasks — examples include buffering a keystroke in response to the keyboard interrupt, or flashing a cursor in the timer interrupt. Implementing the equivalent of a full program in interrupt handlers would be horribly complicated, however.

2.8.1 The stack holds: *Function arguments, return addresses* : yes, they are pushed onto the stack before calling a function. *Global variables* : no, there is only one copy of each global variable, so they are allocated fixed locations in memory. *Local variables* : yes, this way there is a separate copy of each local variable each time a function is called, even if it is called recursively, and the memory is automatically freed when the function returns.

2.8.2 the return instruction doesn't know anything about the corresponding `CALL` — it just uses the address on the top of the stack. It is the responsibility of the `CALL` instruction to put the return address there, and of the code in the function to make sure that address is not corrupted.

2.8.3 `RET`. Process A uses `CALL` to invoke the switch function, but it is the `RET` at the end of switch, after B's saved stack pointer is restored, that actually results in resuming execution of B's code.

2.9.1 1 *(share entire memory space)* No, in this case each process would see its variables change unexpectedly as the other processes updated them.

2 *(share code, not data)* Yes, it might be simpler to give each process a separate copy of its program code, but it's not necessary. Writable data (and stack) must be separate, however.

3 *(cannot share code)* No, the CPU is only executing one instruction at a time, and really doesn't care what another process might do sometime in the future after a context switch.

Chapter 3

Synchronization – Safety & Sequencing

3.1 Problem Introduction

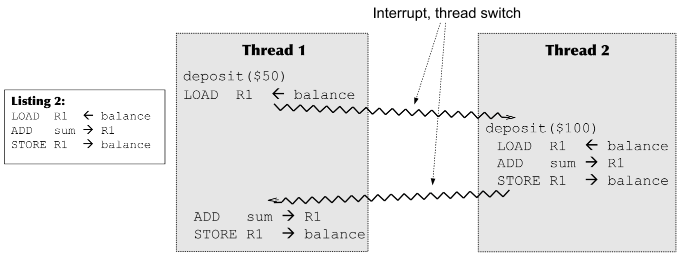
One of the key responsibilities of an operating system is that of synchronization—handling nearly simultaneous events in a reasonable way, and providing mechanisms for user applications to do so as well.

In Figure 3.1 we see a simplified example of a program to maintain a bank account balance at the Bank of Lost Funds. When running on a single CPU, the `deposit` function is trivially correct: after it completes execution, the value of `balance` will be `sum` greater than it was before the function was invoked.

In Figure 3.1, however, we see one possible result when this function is invoked by two threads nearly simultaneously. In this case thread 1 is interrupted after it has read the current value of `balance`, but before it could store the new value back to memory. The result is that the update performed by thread 2 is lost, being over-written by thread 1's computation, and after depositing a total of $150 to the account we have a final balance of $50.

```
money_t balance;
function deposit(money_t sum) {
   balance = balance + sum;
}
```

Listing 3.1: Simple bank account example



```
LOAD  R1  ← balance
ADD   sum → R1
STORE R1  → balance
```

```
deposit($50)
LOAD  R1  ← balance

  ADD   sum → R1
  STORE R1  → balance
```

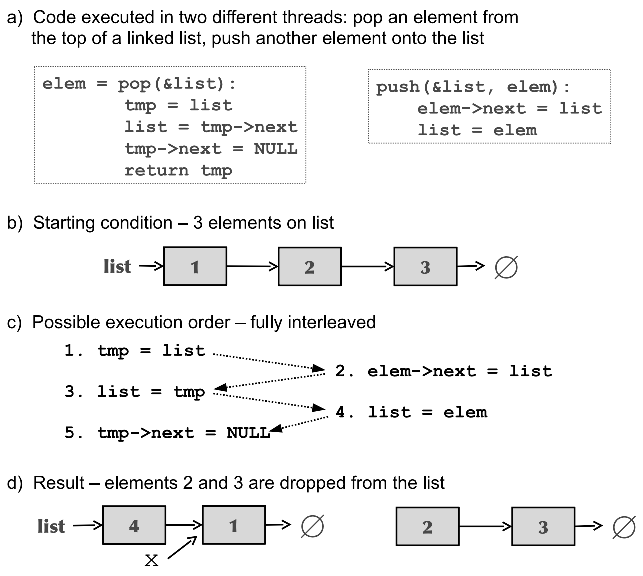
```
deposit($100)
  LOAD  R1  ← balance
  ADD   sum → R1
  STORE R1  → balance
```

Figure 3.1: Incorrect operation of banking example. An interrupt causes a thread switch *after* thread 1 has loaded `balance` into R1 and *before* it writes the updated value back into `balance`, so thread 2's update is lost.

3.2 Race Conditions and Mutual Exclusion

Such errors are referred to as *race conditions*, because the result depends on a "race" between threads, where it is unknown which will execute some piece of code first.

Another example of such a race condition is shown in Figure 3.2(a) and


a) Code executed in two different threads: pop an element from the top of a linked list, push another element onto the list
b) Starting condition – 3 elements on list
list
1
2
3
∅
c) Possible execution order – fully interleaved
d) Result – elements 2 and 3 are dropped from the list
list
4
1
∅
X
2
3
∅

```
elem = pop(&list):
        tmp = list
        list = tmp->next
        tmp->next = NULL
        return tmp
```

```
push(&list, elem):
    elem->next = list
    list = elem
```

```
1. tmp = list
                        2. elem->next = list
3. list = tmp
                        4. list = elem
5. tmp->next = NULL
```

Figure 3.2: Linked list corruption. (a) code for push and pop, (b) starting data structure, (c) interleaving of pop and push, (d) final state. Items 2 and 3 are no longer on the list, and item 1 is both on the list and the return value from pop

```
mutex_t n = mutex_create()
mutex_lock(n)
mutex_unlock(n)
mutex_destroy(n)
```

Listing 3.2: Hypothetical operating system interface to create, use, and destroy mutexes.

(b), which shows a simple linked list, along with the code to use it as a push-down stack by pushing and popping elements. In Figure 3.2(c) and (d) we see what happens when a push and a pop conflict with each other, causing the list to become disconnected; in this case the right-hand side of the list is effectively "lost", with potentially disastrous consequences.

The most insidious aspect of each of these race conditions is that they occur in otherwise bug-free code; in particular, there is no amount of testing which is guaranteed to find them.

The solution to race conditions is fairly obvious, although not always simple: we identify all the cases where data must be protected against simultaneous modification or access, and prevent this from occuring[1]. To do this we create an object called a `mutex` (see Figure 3.2) which has the ability to guard against simultaneous access. This object has two methods, `lock` and `unlock`, and the following properties:

> In classic operating systems textbooks this is referred to as the *critical section problem*, defined as the case where there is a *critical section* of code which must be guarded against simultaneous execution. This is unfortunately a misleading term, as it should be obvious that it is the *data* that must be protected, not the code. For instance, in an object-oriented program a class may have two (or more) methods which can interfere with each other, even though different sections of code are being executed; conversely no interference will occur if any of these methods are invoked simultaneously on separate object instances.

- Given a mutex `m`, once some thread T1 returns from `m.lock()`, no other thread T2 will return from `m.lock()` until T1 enters `m.unlock()`.
- If thread T1 is holding mutex `m` (i.e. it has entered and returned from `m.lock` and T2 is waiting for `m` (it has entered but

[1]The simplest way to do this is to only allow single-threaded programs. This was the case for almost all operating systems until the mid-90s; multi-threading and locking were obscure concerns which only kernel programmers had to worry about

```
object account is:
    mutex m
    int   balance

    method deposit(int amount):
        m.lock()
        balance = balance + amount
        m.unlock()

    method get_balance():
        return balance
```

Listing 3.3: Safe bank account object. Note that other actions which modify the balance, such as `withdraw()`, must lock mutex `m` as well.

not returned from `m.lock()`), then when T1 enters `m.ulock()`, T2 (or some other thread blocked on `m`) will "promptly" return from `m.lock()`.

(these properties are also termed *mutual exclusion*—hence the name mutex—and *progress*, and are two of the three formal requirements for a solution to the critical section problem.)

When thread T1 returns from `m.lock()`, we often say that T1 has *acquired* the mutex `m`, or that it is *holding* it; when T1 invokes `m.unlock()` it *releases* the mutex. Note that other threads are free to *call* the lock method on `m` while `m` is held by T1; however none of those threads will *return* from the call until the mutex is released. If T1 were to hold the mutex for a long time, this would delay the other threads; if it fails to ever release the mutex (e.g. due to raising an exception before the call to `unlock()`) it would be a serious bug, typically causing the program to freeze.

We can now write a thread-safe version of our bank account object, as seen in Figure 3.3. It avoids the race condition described in the beginning of the chapter by using a per-instance mutex to guard operations which modify the balance. By doing this we have made the modification of the balance *atomic*[2], at least with respect to any other code which properly locks the mutex—i.e. it appears to happen as a single operation, with any other modification happening either before or after, but not simultaneously.

In Figure 3.3 we can (on most computers) safely read the balance without locking the mutex, because the hardware can usually be trusted to perform a read of a single integer atomically. Another way to state this is that the

[2]The name *atom* derives from the ancient Greek word for *indivisible*, and so is something that can't be cut or divided. (or at least couldn't be until the physicists got to work on it) An *atomic operation* cannot be divided into parts by another operation.

```
object account is:
   mutex  m
   int    balance_dollars
   int    balance_cents

   method deposit(int dollars, int cents):
      m.lock()
      balance_cents = balance_cents + cents
      if balance_cents >= 100:
         balance_dollars = balance_dollars + 1
         balance_cents = balance_cents - 100
      balance_dollars = balance_dollars + dollars
      m.unlock()

   method get_balance(out &d, out &c): // d,c are outputs
      m.lock()
      d = balance_dollars
      c = balance_cents
      m.unlock()
```

Listing 3.4: Bank account object with more complex state. To avoid observing invalid state (e.g. a cents value greater than 99) we must lock the mutex when reading as well as writing.

object is in a *safe* state at all times—it changes atomically from one safe state to another. In Figure 3.4 we see a bank account object with a slightly more complex state, representing integer dollars and cents separately; in this case reading the object state in the middle of an update could give incorrect results, e.g. showing $balance\_cents > 99$. (more serious problems such as null pointer errors can occur when accessing complex data structures such as linked lists or trees during an update) To prevent this, the code in Figure 3.4 locks the object *when observing its state*, so that it only sees the consistent state found after an update has fully completed.

Review Questions

3.2.1. Race conditions can be detected by exhaustively testing all the possible orders in which inputs may be sent to your program:
True / False

3.2.2. You have just been asked to write the `withdraw()` method for our bank account object. Which of these locking options will ensure that it works correctly?

a) Add a second mutex (i.e., m2) to the object, and lock/unlock this second mutex when making a withdrawal.
b) Lock mutex m at the beginning of the withdrawal method, and unlock it at the end.
c) There's no need to use a lock here, because the value of the balance is being decreased instead of increased.

3.3 Implementing Mutexes

So mutexes are great, but how do they actually work? In Figure 3.2 we saw a hypothetical system call interface which allows us to create, destroy, lock and unlock mutexes. Internal to the OS we can assume that each mutex has a state—locked or unlocked—and a list of threads waiting for the mutex. If a process calls `mutex_lock` on an unlocked mutex, the mutex is marked as locked and `mutex_lock` returns immediately. If the mutex is locked, then the call is treated almost exactly like waiting for I/O: the OS puts the thread on the mutex wait queue, and then switches to the next active thread. When `mutex_unlock` is called, the OS takes the first thread (if any) off the queue and puts it back on the active list.

So now that we know exactly how our mutex system calls are supposed to behave, how do we implement them? In addition, how does the operating system protect its own data structures, which (in e.g. Linux and Windows) reside in a single address space and are accessed from not only multiple user processes (via system calls) and kernel threads, but also from exception handlers for e.g. page faults and hardware interrupts?

On a single-processor system this is fairly straightforward. Code runs in a straight line unless it is interrupted by a hardware interrupt or an exception such as a page fault, so all we need to do is to (a) disable interrupts, and (b) ensure that the operating system code and data (or at least the code and data needed for mutexes) is always mapped into physical memory, to avoid page faults.

```
structure mutex:
    bool locked = False // guarded by IRQ disable
    queue waitlist    // waiting threads (also guarded)

mutex_lock(mutex m):
    disable_interrupts()
    if not m.locked
        m.locked = True
        enable_interrupts()
    else:
        pause(current_process) // remove it from active list
        m.waitlist.add(current_process)
        enable_interrupts()
        sleep()             // wake here when mutex acquired

mutex_unlock(mutex m):
    disable_interrupts()
    if waitlist is empty:
        m.locked = False
        enable_interrupts()
    else
        local next_thread = m.waitlist.pop_from_head()
        enable_interrupts()
        wake(next_thread) // add it to the active list
```

Listing 3.5: Simple single-CPU kernel mutex. The "locked" flag and list of waiting processes are guarded by disabling interrupts

(Note that user-level code is not allowed to disable interrupts, as doing so for more than a brief period is likely to crash the machine.)

In Figure 3.5 we see a mutex implementation based on this. We assume the same context-switching structure used in Figure 2.25 in the previous chapter, with a thread control structure containing fields such as the saved stack pointer as well as links for creating lists:

- `current` points to the currently running thread
- `active` is a list of other threads ready to run
- `sleep` pops the next runnable thread from `active`, assigns it to `current`, and switches to it[3].
- `wake` appends a thread to the active list so that it can run again.

On a single-CPU system the fields of the mutex structure are protected from race conditions, as no interrupts will occur during modifications. We can see that our mutex requirements will be met, by noting that:

[3]As opposed to `yield`, which adds the current thread to the end of the active queue before performing the same steps.

```
typedef int spinlock_t
spin_lock(spinlock_t *lock_addr):
    register r = 1
    while r == 1:
        SWAP r, lock_addr

spin_unlock(spinlock_t *lock_addr):
    *lock_addr = 0
```

Listing 3.6: Spinlock implementation. If the lock contains 0, it is unlocked; if 1, then it is locked, in which case a second thread (or CPU) trying to acquire it will "spin" (i.e. loop) until it is released.

- the first thread to call `lock(m)` will set `m.locked` to true and return immediately.
- if another thread calls `lock(m)` before the mutex is unlocked, it will queue itself on `m.waitq` and sleep.
- when `unlock(m)` is called, if there are any threads waiting then the first one will be woken up (and thus continue from its `sleep` call and return from `lock(m)` the next time it is scheduled), and the mutex will remain locked;
- if no threads are waiting the mutex will be unlocked.

On a multi-core system the problem is more complicated, however, as the CPU cores are all executing simultaneously, accessing the same memory, whether interrupts are enabled or not. Implementing a mutex on a multi-core system requires coordinating via the memory system shared between all the CPUs, using special instructions which are guaranteed to execute uninterrupted by instructions running on any of the other CPU cores.

An exercise for the reader - many textbooks describe Dekker's and Peterson's algorithms for mutual exclusion, which use normal memory load and store instructions to provide mutual exclusion. Try implementing Peterson's algorithm as described in Wikipedia, with two threads each looping N times, each time (a) entering the critical section, (b) incrementing a counter, and (c) leaving the critical section. For large N (e.g. 10^7) does the counter always get incremented 2N times? Why not? (feel free to ask in class if you don't find the answer)

There are a number of specialized CPU instructions which are typically provided to implement mutual exclusion; we will consider one of them, the atomic SWAP instruction[4]:

[4]Another such instruction is Compare And Swap (e.g. the Intel CMPXCHG instruction), which only performs the swap if the value in memory matches an expected value.

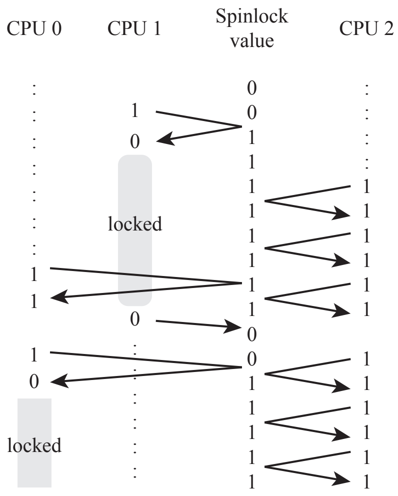


Figure 3.3: Spinlock operation. Here we see CPU 1 acquire the lock, after which CPU 2 and then CPU 0 attempt to acquire it. After CPU 1 releases the lock (by writing 0) one of the waiting CPUs (in this case 0) is then able to acquire it.

- SWAP *register*, *address*

This instruction swaps the contents of a register with the data in a specified memory location, and unlike normal instructions it is guaranteed to do so atomically. In other words, no matter how many CPU cores are trying to swap with the same memory location simultaneously, one of them will do so first, another second, and so on, and every CPU will see the location change values in the same order.

This is in contrast to normal load/store instructions, where different CPU cores may see differences in the order in which changes occur. This is not surprising when you consider that each CPU is handling multiple instructions at once, possibly out of order, and writing into cache lines which are only later flushed to main memory. For instance, if CPU 1 writes to cache line A and then to cache line B, they could conceivably be flushed to memory in the opposite order, so while CPU 1 sees A written before B, other CPUs see B written before A. Although it's possible to achieve consistent ordering—that's what atomic instructions do—it's much slower.

The SWAP instruction allows us to implement what is called a *spinlock*, as shown in Figure 3.6. An example of its operation is shown in Figure 3.3: in effect the 0 value is treated as a token that is passed between waiting CPUs

```
structure mutex:
    int  spinlock
    bool free = True // guarded by spinlock
    queue waitlist  // waiting threads, guarded by spinlock

mutex_lock(mutex m):
    disable_interrupts()
    spin_lock(&m.spinlock)
    if m.free
        m.free = False
        spin_unlock(&m.spinlock)
        enable_interrupts()
    else:
        pause(current_process) // remove it from active list
        m.waitlist.add(current_process)
        spin_unlock(&m.spinlock)
        enable_interrupts()
        sleep()              // wake here when mutex acquired

mutex_unlock(mutex m):
    disable_interrupts()
    spin_lock(&m.spinlock)
    if waitlist is empty:
        m.free = True
        spin_unlock(&m.spinlock)
        enable_interrupts()
    else
        local next_thread = m.waitlist.pop_from_head()
        spin_unlock(&m.spinlock)
        enable_interrupts()
        wake(next_thread) // add it to the active list
```

Listing 3.7: Multi-core-safe implementation of the mutex from Figure 3.5, with spinlock for additional protection

(or threads) and the lock memory location. This lock is extremely simple, and by making use of the hardware-provided atomic SWAP instruction, it guarantees mutual exclusion. However as we see in the figure it can be (a) unfair, as it does not respect the order in which CPUs begin to wait for the lock, and (b) inefficient, as CPUs 2 and 0 are unable to perform any work while waiting. We therefore use spinlocks to guard very short pieces of code, and then use these pieces of code to construct efficient and well-behaved primitives for applications to use.

A spinlock-enhanced version of the mutex in Figure 3.5 is shown in Figure 3.7; it is identical except for the addition of a spinlock, which is used in addition to disabling interrupts to guard the `locked` flag and wait queue.

This implementation retains almost all the efficiency of the single-CPU

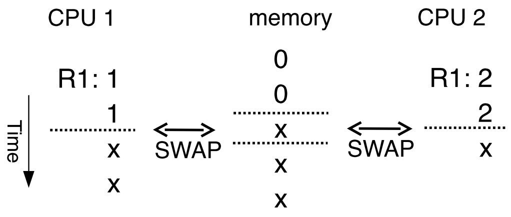


Figure 3.4: Scenario for question 3.3.1

version, as the spinlock is never held for more than a few instructions, limiting the length of time that other CPUs are stuck busy-waiting[5]. Unlike the basic spinlock, this mutex is also fair, as waiting threads will be queued and acquire the mutex in FIFO order. (at most, any unfairness in the underlying spinlock mechanism will effect the order in which threads go onto the list, not how many turns they get holding the mutex.)

More formally, what we mean by "fair" in this case is *bounded waiting*—i.e. no thread can be "starved" while other threads repeatedly acquire and release the mutex. (this is the third requirement for solutions to the critical section problem)

A question for the reader - why is it important to unlock the spinlock and enable interrupts before calling `sleep()` in `mutex_lock`?

In particular, if thread A is waiting for the mutex, bounded waiting means that another thread B cannot acquire and then release it many times while A is still waiting. (note that spinlocks cannot guarantee this property, as any waiting thread can acquire the lock, regardless of how long it has been waiting.) If multiple threads (on separate CPUs) call `mutex_lock` at once, the spinlock will determine what order they will be added on the queue, but the FIFO ordering of the queue ensures that if a thread acquires the mutex and releases it, when it tries to lock the mutex again it will go to the tail of the line.

Review Questions

3.3.1. In the example in Figure 3.4, two CPUs execute SWAP instructions with the same location in memory. CPUs 1 and 2 start with the values 1 and 2 in their registers, and the initial memory location is

[5]Sort of. On massively multi-core machines—e.g. 72 cores is a common number nowadays—highly contented locks are still inefficient, as waiting for 71 other CPUs to do a few instructions each can take a while.

zero. Which of these is a valid result after both SWAP instructions have completed?

a) CPU 1: R1=2, CPU 2: R1=1, memory: 2
b) CPU 1: R1=2, CPU 2: R1=1, memory: 0
c) CPU 1: R1=2, CPU 2: R1=0, memory: 1

3.3.2. A mutex is: a) A type of spinlock b) An application-defined class c) An OS-defined lock object

3.4 The Bounded Buffer Problem and Semaphores

Mutexes can be used to *prevent* certain orders of execution—e.g. multiple threads executing certain operations at the same time—but what if we want to *cause* a certain order of execution? (for instance, waking a thread which is waiting for keystroke input.) We refer to this as *synchronization*, and to the primitives which are used for this purpose as *synchronization primitives*.

To begin we'll examine a "classic" or pedagogical[6] synchronization problem frequently used as an example of multi-threaded programming: the *Bounded Buffer Problem*, which may be defined as follows:

1. An object `buffer` has methods `put` and `get`.
2. Successive calls to `buffer.put(item)` insert items into the buffer.
3. Successive calls to `item = buffer.get()` remove items from the buffer in the same order as they were inserted.
4. If the buffer contains no items, `buffer.get()` will block until an item is inserted.
5. If the buffer contains N items, `buffer.put()` will block until an item is removed.

We can start with a single-threaded version of the bounded buffer. In this case parts 3 and 4 of the definition must be modified, as no other thread will arrive to insert or remove an item; instead we will return NULL if no item is available, and ERROR if the buffer is full, as seen in Figure 3.5.

By adding a mutex we can safely handle multiple threads, as seen in Figure 3.6.[7]

However we still don't have a full solution to the bounded buffer problem—we need to not only protect the threads from each other,

[6]which means "for teaching purposes only", i.e. not necessarily practical.

[7]Note how locks complicate control flow—you have to make sure that all locks are released, even in failure cases.

```
list buffer

put(item):
    if len(buffer) >= N
        return ERROR
    else
        buffer.add_tail(item)
```

```
        return OK

get(item):
    if len(buffer) == 0
        return NULL
    else
        return buffer.remove_head()
```

Figure 3.5: Simple bounded buffer

but to *coordinate* or *synchronize* them, so that e.g. one thread sleeps in `get()` until another thread invokes `put()`. We haven't seen how to use a mutex for this purpose, and in fact many real-world mutex implementations cannot be used to do this[8].

> The two operations on a semaphore were originally given Dutch abbreviations *P* and *V* by their inventor, Edsger Dijkstra. Since then they have also been called *down* and *up*, *acquire* and *release*, *wait* and *signal*, *await* and *notify*, etc. We will call them *wait* and *signal*.

Instead we introduce a new object called the *counting semaphore*, which is deliberately designed for synchronizing the actions of multiple threads. Like a mutex, a semaphore is an OS-provided object; however an initial count N is specified when it is created. It has two methods, `wait()` and `signal()`, with the following behavior:

- For semaphore S with initial count N, if N_w is the total number of

```
mutex m
list  buffer

put(item):
    m.lock()
    if len(buffer) >= N
        result = ERROR
    else
        buffer.add_tail(item)
        result = OK
    m.unlock()
```

```
    return result

get(item):
    m.lock()
    if len(buffer) == 0
        result = NULL
    else
        result = buffer.remove_head()
    m.unlock()
    return result
```

Figure 3.6: Thread-safe bounded buffer

[8]In particular, for debugging purposes many implementations (such as the POSIX threads implementation in Linux) require that a mutex be unlocked by the same thread that locked it.

```
mutex     m
list      buffer
semaphore space = semaphore(N)
semaphore items = semaphore(0)

put(item):
    space.wait()
    m.lock()
    buffer.add_tail(item)
    m.unlock()
    items.signal()

get(item):
    items.wait()
    m.lock()
    result = buffer.remove_head()
    m.unlock()
    space.signal()
    return result
```

Figure 3.7: Semaphore-based bounded buffer

times any thread has returned from `S.wait()`, and N_s is the number of times any thread has entered `S.signal()`, then $N_w - N_s \leq N$.

Intuitively a semaphore may be understood by assuming that it maintains a count initialized to N. When *wait* is called it (a) waits until the count is greater than zero, then (b) decrements the count and returns. Calling *signal* increments the count, possibly waking up one of the threads waiting for $count > 0$. In practice this is done by maintaining a list of waiting threads; if there are threads waiting on this list then *signal* wakes the first one rather than incrementing the count.

A *binary semaphore* is a semaphore which can only take on the values 0 and 1, and is the same thing as a mutex. (well, disregarding implementation details of many mutexes, such as ownership checks.) Note that this behaves slightly differently from a counting semaphore initialized to 1, specifically in the case where `signal()` is called multiple times without intervening calls to `wait`[9].

A question for the reader - if you are given a function `NewSemaphore0()` which creates a new counting semaphore with its count initialized to 0, how would you write a function `NewSemaphore(N)` which returns a semaphore initialized to an arbitrary positive count N?.

Note that the behavior of the wait and signal methods of a counting semaphore are almost exactly the same behaviors as those we want for the put and get methods in our bounded buffer, keeping track of a count and blocking when that count reaches a limit. Using one semaphore to track the number of items in the buffer, and another to track the number of free spaces, we have the implementation in Figure 3.7.

[9]Not that it really matters, as a well-behaved program probably wouldn't do this.

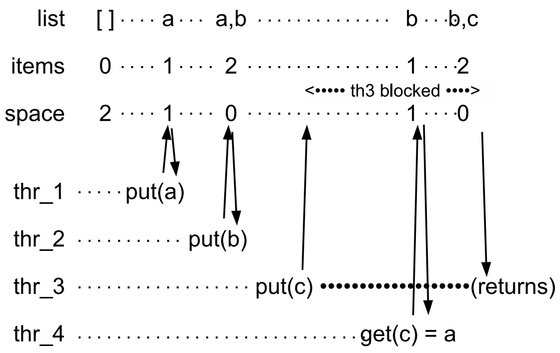


Figure 3.8: Operation of bounded buffer from Figure 3.7, limit=2

Note that we still need a mutex to protect the linked list, as although the semaphore limits the number of threads which can be modifying the list simultaneously, that limit is greater than 1. (alternately we could implement a "thread-safe linked list" class which included a mutex, thus simplifying any threaded code which used it.)

In Figure 3.8 we see this in operation. With a limit of 2 items, the first two calls to `put` return immediately; however the third one blocks as the "space" semaphore has dropped to zero. When a call to `get` from thread 4 increments the "space" semaphore again, thread 3 is able to return from `space.wait()`, decrementing its value to zero again, and can then insert its item into the list.

Review Questions

3.4.1. The bounded buffer solution with mutexes shown in Figure 3.6 is not a full solution to the bounded buffer problem because:

a) It doesn't block in `put()` or `get()` when the buffer is full or empty.
b) It sometimes loses items.
c) It doesn't maintain the items in order.

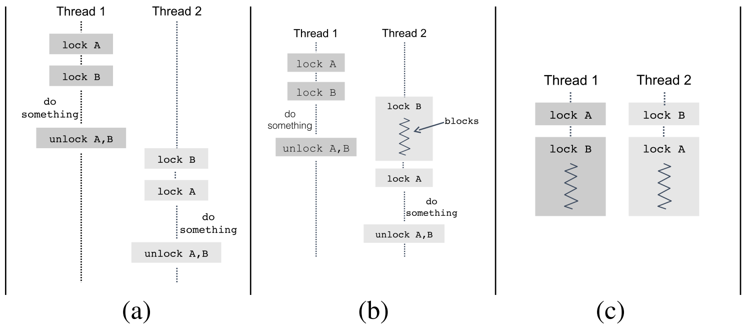


Figure 3.9: Three possible interleavings of `foo()` and `bar()`.

3.5 Deadlock

Consider the ways that the following code can execute, with thread 1 executing `foo()`, and thread 2 executes `bar()`:

```
mutex A, B;

foo:                    bar:
    lock A                  lock B
    lock B                  lock A
     ...                     ...
    unlock B                unlock A
    unlock A                unlock B
```

- If thread 1 starts early enough, we may see the result in Figure 3.9(a), where thread 1 or alternately thread 2) finishes completely before thread 2 starts.
- Or, if they start close enough in time, they may overlap somewhat but still complete successfully, as in Figure 3.9(b).
- But if they start at about the same time, there is a chance of getting the situation in Figure 3.9(c), where both threads are blocking on their second lock operation.

This is a deadlock, where two threads are each waiting for a lock held by the other thread. As you can see, it can halt program execution just as completely as a program crash or infinite loop, and typically requires the application to be killed and restarted.

Classic Conditions for Deadlock

Intuitively a deadlock is when multiple processes (or threads) are waiting for locks held by other processes in the group, each unable to give up the locks it is holding before it acquires the lock that it is waiting for. More generally, deadlocks can occur when acquiring not just locks, but other sorts of *resources*: e.g. each process might be trying to allocate N buffers out of a fixed-sized pool.

Phrased more formally, there are four classic conditions for deadlock among multiple processes contending for resources:

1. **Mutual exclusion**: A deadlock requires resources (like mutexes) that can only be held by one process
2. **Hold and wait**: A process holds one or more acquired resources and then blocks waiting to acquire another resource
3. **No preemption**: Resources are only released when a process is done with them and calls the release function (like unlock). One process cannot force another to release a resource.
4. **Circular wait**: Given the three prior conditions, if there is a circular wait then there is a deadlock

The processes that deadlock can be any form of concurrent activity: threads, processes, or interrupts vs. a foreground process. There can be any number of processes, and in some cases a process can even deadlock with itself. Finally, the resources being acquired can be anything which has both the mutual exclusion and hold and wait properties. These resources aren't just mutexes and semaphores, but things like memory buffers or the process of obtaining exclusive access to a file.

Finally, there is a deadlock case not quite covered by these conditions—the one where the programmer forgot to release a lock. Try not to do that.

Avoiding Deadlock: Lock Ranking

If any one of these four conditions can be avoided, deadlock cannot occur. If locks are always acquired in the same order, no matter what thread is acquiring them via which code path, then there will be no circular wait and thus no deadlock, as you can see in Figure 3.10.

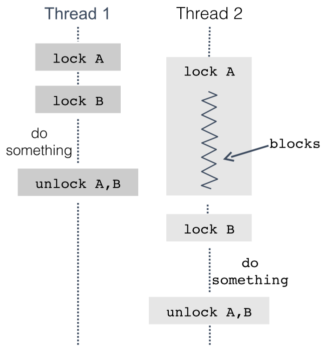


Figure 3.10: Lock ranking

Using lock ranking requires three steps:

1. Find all locks in a program.
2. Number them in the order ("rank") in which they should be acquired
3. Verify that no lock is acquired out of order, via e.g. the use of debug assertions and extensive testing.

This technique is difficult to implement, and cannot be used in every case. An example of its use is in the VMware virtualization product, where several hundred (as of when I worked there in 2007) locks are ranked in order, and beta builds will assert and crash if a lower-priority lock is acquired while holding a higher-priority one.

Review Questions

3.5.1. Given a set of processes, deadlock occurs when:

a) Each process in the set is blocked waiting for a resource (i.e. lock) held by another process in the set
b) Each process is waiting on the same resource, and that resource is held by a process not in the set
c) One of the processes terminates

3.5.2. Deadlock can be prevented by ensuring that processes always acquire locks in the same order: *true / false*

3.5.3. Deadlock can be prevented by ensuring that a process only holds one lock at a time: *True / False*

3.6 Monitors

Semaphores do a good job of solving simple problems like the bounded buffer, and in theory are sufficient to solve any synchronization problem[10], but become quite complicated to use when a problem can't be solved by simple counting. As an example, we'll look at what we'll call the *Weighted Bounded Buffer Problem*, which differs from the bounded buffer problem in these ways:

1. Each item has a weight, `item.weight`
2. The total weight of the items in the buffer cannot exceed N. If `buffer.put()` would cause this limit to be exceeded, then it will block until enough space is available.

At first it seems like it would be sufficient for `put` and `get` to call `signal` and `wait` W times if W is the weight of the item being added or removed; however this could cause problems if two threads called `put` or `get` simultaneously, and is not possible at all if `weight` is a continuous (i.e. floating point) value. Unlike the simple case, we're going to have to write our own code to maintain counts and make decisions about when to sleep, and if we do this with semaphores it's going to be quite ugly.

Instead we introduce a programming language feature for synchronization called a *monitor*. Unlike mutexes and semaphores, which are operating system-defined types, a monitor is a special type of user-defined object or class, where the language provides support for constructing user-defined synchronization behavior.

In particular, a monitor has (a) special instance variables called *conditions*, which support the methods `wait`, `signal`, and `broadcast`, and (b) a per-instance *implicit mutex*, which ensures that only one thread is *in* the monitor (instance) at any one time, executing method code. More precisely, what we mean by this is:

- A thread *enters* the monitor by entering one of its methods. Any number of threads can try to invoke methods on the same instance at once, but only one will get through and begin to execute method code.
- A thread *leaves* the monitor when it returns from a method. This is pretty obvious.
- A thread also *leaves* the monitor when it calls wait on any of the instance condition variables. This is less obvious, but important, as otherwise no other thread would be able to enter the monitor to wake it up.

[10]Or at least any that can be solved by other techniques described in this text.

```
monitor weighted_bb:
    condition C_put, C_space, C_get
    total = 0
    space_needed = 0
    buffer

    method put(item):
1       while space_needed > 0
1           wait(C_put)
        space_needed = item.weight
2       while item.weight + total > max
2           wait(C_space)
        buffer.add_tail(item)
        total = total + item.weight
4       signal(C_get)
1       space_needed = 0
1       signal(C_put)

    method get():
3       while total == 0
3           wait(C_get)
        item = buffer.remove_head()
        total = total - item.weight
2       if total + space_needed <= max
2           signal(C_space)
        return item
```

Figure 3.11: Monitor implementation of weighted bounded buffer

- A thread then *enters* the monitor again when it returns from wait. Note that this can't actually happen until *after* the thread which is currently in the monitor—usually the one that called notify—leaves the monitor.

When a thread calls `wait(C)` it goes to sleep, and must be woken by a future call to notify or broadcast. When a thread calls `signal(C)`, a thread waiting on C is made eligible to return from `wait()`, and will do so as soon as it gets a chance to re-enter the monitor. On most systems threads waiting on C are picked in FIFO order, but this is not guaranteed. Finally, when a thread calls `broadcast(C)`, all threads waiting on C are made eligible to return from `wait()`, and again will do so as soon as they are able to. If either notify or broadcast are called on a condition with no waiting threads, nothing will happen and no error will occur. Unlike calling `signal` on a semaphore with a positive count, the call won't be "saved up" for future calls to `wait`. And unlike unlocking a free mutex, it won't result in an error.

Here we see a monitor implementation of the weighted bounded buffer. Despite the increased complexity of the problem, this solution is only slightly longer than the semaphore solution to the simpler problem. A more detailed description of its operation:

(1) The lines marked 1 serve as "gatekeepers": only one thread at a time can be executing the lines in the middle, including the `wait(C_space)` call. After leaving this section of code we signal the next waiting thread, if any.

(2) Here a thread calling `put()` waits for space, and `get()` wakes it up if it has created enough space by removing an object.

(3) Here a thread calling `get()` waits for an item if the buffer is empty,

and is signalled by a thread at (4) calling `put()`. Note that this interaction is simpler, because (as in the simple bounded-buffer case) there is a one-to-one relationship between items and calls to `get()`.

3.7 Using Conditions

Like many programming features, there are different ways to use condition variables, and some of them are "better" than others, being easier to understand, write correctly, and debug. In this class we teach the following rule for using them:

- Each condition C is associated with a boolean predicate P, and that condition is used in "guards" of the form `while (not P) wait(C)`, so that after the guard has been executed the invariant P is true.

In the example above, for instance, `C_space` is associated with the predicate $item.weight + total \leq max$, or in other words that there is enough room for the item. If there isn't then we wait; immediately after passing these two lines (marked 2 in the listing) we can be sure that there is indeed enough room.

How can we be sure? If the predicate is true, and we don't have to wait, the answer is trivial. In the other case, we need to make sure that every piece of code which *might* make the predicate become true checks it, and if the predicate actually *has* become true it signals the associated condition variable.

Note that this association only exists in the mind of the programmer, and is not enforced in any way by the programming language. Multithreaded programming would be much easier if we could just wait on the boolean predicate itself, but no one has yet invented a way to do this efficiently. Instead the programmer is responsible for the job of identifying what other pieces of code might make the predicate become true, with the resulting bugs if you miss any cases.

`while (condition)` **vs** `if (condition)`: In Figure 3.11 it would be nice if we could just call `wait(C_put)` or `wait(C_space)` and assume that the associated predicate is true after returning from wait. Unfortunately, it's not really possible, or at least not efficiently—even if mutexes and condition variables preserve FIFO ordering, there's often a window between when a thread calls `signal(C)` and the thread blocked in `wait(C)` returns, where a third thread can call the monitor method and grab the monitor mutex before the second thread is able to acquire it while returning from `wait(C)`.

To handle this race condition we loop checking the predicate and waiting on the condition variable. In the (very rare) case where another thread entered the monitor while we were waking up, and e.g. grabbed whatever thing or resource we were waiting for, we go back to sleep and wait for another one.

Review Questions

3.7.1. A monitor is different from a semaphore in which of these ways:

a) It is a user-defined type, rather than OS-defined
b) It can have multiple queues of waiting threads
c) Both of the above

3.7.2. A thread "leaves" the monitor when:

a) It returns from a method
b) It calls `wait()`
c) It calls `signal()`
d) Answers 1 and 2
e) All of the above.

3.7.3. A condition variable contains a boolean predicate, and a thread waiting on it blocks until that predicate becomes true: *true / false*

Implementing Monitors

So far we've described monitors as a language feature, but if you look at the languages in use today you won't find the 'monitor' keyword anywhere. Java has very limited direct support for monitors—a synchronized class is essentially a monitor with a single condition variable, accessed implicitly via `acquire()` and `release()`. In general, however, you have to implement monitors yourself, using some sort of condition variable object supplied by the operating system or thread library.

POSIX threads[11]: This threading package, provided on Unix-like systems such as Linux and OSX, provides the following types and functions we can use:

```
pthread_mutex_t mutex
pthread_mutex_lock(mutex)
pthread_mutex_unlock(mutex)
pthread_cond_t cond
pthread_cond_wait(cond, mutex)
pthread_cond_signal(cond)
pthread_cond_broadcast(cond)
```

Since the language doesn't provide an implicit monitor mutex, we allocate an *explicit* per-object mutex, locking it on entry to each method and unlocking before returning from the method. Condition variables are also provided directly, e.g. by the pthread_cond_create function; however the thread library cannot know what object instance and mutex a condition variable is associated with, and so we have to pass the mutex explicitly when we wait on a condition. More precisely, the translation (as shown in Figure 3.12) is:

1. (implicit mutex) : create a per-instance mutex `m` which is locked on entry to each method and unlocked on exit. (being careful with multiple exits, or worse yet exceptions)
2. condition variables : translate each to an instance variable of type `pthread_cond_t`
3. `signal(C)`, `broadcast(C)` : `pthread_cond_signal(C)` and `pthread_cond_broadcast(C)`
4. `wait(C)` : `pthread_cond_wait(C, m)` where `m` is the per-instance mutex.

Note that for programming exercises in this class we may implement singleton objects in C, in which case we can simplify our implementation somewhat:

[11]The same threading model is available in C11, with slightly different names—e.g. mutexes are of type `mtx_t`, with functions `mtx_lock` and `mtx_unlock`

```
monitor myclass:                   class myclass {
    condition C1, C2               private:
                                       pthread_mutex_t m;
    method m1():                       pthread_cond_t C1, C2;
        C1.wait()
        C2.signal()                public:
        return                         void m1(void) {
                                           pthread_mutex_lock(&m);
                                           pthread_cond_wait(&C1, &m);
                                           pthread_cond_signal(&C2);
                                           pthread_mutex_unlock(&m);
                                       }
```

Figure 3.12: Implementation of monitor in Posix threads.

- Methods become functions, as there is no need to specify which object instance to apply a method to.
- Instance variables become global variables, because we only need one copy of them, but they must be shared between methods.

```
pthread_mutex_t m;
pthread_cond_t C1, C2;
void m1() {
    pthread_mutex_lock(&m);
    pthread_cond_wait(&C1, &m);
    pthread_cond_signal(&C2);
    pthread_mutex_unlock(&m);
}
```

Listing 3.8: Singleton monitor implementation in C.

Java: In this case we use an instance of *ReentrantLock* (in java.util.concurrent.locks) as our mutex, with methods `lock` and `unlock`. Condition variables are associated with a ReentrantLock (i.e. mutex), so given a ReentrantLock m created to be the per-object mutex, for each condition variable C in the original monitor we create a Condition via m.newCondition(); operations on these conditions are *wait*, *notify*, and *notifyAll*.

```
import ReentrantLock from java.util.concurrent.locks;
class myclass {
   ReentrantLock m = new ReentrantLock();
   Condition C1 = m.newCondition(), C2 = m.newCondition();

   void m1() {
      m.lock();
      C1.wait();
      C2.notify();
      m.unlock();
   }
```

Listing 3.9: Monitor implementation in Java

Python: The module `threading` implements two classes, `Lock` and `Condition`, which we use as above. (note that the methods for `threading.Lock` are `acquire` and `release`) Like Java, conditions are associated with locks at the time of creation, so there is no need to remember to pass the mutex in the `wait()` function.

Review Questions

3.7.1. When implementing a monitor in POSIX threads, you need a separate mutex for each condition variable: *true / false*

3.7.2. Race conditions can occur in monitors because:

a) Multiple threads can be executing methods at the same time
b) The order in which threads enter the monitor may differ
c) Both of the above

```
import threading
class myclass:
   def __init__(self):
      self.m = threading.Lock()
      self.C1 = threading.Condition(self.m)
      self.C2 = threading.Condition(self.m)

   def m1(self):
      self.m.acquire()
      self.C1.wait()
      self.C2.notify()
      self.m.release()
```

Listing 3.10: Monitor implementation in Python

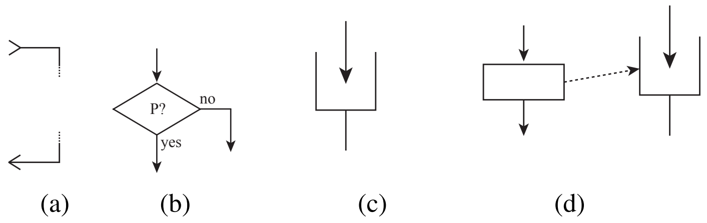


Figure 3.13: Elements of the graphical language: (a) method, (b) choice (if/then statement), (c) condition, and (d) signalling a condition.

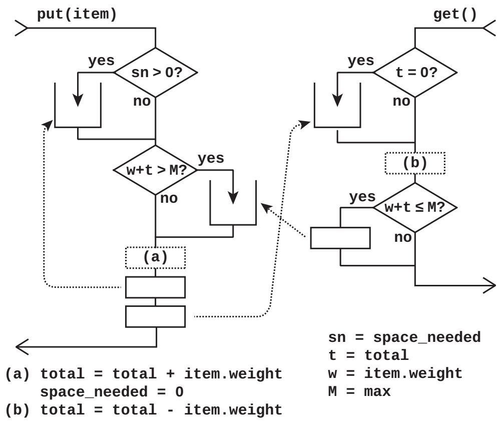


Figure 3.14: Graphical representation for weighted bounded buffer solution shown in Figure 3.11

3.8 Graphical Notation

Reasoning about multi-threaded programs is harder than single-threaded ones. For single-threaded programs most people can visualize how program execution moves from one line of code to another; however in the multi-threaded case you have to be aware of many possible copies of the same code, each possibly executing a different line.

In Figure 3.13 we see the elements of a graphical representation for a monitor, which allows us to see more directly how different threads interact in the execution of a multi-threaded program. Each method is represented by a path (a), which may involve decisions (b), waiting on conditions (c), and signalling those conditions (d).

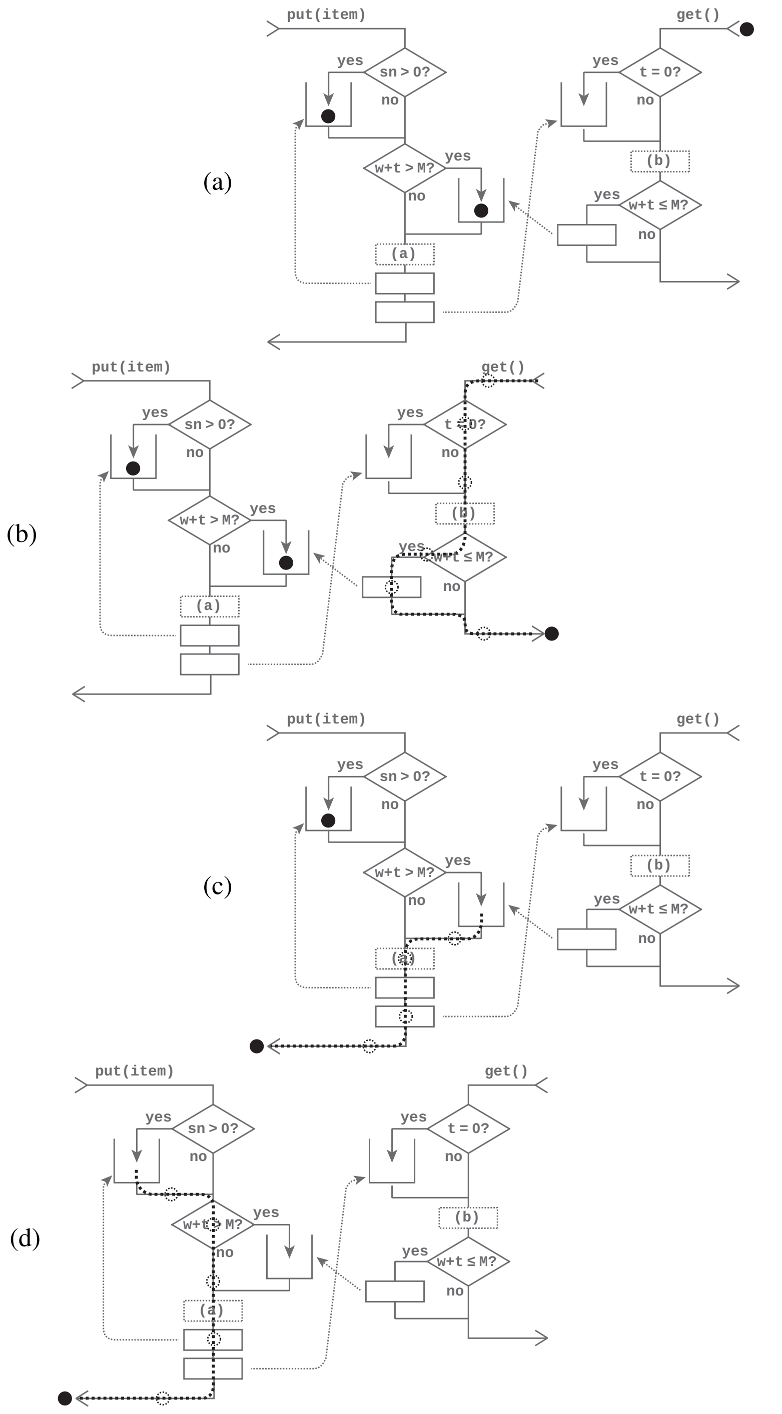


Figure 3.15: Multiple threads shown as black dots moving through the monitor code.

In Figure 3.14 we see the weighted bounded buffer solution from Figure 3.11 represented in this graphical notation, and in Figure 3.15 we see multiple threads moving through this representation.

(Note that the figures have been simplified slightly by using `if (!P) wait(C);` instead of `while (!P) wait(C)`.)

3.9 Putting it all together

Most of the synchronization techniques discussed in this chapter are applicable to multi-threaded application programs, rather than operating systems themselves; however synchronization and the prevention of race conditions are still key techniques within an OS.

Condition variables and signal(): The I/O wait mechanism is an example of this. When the shell invokes the `read` system call to read characters from the keyboard, the process is removed from the active list and placed on a wait queue in the kernel; the keyboard interrupt handler then wakes a process waiting on this queue when a character is received. The semantics of this I/O wait queue and the operation to wake a process from it are identical to those of a condition variable with `wait` and `signal`. (the design choices are similar, too. Simple operating systems may use the equivalent of `broadcast`, waking all processes waiting on any sort of I/O and having each of them re-check the condition they are waiting on before going to sleep, while for highest performance more complete OSes have separate wait queues per I/O source, and when data arrives a single waiting process will be woken.)

Mutexes: An operating system is full of potential race conditions, and heavy use is made of locking mechanisms to prevent errors or crashes. Asynchronous events can occur due to timer or I/O interrupts, and on a multi-core CPU there can be OS code running on multiple cores at the same time. In either case it is essential to protect key OS data structures, such as the list of active processes, which is typically implemented as a singly- or doubly-linked list.

Data structures such as this will typically be protected by a combination of spinlocks and disabling interrupts—e.g. to modify the active process list, OS code will (1) disable interrupts, (2) acquire a spinlock which guards that list, (3) perform the modifications, (4) release the spinlock and (5) re-enable interrupts. (Interrupts are typically disabled while an interrupt handler executes, so when accessing these data structures from an interrupt handler it is sufficient to acquire the spinlock.)

When switching to the next runnable process, it's necessary to protect not only the active process list, so that it doesn't get corrupted, but to also protect the variable identifying the current process on each CPU, to prevent two processes from being assigned to the same CPU at the same time. A simple way of doing this is to have a `schedule()` function which is called under a lock, and which pops the next runnable process off the active list, makes it the current process, and switches to it; e.g. an implementation using simple round-robin scheduling might be as shown

```
[e.g. yield:]
      ...
   lock(plist_lock);
   schedule();
   unlock(plist_lock);
      ...
```

```
schedule() {
   active_tail->next = current;
   active_tail = current;
   current = active_head;
   active_head = active_head->next;
   switch_to(current);
}
```

Figure 3.16: Simple round-robin thread scheduler

in Figure 3.16.

Note that the lock can't be "encapsulated" within `schedule` and hidden from other code, because special handling is required when creating processes—when a new process begins it will execute a "trampoline" function, rather than the second half of the `schedule` function, and must drop the lock that was acquired when switching to it.

Finally, deadlocks are a risk when implementing an operating system. In many cases the objects of contention are not mutexes themselves, but resources such as pages of memory E.g. consider the case[12] where a process tries to allocate a page of memory when (almost) all pages are in use. The OS finds a page it can "steal" from another process after writing its contents to disk; however if that page is associated with a network file, the OS may need to temporarily allocate another page of memory in order to send the network message to write it back.

The solution to this is to reserve the last few blocks of memory to various high-priority uses. This works in much the same way as lock ranking, because the original request is made at low priority (i.e. by the process) and thus can't acquire and hold the resources which would be needed by the higher-priority page-out and networking tasks.

[12]Yes, I know we haven't covered some of the parts of this yet, but we'll get to them in the next chapter...

Answers to Review Questions

3.2.1 *(race conditions can be detected by exhaustive testing)* False. The outcome of a race condition is determined by the internal order in which threads execute instructions within a program. This internal ordering will be affected by the order in which inputs are received, but it will also depend on uncontrollable events such as interrupts, cache behavior, etc.

3.2.2 *(implementing withdraw method)* (2), lock mutex `m`. Both deposit and withdrawal modify the same account balance, and so no combination of the two may be allowed to execute simultaneously.[13]

3.3.1 (3), "CPU 1: R1=2, CPU 2: R1=0, memory: 1". In this case CPU 2 executes the SWAP instruction before CPU 1.

3.3.2 (3), an OS-defined lock object. (note that although spinlocks are a simple kind of mutex, they are not the only kind)

3.4.1 (1) There is no coordination between one thread making room (or adding an item to an empty buffer) and another thread waiting for room or a new item, so the only thing it can do is return EMPTY or FULL.

3.5.1 (1), each process is blocked waiting for a resource held by another process in the set.

3.5.2 True. Ranking locks in order prevents the formation of a circular wait.

3.5.3 True. If a process never acquires more than one lock, then it never holds a lock while waiting for another one.

3.7.1 (3), both of the above. Monitors are user-defined classes, and each condition variable in a monitor is a separate queue that threads can wait on.

3.7.2 (4), it leaves the monitor both when returning from a method and when calling `wait`. It does **not** leave the monitor when calling `signal`.

3.7.3 False. A condition variable has no value, and a thread waiting on it will only wake when another thread calls `signal` or `broadcast`.

3.7.1 False. A single mutex is used to guard the instance variables of the monitor, and is passed in `pthread_cond_wait` when waiting on any of the condition variables of that instance.

3.7.2 (2), the order in which threads enter the monitor may differ. (since two threads cannot execute code in the same monitor at the same time)

[13]Note that your customers may appreciate the lock-less version, as it will occasionally forget that a withdrawal was made.

Chapter 4

Virtual Memory

In chapter 2 we discussed operating systems basics such as I/O, program loading, and context switching primarily for a simple computer with a single *physical address space*. By this we mean that the bits in an address register—for instance the program counter—are the same bits that go out over wires on the motherboard to DIMM sockets and select a particular location in a memory chip, so that no matter what process is executing, the same address (e.g. 0x1000) always refers to the same memory location.

4.1 Base and Bounds translation

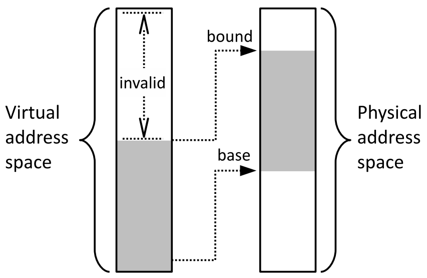


Figure 4.1: Base and bounds translation

We first looked at direct physical addressing, where no matter which process is executing, the same address (e.g. 0x1000) refers to the same memory location. In addition we reviewed a very simple form of address translation, shown here in Figure 4.1, where base and bounds registers are used to relocate a section of the *virtual address space*—the addresses seen by the program, corresponding to values in the CPU registers—to somewhere else in the physical address space. By changing these translations the operating system can create multiple virtual address spaces, one per process; however there is still only one physical address space, uniquely identifying each byte in each memory chip. In this chapter we introduce *paged address translation*, a more complex address

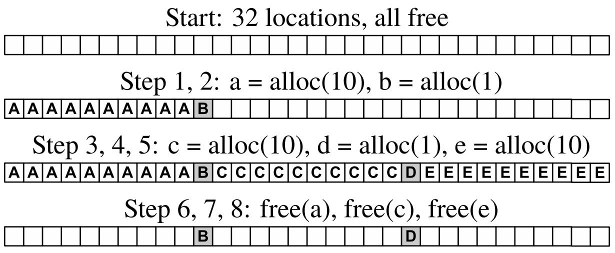

Start: 32 locations, all free
Step 1, 2: a = alloc(10), b = alloc(1)
A A A A A A A A A A B
Step 3, 4, 5: c = alloc(10), d = alloc(1), e = alloc(10)
A A A A A A A A A A B C C C C C C C C C C D E E E E E E E E E E
Step 6, 7, 8: free(a), free(c), free(e)
B D

Figure 4.2: Memory fragmentation example - after step 8 there are 30 free locations, but the largest range that can be allocated is 10.

translation mechanism used by most modern CPUs, and present the 32-bit Intel implementation as an example.

Limitations of base+bound translation: Modern hardware and operating systems provide a very similar process address space model, but no longer use base and bounds registers for address translation[1], despite it being simple, cheap, and quite possibly faster than alternate methods. There are a number of reasons why base and bounds translation is no longer used, but the fundamental reason is memory fragmentation.

Base and bounds address translation requires a contiguous memory region for each process. If memory is allocated and de-allocated in chunks of different sizes and at different times, then it can become *fragmented* so that even if large amounts of memory are free, it will be divided into smaller fragments, separated by longer-lived small allocations, as seen in Figure 4.2.

In the last line, you can see that only 2 units of memory (out of 32) remain allocated, but the largest amount that can be allocated at one time is 10 units. If all allocation requests are small, this might not be a problem; however, in an operating system it is common to have one or two very large processes (e.g., a web browser and word processing software), and many small, long-running processes (e.g., the on-screen battery display or wifi signal strength indicator). In this case, large memory allocations may fail, even when there is enough total memory free, because long-lived small allocations fragment the available contiguous memory into smaller pieces.

[1]Not even on Intel CPUs, which support base+bounds translation using *segment registers*. Nearly every operating system running on these CPUs sets base=0 and bound=max as one of the very first steps in hardware initialization.

4.2 Paging - Avoiding Fragmentation

The fragmentation in Figure 4.2 is termed *external fragmentation*, because the memory wasted is *external* to the regions allocated. This situation can be avoided by *compacting* memory—moving existing allocations around, thereby consolidating multiple blocks of free memory into a single large chunk. This is a slow process, requiring processes to be paused, large amounts of memory to be copied, and base+bounds registers modified to point to new locations[2].

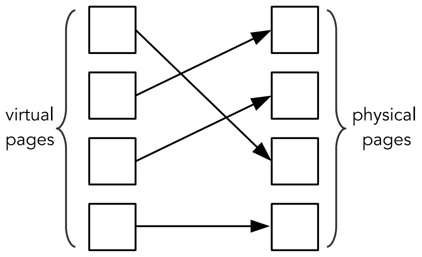


Figure 4.3: Paged memory allocation

Instead, modern CPUs use *paged address translation*, which divides the physical and virtual memory spaces into fixed-sized pages, typically 4KB, and provides a flexible mapping between virtual and physical pages, as shown in Figure 4.3. The operating system can then maintain a list of free physical pages, and allocate them as needed. Because any combination of physical pages may be used for an allocation request, there is no external fragmentation, and a request will not fail as long as there are enough free physical pages to fulfill it.

Internal Fragmentation

Paging solves the problem of external fragmentation, but it suffers from another issue, *internal fragmentation*, because space may be wasted *inside* the allocated pages. E.g. if 10 KB of memory is allocated in 4KB pages, 3 pages (a total of 12 KB) are allocated, and 2KB is wasted. To allocate hundreds of KB in pages of 4KB this is a minor overhead: about $\frac{1}{2}$ a page, or 2 KB, wasted per allocation. But internal fragmentation makes this approach inefficient for very small allocations (e.g. the `new` operator in C++), as shown in Figure 4.4. (It is also one reason why even though most CPUs support multi-megabyte or even multi-gigabyte "huge" pages, which are slightly more efficient than 4 KB pages, they are rarely used.)

[2]This is similar to *garbage collection* in Java and other languages; however in that case pointers to the garbage-collected memory must be changed to point to the new locations.

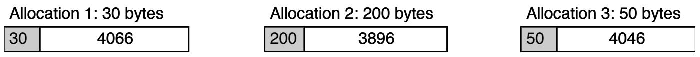


Figure 4.4: Internal fragmentation for very small allocations—total allocated memory is 30+200+50=280 bytes, overhead is 12008 bytes.

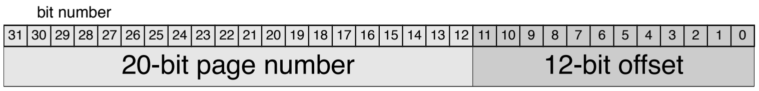


Figure 4.6: Page number and offset in 32-bit paged translation with 4KB pages

4.3 Paged Address Translation

We examine a single model of address translation in detail: the one used by the original Pentium, and by any Intel-compatible CPU running in 32-bit mode. It uses 32-bit virtual addresses, 32-bit physical addresses, and a page size of 4096 bytes. Since pages are 2^{12} bytes each, addresses can be divided into 20-bit page numbers and 12-bit offsets within each page, as shown in Figure 4.6

The Memory Management Unit (MMU) maps a 20-bit virtual page number to a 20-bit physical page number; the offset can pass through unchanged, as shown in Figure 4.5, giving the physical address the CPU should access.

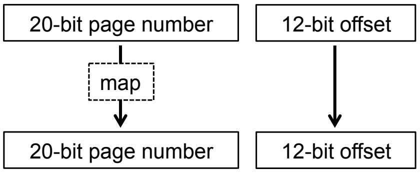


Figure 4.5: 32-bit paged address translation

Although paged address translation is far more flexible than base and bounds registers, it requires much more information. Base and bounds translation only requires two values, which can easily be held in registers in the MMU. In contrast, paged translation must be able to handle a separate mapping value for each of over a million virtual pages. (although most programs will only map a fraction of those pages) The only possible place to store the amount of information required by paged address translation is in memory itself, so the MMU uses page tables in memory to specify virtual-to-physical mappings.

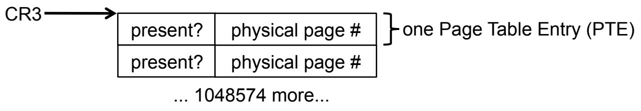


Figure 4.7: Single-level 32-bit page table

```
PA = translate(VA):
    VPN, offset = split[20 bits, 12 bits](VA)
    PTE = physical_read(CR3 + VPN*sizeof(PTE), sizeof(PTE))
    if not PTE.present:
        fault
    return PTE.PPN + offset
```

Listing 4.1: Address translation pseudo-code for single-level page table.

Single-level Page Table

One of the simplest ways to structure a page table for mapping 20-bit page numbers is as a simple array with 2^{20} entries. With this configuration, each virtual page has an entry, and the value in that entry is the corresponding physical page number, as seen in Figure 4.7. This single-level table is located in physical memory, and the MMU is given a pointer to this table, which is stored in an MMU register. (On Intel-compatible CPUs, the page table pointer is Control Register 3, or CR3.) This is shown in Figure 4.7, where we see the first two entries in a 2^{20} or 1048576-entry mapping table. In addition to the translated page number, each entry contains a *P* bit to indicate whether or not the entry is "present," i.e., valid. Unlike in C or Java we can't use a special null pointer, because 0 is a perfectly valid page number[3].

In Figure 4.1 we see pseudo-code for the translation algorithm implemented in an MMU using a single-level table; VA and PA stand for virtual and physical addresses, and VPN and PPN are the virtual and physical page numbers.

Note that this means that every memory operation performed by the CPU now requires two physical memory operations: one to translate the virtual address, and a second one to perform the actual operation. If this seems inefficient, it is, and it will get worse. However, in a page or two we'll discuss the *translation lookaside buffer* or TLB, which caches these translations to eliminate most of the overhead.

[3]Besides, the hardware designers would rather check the value of a single wire than compare a whole bunch of bits at once.

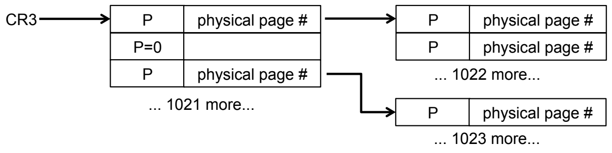


Figure 4.8: Two-level page table for 32-bit addresses and 4 KB pages

The single-level page table handles the problem of encoding the virtual-to-physical page map, but causes another: it uses 4 MB of memory per map. Years ago (e.g. in the mid-80s when the first Intel CPUs using this paging structure were introduced) this was entirely out of the question, as a single computer might have a total of 4 MB of memory or less. Even today, it remains problematic. As an example, when these notes were first written (2013), the most heavily-used machine in the CCIS lab (login.ccs.neu.edu) had 4 GB of memory, and when I checked it had 640 running processes. With 4 MB page tables and one table per process, this would require 2.5GB of memory just for page tables, or most of the machine's memory. Worse yet, each table would require a contiguous 4MB region of memory, running into the same problem of external fragmentation that paged address translation was supposed to solve.

2-level Page Tables

To fix this, almost all 32-bit processors (e.g. Intel, ARM) use a 2-level page table, structured as a tree, as seen in Figure 4.8.

The top ten bits of the virtual page number index into the top-level table (sometimes called the *page directory*), which holds a pointer to a second-level table. The bottom ten bits of the virtual page number are used as an index into this second-level table, giving the location where the actual physical address will be found. At first glance, it appears that this structure takes just as much space as a single-level table. To map a full 4 GB of memory, it still requires 4 MB (plus 1 additional page) for page tables. But if a process only needs a small amount of memory, most of the entries in the top-level directory will be empty (shown here as P=0), and only a small number of second-level tables will be needed; small-memory processes will thus have small page tables. And since the table is made out of individual pages, we can use whatever set of 4 KB pages are available, instead of needing a contiguous 4 MB block.

Note that this is a key characteristic of almost every page table implementation: a page table is made up of pages, allowing the same pool of free pages to be used for both user memory allocation and for page tables themselves. In addition it means that each sub-table starts at the beginning of a page and fits within that page, which simplifies array lookups when translating a page number.

2-Level Page Table Operation

In Figure 4.9 we see a page table constructed of 3 pages: physical pages 00000 (the root directory), 00001, and 00003. Two data pages are mapped: 00002 and 00004. Any entries not shown are assumed to be null, i.e., the present bit is set to 0. As an example we use this page table to translate a read from virtual address 0x0040102C.

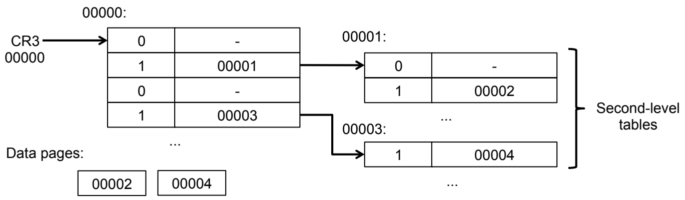
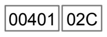


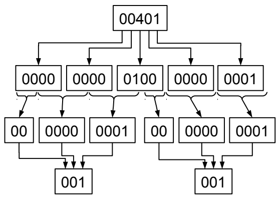
Figure 4.9: 2-level Page Table Example

The steps involved in translating this address are:

1) Split the address into page number and offset



2) Split the page number into top and bottom 10 bits, giving `0x001` and `0x001`. (in the figure the top row is hex, the middle two rows are binary, and the bottom is hex again.)



3) Read entry `[001]` from the top-level page directory (physical page 00000) (note sizeof(entry) is 4 bytes):

```
address = start [00000000] + index [001] * sizeof(entry)
read 4 bytes from physical address 00000004 (page 00000, offset 004)
result = [p=1, pgnum = 00001]
```

4) Read entry `[001]` from the page table in physical page 00001:

```
address = 00001000 + 001*4 = 00001004
read 4 bytes from physical address 00001004
:result = [p=1, pgnum = 00002]
```

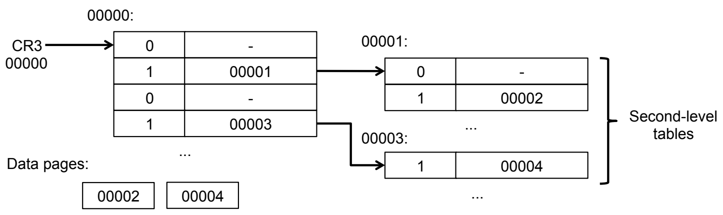


Figure 4.10: Reference page table for review questions

This means that the translated physical page number is 00002. The offset in the original virtual address is 02C, so combining the two we get the final physical address, 0000202C.

Review questions

4.3.1. (all numbers are in hex) When translating the address 0x00C001C0, the virtual page number is: a) 0x00C00 b) 0x1C0 c) 0x008

4.3.2. Referring to the image in Figure 4.10, to translate the address 00C001C0, splitting 00C00 into its top and bottom 10 bits gives 003, 000. Which page table entry is read from the top-level page directory?

a) P=0, PPN=null
b) P=1, PPN=00001
c) P=1, PPN=00003

4.4 Translation Look-aside Buffers (TLBs)

The 2-level table address translation processes you just learned about is highly inefficient, even more so than the single-level table. Even if MMU accesses to memory can be satisfied from the L1 cache, this will still slow down the CPU by a factor of three or more. To reduce this inefficiency, a special-purpose cache called the Translation Look-Aside Buffer (TLB) is introduced. Instead of holding memory values, like the L1 and L2

A famous computer science quote attributed to David Wheeler is: “All problems in computer science can be solved by another level of indirection,” to which some add “except the performance problems caused by indirection.” A corollary to this is that most performance problems can be solved by adding caching. How are these quotes applicable to paged address translation?

caches, the TLB holds virtual page number to physical page number mappings. The TLB is typically very small: examining the machines I have readily available, I see a TLB size ranging from 64 mappings (on certain Intel Atom CPUs) to 640 mappings on Core i7 and Xeon E7 CPUs. One reason for this small size is because the TLB has to be very fast—they are needed for every memory operation before the CPU can look in its cache for a value.

Using the TLB, the translation process now looks like this:

```
translate VA -> PA:
    (VPN, offset) = split([20,12],VA)
    if VPN is in TLB:
        return TLB[VPN] + offset
    (top10, bottom10) = split([10,10],VPN)
    PDE = phys_read(CR3 + top10*4)
    PTE = phys_read(PDE.pg<<12 + bottom10*4)
    PPN = PTE.pg
    add (VPN->PPN) to TLB, evicting another entry
    return PPN + offset
```

Listing 4.2: Paged address translation with TLB

where PDE is the page *directory* (i.e. top-level) entry, PTE is the page *table* (second-level) entry, and VPN, PPN are virtual and physical page numbers as before.

How well does this perform? If all of the code and data fits into 640 pages (about 2.5MB) on a high-end machine, all translations will come out of the TLB and there will be no additional overhead for address translation. If the *working set* (the memory in active use) is larger than this then some accesses will miss in the TLB and require page-table lookup in memory; however in most cases the translated mapping will be used many times before being evicted from the TLB, and the overhead of accessing in-memory page tables will be modest. (In addition, note that MMU accesses to the page table go through the cache, further speeding up the translation process)

4.5 TLB Consistency

Like any other cache, a TLB only functions correctly if it is consistent, i.e. the entries in the TLB accurately reflect the in-memory values (i.e. page tables) which they are caching. Since the values loaded into the TLB come from a page table in memory at the address identified by CR3, the values may become invalid if either (a) the page table values in memory change (due to CPU writes) or (b) CR3 is modified, so that it points to a different page table. In other words, inconsistencies can arise due to:

Individual Entry Modifications: Sometimes the OS must modify the address space of a running program, e.g. during demand paging (covered below), where the OS maps in new pages and un-maps others. When changing the page table in memory, the OS must ensure that the TLB is not caching a copy of the old entry.

Context switches: The OS provides each process with a separate *virtual address space*, or set of virtual to physical mappings; the same virtual address may be mapped to a different physical memory location in each process. (i.e. to a memory location "owned" by that process.) When switching between processes the OS changes CR3 to point to the address space of the new process, and it's clearly important for both security and correctness to ensure that the MMU uses these mappings, not the old ones.

Preventing TLB Inconsistencies

The issue of modifications can be solved in a fairly straightforward way: the MMU provides one instruction to flush the TLB entry for a particular page, and another to flush the entire TLB (e.g. if a large number of mappings are modified). When entries are flushed from the TLB, there is almost always a performance impact, because of the extra memory accesses needed to reload those entries the next time they are required. In this case, this overhead is not that significant, because (a) the OS is already spending a lot of time modifying the page table, and (b) it doesn't do this very often, anyway.

However, the issue with context switches is harder to solve. The easy solution is to ignore the performance overhead and flush the entire TLB on every context switch, as is done on most Intel-compatible CPUs. With a 500-entry TLB and a 4-level page table[4], this results in throwing away 2000 memory accesses worth of work on each context switch. Another solution is to tag each TLB entry with an identifier (an Address Space ID or ASID) identifying the context in which it is valid, allowing entries from multiple contexts to remain in the TLB at once. A special MMU register specifies the ASID of the

Note that measuring the "cost" of an OS operation is often problematic. In a case like this, the operation may complete quickly, but cause other operations to slow down.

[4]Both values typical of 64-bit desktop CPUs.

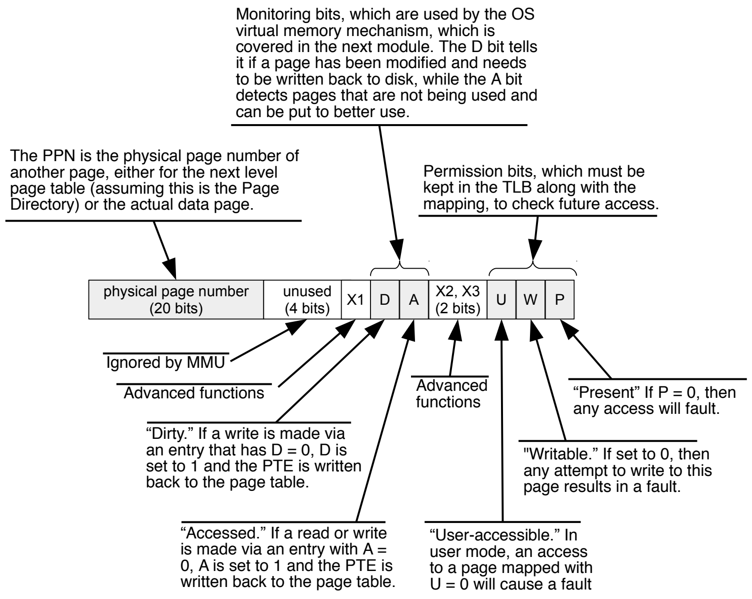

Monitoring bits, which are used by the OS virtual memory mechanism, which is covered in the next module. The D bit tells it if a page has been modified and needs to be written back to disk, while the A bit detects pages that are not being used and can be put to better use.
The PPN is the physical page number of another page, either for the next level page table (assuming this is the Page Directory) or the actual data page.
Permission bits, which must be kept in the TLB along with the mapping, to check future access.
physical page number (20 bits)
unused (4 bits)
X1
D
A
X2, X3 (2 bits)
U
W
P
Ignored by MMU
Advanced functions
Advanced functions
"Present" If P = 0, then any access will fault.
"Dirty." If a write is made via an entry that has D = 0, D is set to 1 and the PTE is written back to the page table.
"Writable." If set to 0, then any attempt to write to this page results in a fault.
"Accessed." If a read or write is made via an entry with A = 0, A is set to 1 and the PTE is written back to the page table.
"User-accessible." In user mode, an access to a page mapped with U = 0 will cause a fault

Figure 4.11: 32-bit Intel page table entry (PTE).

current process, and entries tagged with other ASIDs are ignored. If a process is interrupted for a short time, most of its TLB entries will remain cached, while the ASID field will prevent them from being mistakenly used by another process[5].

Page Table Entries

The components of a 32-bit Intel page table entry are shown in Figure 4.11; for more information you may wish to refer to `http://wiki.osdev.org/Paging`.

Page Permissions - P, W, and U bits

Page tables allow different permissions to be applied to memory at a per-page level of granularity.

P=0/1 - If the present bit is zero, the entry is ignored entirely by the MMU, thus preventing any form of access to the corresponding virtual page.

[5]ASIDs are supported in most modern x86 processors as part of hardware virtualization extensions, which are discussed (in not very much detail) later in this book.

W = 0/1 - Write permission. If the W bit is zero, then read accesses to this page will be allowed, but any attempt to write will cause a fault. By setting the W bit to zero, pages that should not be modified (i.e., program instructions) can be protected. Since correctly-functioning programs in most languages do not change the code generated by the compiler, any attempt to write to such a page must be a bug, and stopping the program earlier rather than later may reduce the amount of damage caused.

U = 0/1 - User permission. If the U bit is zero, then accesses to this page will fail unless the CPU is running in supervisor mode. Typically the OS kernel will "live" in a portion of the same address space as the current process, but will hide its code and data structures from access by user processes by setting U=0 on the OS-only mappings.

Page Sharing

What happens if a single physical memory page is mapped into two different process address spaces? It works just fine. Each process is able to read from the page, and any modifications it makes are visible to the other process, as well. In particular, note that the MMU only sees one page table at a time, and doesn't care how a page is mapped in a page table that might be used at some point in the future. If the two processes are running on different CPU cores, then each core has a separate MMU and will not know or care what translations the other cores are using[6].

> A question for the reader - why doesn't sharing read-only pages violate the security principle of preventing access from one process to another's memory space?

There are two ways in which page sharing can be used:

Information sharing: Some databases and other large programs use memory segments shared between processes to efficiently pass information between those processes.

Memory saving: Most processes use the same set of libraries to communicate with the OS, the graphical interface, etc., and these libraries must be mapped into the address space of each process. But most of the memory used by these libraries (program code, strings and other constant data)

[6]Conversely, if two threads from the same process are running on different cores, then the MMU for each core will be pointing at the same page table and thus use the same mappings.

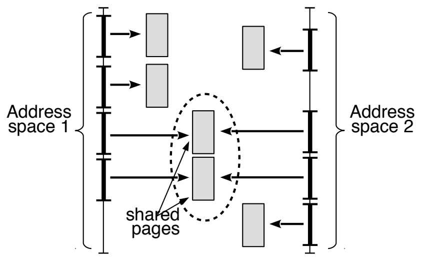


Figure 4.12: Page sharing between two process address spaces

is read-only, and so a single copy can be safely mapped into the address space of each process using the library.

4.6 Page Size, Address Space Size, and 64 Bits

The page size of a processor plays a large role in determining how much address space can be addressed. In particular, assuming that the page table tree is built out of single pages, a 2-level page table can map N^2 pages, where N is the number of page table entries that fit in a single page. Thus, if the address space is about 32 bits, so that a page table entry (physical page number plus some extra bits) can fit in 4 bytes, the maximum virtual memory that can be mapped with a 2-level page table is:

2K pages: 512 (2^9) entries per page = virtual address space of 2^{18} pages of 2^{11} bytes each = 2^{29} bytes (0.5 GB)

4K pages: 1024 (2^{10}) entries per page = virtual address space of 2^{20} pages of 2^{12} bytes each = 2^{32} bytes (4GB)

8K pages: 2048 (2^{11}) entries per page = virtual address space of 2^{22} pages of 2^{35} bytes each = 2^{35} bytes (32GB)

In other words, 2K pages are too small for a 32-bit virtual address space unless the process moves to a deeper page table, while 8K pages are bigger than necessary. (The SPARC and Alpha CPUs, early 64-bit processors, used 8KB pages.)

64-bit Intel-compatible CPUs use 4K pages for compatibility, and 8-byte page table entries, because four bytes is too small to hold large physical page numbers. This requires a 4-level page table, as shown in Figure 4.13.

Since each of the 4 levels maps 9 bits of address, for a total of 36 bits mapped, and the offset is 12 bits, the total virtual address space is 48 bits—not the full 64 bits, but still huge (256 TB). Clearly the penalty for TLB misses is higher in this case than for 32-bit mode, as there are

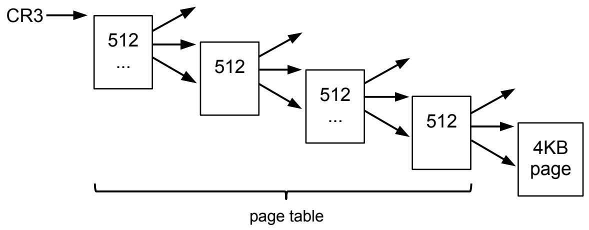


Figure 4.13: 4-level page table for 64-bit mode.

```
char hello[] = ‘‘hello world\n’’;
void _start(void)
{
   syscall(4, 1, hello, 12); /* syscall 4 = write(fd,buf,len) */
   syscall(1);              /* syscall 1 = exit() */
}
```

Listing 4.3: Simple program described in section 4.7

four memory accesses to the page table for a single translation instead of two. To support virtual address spaces greater than 256 TB, it will be necessary to go to a deeper page table, or larger pages, or perhaps another organization entirely.

4.7 Creating a Page Table

To see how a page table is created, we start by examining the virtual memory map of perhaps the simplest possible Linux program, shown in Figure 4.3. This program doesn't use any libraries, but rather uses direct system calls to write to standard output (always file descriptor 1 in Unix) and to exit. In Linux, _start is the point at which execution of a program begins; normally the _start function is part of the standard library, which performs initialization before calling main.

When this program runs and its memory map is examined (using the `pmap` command) you see the following:

```
00110000   4K r-x--   [ anon ]     <- file header - used by OS
08048000   4K r-x--   /tmp/hello   <- .text segment (code)
08049000   4K rwx--   /tmp/hello   <- .data segment
bffdf000   128K rwx-- [ stack ]
```

```
VPN 00110 = 0000 0000 00 01 0001 0000
    top10 = 000 bottom10 = 110
VPN 08048 = 0000 1000 00 00 0100 1000
top10 = 020 bottom10 = 048
VPN 08049 = 0000 1000 00 00 0100 1001
top10 = 020 bottom10 = 049
VPN BFFDF = 1011 1111 11 11 1101 1111
top10 = 2FF bottom10 = 3DF
```

Listing 4.4: Virtual page numbers from the simple 4-segment program

The address space is constructed of a series of contiguous *segments*, each a multiple of the 4 KB page size (although most are the minimum 4 KB here), with different permissions for each. (realistic programs will have many more segments; as an example, the address space for the Nautilus file manager process on my Ubuntu 15.10 system has more than 800 segments.) To create a page table for this program, the first step is splitting the page numbers into top and bottom halves (all numbers given in hex or binary), as shown in Figure 4.4.

The first three segments are one page long; note that the last segment is 32 pages (128 KB), so it uses entries 0x3DF to 0x3FF in the second-level page table.

The program needs four physical pages for the table; assume that pages 0000, 0001, 0002, and 0003 are used for the table, and pages 00004 and up for data/code pages. The actual page table may be seen in Figure 4.14. (note that the choice of physical pages is arbitrary; the page numbers within the page directory and page table entries would of course change if different physical pages were used.)

Review questions

4.7.1. Translating 08049448 in the page table shown in Figure 4.14 requires reading the following physical addresses:

a) 00000080, 00002124
b) 00000020, 00002049
c) 00000080, 00001440
d) 00002080, 00006124

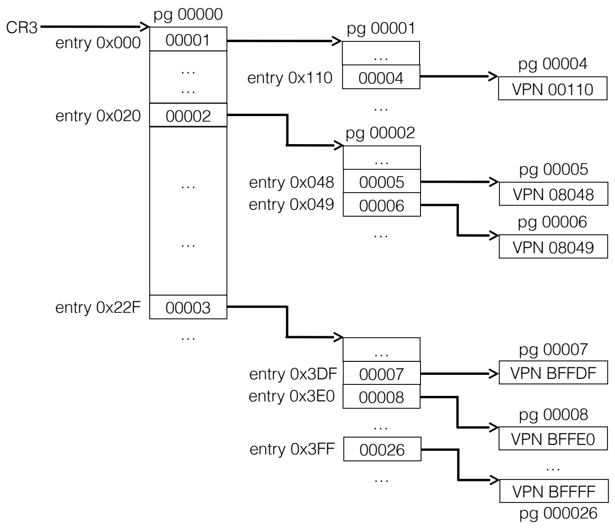


Figure 4.14: Page table corresponding to memory map for Figure 4.3, also used for review questions.

4.8 Page Faulting

In the previous section you saw how the MMU in a Pentium-like CPU determines whether a memory access will succeed:

```
if the top-level entry has P=1
  and is(read) or W=1
  and is(supervisor) or U=1:

  if the 2nd-level entry has P=1
     and is(read) or W=1
     and is(supervisor) or U=1:

       use translated address.
```

If translation fails at any one of the six possible points above (P, W, or U at each level) then a page fault is generated.

Page Faults

A page fault is a special form of exception that has the following two characteristics: first, it is generated when an address translation fails, and second, it occurs in the middle of an instruction, not after it is done, so that the instruction can be continued after fixing the problem which caused the page fault. Typical information that the MMU passes to the page fault handler is:

1. The instruction address when the page fault occurred. (this is the return address pushed on the stack as part of the exception handling process)
2. The address that caused the page fault
3. Whether the access attempt was a read or a write
4. Whether the access was attempted in user or supervisor mode

After the page fault handler returns, the instruction that caused the fault resumes, and it retries the memory access that caused the fault in the first place.

A single instruction can cause multiple, different page faults, of which there are two different types:

Many of the examples in this section are illustrated using Linux, as the source code is readily available, but same principles (although not details) hold true for other modern OSes such as Windows, Mac OS X, or Solaris. In addition, keep in mind that the virtual memory map for a process is a software concept, and will almost certainly differ between two unrelated operating systems. In contrast, the page table structure is defined by the CPU itself, and must be used in that form by any operating system running on that CPU.

- **Instruction fetch:** A fault can occur when the CPU tries to fetch the instruction at a particular address. If the instruction "straddles" a page boundary (i.e., a 6-byte instruction that starts 2 bytes before the end of a page) then you could (in the worst case) get two page faults while trying to fetch an instruction.
- **Memory access:** Once the instruction has been fetched and decoded, it may require one or more memory accesses that result in page faults. These memory accesses include those to the stack (e.g., for CALL and RET instructions) in addition to load and store instructions. As before, accessing memory that straddles a page boundary will result in additional faults.

Handling Page Faults

Operating systems use two primary strategies in handling page faults:

Kill the program. If the access is in fact an error, the default action is to kill the process, so that the page fault handler never returns.[7]

[7]You are no doubt familiar with this process from debugging C programs.

Resolve the fault. The OS modifies the page tables to establish a valid mapping for the failing address, and then returns from the page fault handler. The CPU retries the memory access, which should succeed (or at least continue farther) this time.

In fact, a single instruction can in the worst case result in quite a large number of page faults:

- On an Intel or similar CPU, multi-byte instructions and data may cross page boundaries; e.g. reading a 4-byte integer at address 0x1FFE (occupying bytes 0x1FFE, 1FFF, 2000, and 2001) could trigger page faults on both page 0x1000 and 0x2000.
- Every instruction can fault on instruction fetch; memory instructions like LOAD and STORE can also fault on data access.
- Finally, remember that the stack is in memory, too, so that CALL, PUSH, POP, and RET can all fault if the operation causes an access to a non-mapped stack address.

If the page fault handler updates the page table (to point to an appropriately initialized page of memory) and then returns promptly, the whole page fault process is invisible to the user or programmer.

The page fault handler for an operating system typically only uses the four responses described above—crash, demand-allocate, demand-page, and copy-on-write. More complex page fault mechanisms are used in hardware virtualization, to support virtual machines; those mechanisms will be described later in this book.

Review questions

4.8.1. One instruction can only result in one page fault: *true / false*

4.8.2. Assume a Pentium-like CPU which can (a) load 4-byte words from unaligned (non-multiple-of-4) addresses, and (b) execute unaligned instructions - in particular, this means that an instruction or a data word may cross over a page boundary. In addition, assume (unlike a Pentium) that each instruction can do only one memory load or store in addition to the instruction fetch. What is the maximum number of page faults that could occur for a single instruction?

4.8.3. When accessing memory, virtual addresses are translated to physical addresses (a) by the page fault handler, or (b) by the MMU (memory management unit).

Process Address Space, Revisited

How does the OS know how to handle a page fault? By examining its internal memory map for a process. We've talked briefly about process memory maps earlier, but now we will look in more detail at a specific one, from a fairly recent (kernel 2.6 or 3.0) 32-bit Linux system. A more thorough description of the Linux memory layout can be found at
`http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory`

In earlier chapters we saw how simple operating systems may use separate portions of the address space for programs and for the operating system. The same approach is often used in dividing up the virtual address space in more complex operating systems, as seen in the 32-bit Linux memory map in Figure 4.15. In recent Linux versions running on 32-bit Intel-compatible CPUs, the kernel "owns" the top 1GB, from virtual address 0xC0000000 to 0xFFFFFFFF, and all kernel code, data structures, and temporary mappings go in this range.

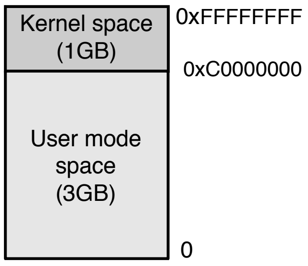


Figure 4.15: Linux 32-bit user/ kernel memory split

The kernel must be part of every address space, so that when exceptions like system calls and page faults change execution from user mode to supervisor mode, all the kernel code and data needed to execute the system call or page fault handler are already available in the current virtual memory map[8] This is the primary use for the U bit in the page table—by setting the U bit to zero in any mappings for operating system code and data, user processes are prevented from modifying the OS or viewing protected data.

Here is the memory map of a very simple process[9], as reported in `/proc/<pid>/maps`:

```
08048000-08049000 r-xp 00000000 08:03 4072226 /tmp/a.out
08049000-0804a000 rw-p 00000000 08:03 4072226 /tmp/a.out
0804a000-0804b000 rw-p 00000000 00:00 0       [anon]
bffd5000-bfff6000 rw-p 00000000 00:00 0       [stack]
```

The memory space has four segments:

08048000 (one page) - read-only, executable, mapped from file *a.out*

[8]In fact the x86 has a way of telling the CPU to switch page tables when an exception occurs, but it's slow. It was used by early Linux versions, but replaced in 1997 or so.

[9]Similar to the program in Figure 4.3, but not exactly the same. I've completely forgotten what program it was, actually.

08049000 (one page) - read/write, mapped from file *a.out*
0804a000 (one page) - read/write, “anonymous”
bffd5000-bfff6000 (33 4KB pages) - read/write, “stack”

Where does this map come from? When the OS creates the new address space in the `exec()` system call, it knows it needs to create a stack, but the rest of the information comes from the executable file itself:

```
$ objdump -h a.out
a.out:    file format elf32-i386

Idx Name         Size      VMA       LMA       File off Algn

  0 .text        00000072 08048094 08048094 00000094 2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata      000006bd 08048108 08048108 00000108 2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data        00000030 080497c8 080497c8 000007c8 2**2
                 CONTENTS, ALLOC, LOAD, DATA
  3 .bss         00001000 08049800 08049800 000007f8 2**5
                 ALLOC
$
```

Executable files on Linux are stored in the ELF format (Executable and Linking Format), and include a header that describes the file to the OS; the information above came from this header. Looking at the file, the following sections can be seen:

`0 ... x93`	various header information	
`00000094 - 00000107`	“.text”	program code
`00000108 - 000007c7`	“.rodata”	read/only data (mostly strings)
`000007c8 - 000007e7`	“.data”’	initialized writable data
(no data)	“.bss”’	zero-initialized data

The BSS section[10]corresponds to global variables initialized to zero; since the BSS section is initialized to all zeros, there is no need to store its initial contents in the executable file.

Executable file and process address space

Here you can see the relationship between the structure of the executable file and the process address space created by the kernel when it runs this

[10]In most compiled languages (e.g. C, C++) global variables which aren’t explicitly initialized have their values set to zero. The compiler and linker lump these values together into a single section, called BSS for an ancient IBM assembly language command that is an abbreviation for something that no one remembers. Since the entire section is going to contain all zero bytes, there is no need to store its contents - just its starting address and length.

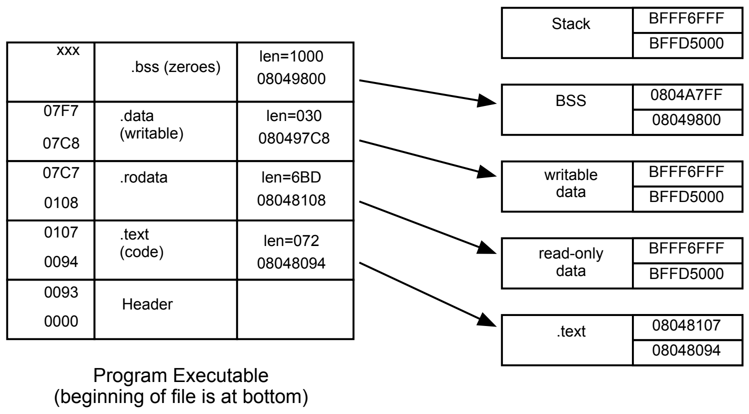


Figure 4.16: Relationship of executable file header to memory map structure

executable. One page (08048xxx) is used for read-only code and data, while two pages (08049xxx and 0804Axxx) are used for writable data.

Review questions

4.8.1. Layout of the per-process address space in operating systems such as Linux is:

a) Determined by the CPU hardware
b) Specified in the executable file header
c) Determined by command-line arguments to the program

4.8.2. When a page fault occurs on an Intel-compatible CPU, the CPU switches from the process address space to the kernel address space: *True / False*

4.8.3. When a page fault occurs on an Intel-compatible CPU, if more than one page fault occurs at the same instruction location the CPU will crash: *True / False*

4.9 Page Fault Handling

In the Linux kernel, the memory map is represented as a list of `vm_area_struct` objects, each corresponding to a separate segment, and each containing the following information:

- Start address
- End+1 address

- Permissions: read/write/execute
- Flags: various details on how to handle this segment
- File, offset (if mapped from a file)

Unlike the page table, which is a simple structure defined by the CPU hardware, the virtual memory map in the OS is a purely software data structure, and can be as simple or complex as the OS writers decide.

With the map from Figure 4.16, the possibilities when the page fault handler looks up a faulting address are:

- No match: This is an access to an undefined address. It's a bug, and the OS terminates the process with a "segmentation fault" error.
- Any page in bff08000-bff29000: These are demand-allocate stack pages. The page fault handler allocates a physical memory page, zeros it (for safety), puts it into the page table, and returns.
- Page 08048000: This page is mapped read-only from the executable file 'a.out,' so the page fault handler allocates a page, reads the first 4KB from 'a.out' into it, inserts it into the page table (marked read-only), and returns.
- Page 08049000: This page is mapped read/write from the executable file. Just like page 08048000, the page fault handler allocates a page, reads its contents from the executable, maps the page in the page table (read/write this time) and returns.
- Page 0804a000: Like the stack, this is demand-allocated and zero-filled, and is handled the same way.

Page Faults in the Kernel

What happens if there is a page fault while the CPU is running kernel code in supervisor mode? It depends.

Although common in the past, modern Windows and Linux systems rarely seem to crash due to driver problems. (Although my Mac panics every month or two.) If you ever develop kernel drivers, however, you will become very familiar with them.

If the error is due to a bug in kernel-mode code, then in most operating systems the kernel is unable to handle it. In Linux the system will display an “Oops” message, as shown in Figure 4.5, while in Windows the result is typically a “kernel panic”, which used to be called a Blue Screen of Death. Most of the time in Linux the process executing when this happens will be terminated, but the rest of the system remains running with possibly reduced functionality.

```
[  397.864759] BUG: unable to handle kernel NULL pointer dereference at
                                                        0000000000000004
[  397.865725] IP: [<ffffffffc01d1027>] track2lba+0x27/0x3f [dm_vguard]
[  397.866619] PGD 0
[  397.866929] Oops: 0000 [#1] SMP
[  397.867395] Modules linked in: [...]
[  397.872730] CPU: 0 PID: 1335 Comm: dmsetup Tainted: G     OE   4.6.0 #3
[  397.873419] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS ...
[  397.874487] task: ffff88003cd10e40 ti: ffff880037080000 task.ti: ffff88003708
[  397.875375] RIP: 0010:[<ffffffffc01d1027>]
[<ffffffffc01d1027>] track2lba+0x27
[  397.876509] RSP: 0018:ffff880037083bd0 EFLAGS: 00010282
[  397.877193] RAX: 0000000000000001 RBX: 0000000000003520 RCX: 0000000000000000
[  397.878085] RDX: 0000000000000000 RSI: 0000000000003520 RDI: ffff880036bd70c0
[  397.879016] RBP: ffff880037083bd0 R08: 00000000000001b0 R09: 0000000000000000
[  397.879912] R10: 000000000000000a R11: f000000000000000 R12: ffff880036bd70c0
[  397.880763] R13: 00000000002e46e0 R14: ffffc900001f7040 R15: 0000000000000000
[  397.881618] FS: 00007f5767938700(0000) GS:ffff88003fc00000(0000)
[  397.915186] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  397.932122] CR2: 0000000000000004 CR3: 000000003d3ea000 CR4: 00000000000406f0
[  397.949459] Stack:
                   ... stack contents and backtrace omitted ...
```

Listing 4.5: Linux kernel “Oops” message due to NULL pointer dereference.

But what about addresses passed by the user in a system call? For example, what if the memory address passed to a `read` system call has been paged out, or not instantiated yet? It turns out that the same page faulting logic can be used in the kernel, as well—the first access to an unmapped page will result in a fault, the process will be interrupted (in the kernel this time, rather than in user-space code), and then execution will resume after the page fault is handled.

But what if the user passes a bad address? We can’t just kill the process partway through the system call, because that would risk leaving internal operating system data structures in an inconsistent state. (Not only that, but the POSIX standard requires that system calls return the EFAULT error in response to bad addresses, not exit.) Instead, all code in the Linux kernel which accesses user-provided memory addresses is supposed to use a pair of functions, `copy_from_user` and `copy_to_user`, which check that the user-provided memory region is valid for user-mode access[11].

In very early versions of Linux the kernel ran in a separate address space where virtual addresses mapped directly to physical addresses, and so these functions actually interpreted the page tables to translate virtual addresses to physical (i.e. kernel virtual) addresses, which was slow but made it easy to return an error if an address was bad. Newer Linux versions map

[11]This is important for security reasons. The chapter on security will talk more about the importance of double-checking user imputs to keep a system secure.

the kernel and its data structures into each process virtual address space, making these functions much faster but more complicated. The speedup is because there is no longer any need to translate page tables in software; instead the two `copy_*_user` functions just perform a few checks and then a `memcpy`. More complicated because if it fails we don't find out about it in either of these functions, but rather in the page fault handler itself. To make this work, if the page fault (a) occurs in kernel mode, and (b) the handler can't find a translation for the address, it checks to see if the fault occurred while executing the `copy_from_user` or `copy_to_user` functions, and if so it performs some horrible stack manipulation to cause that function to return an error code[12].

But what if a page fault occurs in the kernel outside of these two functions? That should never happen, because kernel structures are allocated from memory that's already mapped in the kernel address space. In other words it's a bug, just like the bugs that cause segmentation faults in your C programs. And just like those bugs it causes a crash, resulting in an error message such as the one shown in Figure 4.5. If the kernel was running in a process context (e.g. executing system call code) then the currently-running process will be killed, while if this occurs during an interrupt the system will crash. The equivalent in Windows is called a Blue Screen of Death (although they changed the color several versions back); since almost all Windows kernel code executes in interrupt context, these errors always result in a system crash.

4.10 Shared Executables and Libraries

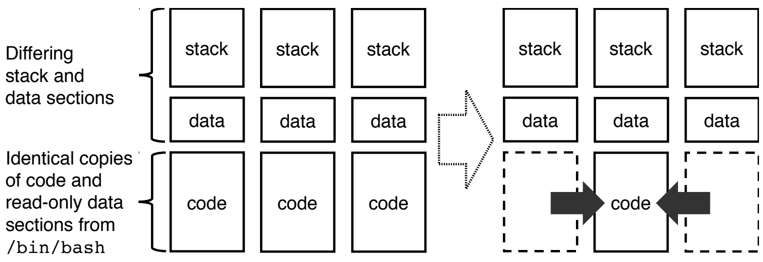
In addition to simplifying memory allocation, virtual memory can also allow memory to be used more efficiently when running multiple processes.

Consider the case of a multi-user computer, where multiple users are running the same program (i.e., the shell, `/bin/bash`) at the same time. If we just follow the rules we've seen above for allocating and filling memory, the memory usage of the three programs will look something like the left-hand side of Figure 4.17.

However since the code section of each process is identical, we can share those pages, giving the picture on the right-hand side of Figure 4.17.[13]

[12]In recent versions it's even more complicated than that, using a table of all the locations in the kernel where the two functions are invoked.

[13]Why are the code sections for each process identical? Because (a) they are mapped from the same file, and so started with the same values, and (b) they are read-only, so those values haven't changed. Is this safe? Doesn't it give a process access to another processes' memory space? It's safe because each process still sees exactly the same data as they would


Differing stack and data sections
Identical copies of code and read-only data sections from /bin/bash
stack
data
code
stack
data
code

(a) without memory sharing (b) with memory sharing

Figure 4.17: Memory usage of three copies of the same program.

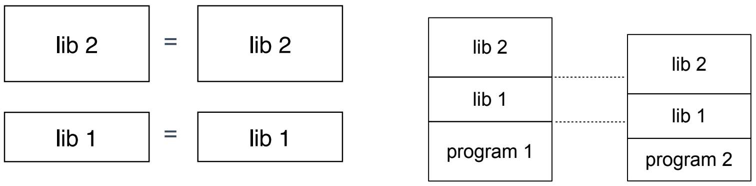


Figure 4.18: Address mismatch when lib1 and lib2 are linked with different programs

How does the OS determine that it can share the same page between two processes? When a page fault happens, and the page fault handler determines that it needs to read (i.e., page 10 from the executable `/bin/bash`) it first searches to see whether that page is already stored in some existing memory page[14]. If so, it can increment a reference count on that page and map it into the process page table, instead of having to allocate a new page and read the data in from the disk. When a process exits, instead of blindly de-allocating any memory mapped by that process, the reference count of each page is decremented, and it is only freed when this count goes to zero, indicating that no other address spaces are mapping that page.

Note that the operating system also provides a way for applications to create memory regions which are explicitly shared between processes, and used for communication between them. This can be used for high-performance communication between processes, and is used in at least one program that people actually use.

without sharing, and can't change that data for other processes.

[14]Most operating systems only check for the case where pages in different processes map to exactly the same page in exactly the same file. If you have two different executable files that happen to be exact copies of each other, the OS will have no idea that they're the same, and will happily load pages from both of them into memory at the same time.

Sharing memory at the program level worked well on multi-user systems, as you just saw, where many people ran the same simple programs (e.g., the shell, editor, and compiler) at the same time. With the advent of graphical interfaces and single-user workstations, it stopped working so well. Instead, now there's a single user running one copy each of several different programs. Worse yet, each program is far more complicated than in the past, as the libraries for interacting with the display, mouse, and keyboard are inevitably larger and more complex than the simple libraries needed to define functions like `printf` for terminal output.[15]

The problem here is that even though your browser, text editor, and email program all use the same libraries, each program ends up being a unique combination of code, combining the actual program code with a specific set of libraries, as seen in Figure 4.18. So even if the operating system *tried* to recognize identical regions in the two files, the differing alignment would make it impossible to share pages between them.

Shared libraries eliminate this wasted space by combining code and libraries in a way that allows sharing in most cases. To do this, the program and the libraries are structured so that different programs can share a single copy of the same library. In simple terms, each library is made to look like a separate program, which means that multiple copies of the same library can be shared, even if the different programs that use it can't be shared.

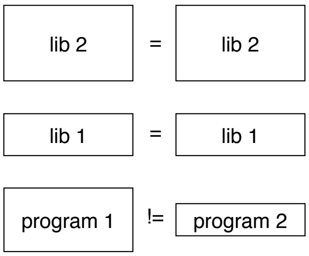


Figure 4.19: Memory sharing with shared libraries

In Figure 4.19 we see how each shared library is given its own region of address space, rather than packing them all into a single segment. The base programs (program1 and program2 below) still differ, but the libraries remain identical and can be shared between address spaces.

This approach is taken in Linux; if we compile the standard "hello world" program shown in Figure 4.6 we can use the `ldd` command to list the libraries which will be loaded at runtime, as seen in Figure 4.7, resulting in the memory map in Figure 4.8.

[15]Example: `xterm` is the original graphical terminal emulator for Unix, and uses very few fancy features. The program itself compiles to about 372KB of machine instructions and some amount of data, but it also uses 26 separate external libraries which add up to 5.6MB of additional program space. A newer program, `gnome-terminal`, uses only 300KB of memory for the program itself, but links in 48 libraries, for a total of 22MB of additional memory. Although both of these examples are taken from Linux, both Apple OS X and Windows use similar large libraries for the graphical interface.

```
#include <stdio.h>
int main()
{
    printf("hello world\n");
}
```

Listing 4.6: Traditional “hello world” program

```
pjd@pjd-fx:/tmp$ ldd a.out
        linux-vdso.so.1 => (0x00007fff99d56000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5a0bb94000)
        /lib64/ld-linux-x86-64.so.2 (0x00005590e6bba000)
```

Listing 4.7: Libraries linked with program in Figure 4.6.

Review questions

4.10.1. Page sharing can be used to (select all that apply):

a) Reduce the amount of memory used for multiple copies of the same program or library
b) Reduce the amount of memory used by different programs and libraries
c) Communicate between processes

```
pjd@pjd-fx:~$ pmap -p 18012
0000000000400000    4K r-x-- /tmp/a.out
0000000000600000    4K r---- /tmp/a.out
0000000000601000    4K rw--- /tmp/a.out
00007ffff7a0f000 1792K r-x-- /lib/x86_64-linux-gnu/libc-2.21.so
00007ffff7bcf000 2048K ----- /lib/x86_64-linux-gnu/libc-2.21.so
00007ffff7dcf000   16K r---- /lib/x86_64-linux-gnu/libc-2.21.so
00007ffff7dd3000    8K rw--- /lib/x86_64-linux-gnu/libc-2.21.so
00007ffff7dd5000   16K rw---   [ anon ]
00007ffff7dd9000  144K r-x-- /lib/x86_64-linux-gnu/ld-2.21.so
00007ffff7fcd000   12K rw---   [ anon ]
00007ffff7ff6000    8K rw---   [ anon ]
00007ffff7ff8000    8K r----   [ anon ]
00007ffff7ffa000    8K r-x--   [ anon ]
00007ffff7ffc000    4K r---- /lib/x86_64-linux-gnu/ld-2.21.so
00007ffff7ffd000    4K rw--- /lib/x86_64-linux-gnu/ld-2.21.so
00007ffff7ffe000    4K rw---   [ anon ]
00007ffffffde000  132K rw---   [ stack ]
ffffffffff600000    4K r-x--   [ anon ]
 total            4220K
```

Listing 4.8: Memory map for hello world program in Figure 4.6

4.10.2. The OS knows it can share a page when the same page in the same file is mapped in two different processes: *True / False*

More Memory Sharing: `fork()` and copy-on-write

In all the cases you've seen so far, page sharing has been used to share read-only pages—these are intrinsically safe to share, because processes are unable to modify the pages and thereby affect other processes. But, can writable pages be shared safely? The answer is yes, but it has to be done carefully.

First, some background on why this is important. The Unix operating system uses two system calls to create new processes and execute programs: `fork()` and `exec()`. `fork()` makes a copy of the current process[16], while `exec(file)` replaces the address space of the current process with the program defined by `file` and begins executing that program at its designated starting point.

UNIX uses this method because of an arbitrary choice someone made 40 years ago; there are many other ways to do it, each of them with their own problems. However this is how UNIX works, and we're stuck with it, so it's important to be able to do it quickly.

In early versions of Unix, `fork()` was implemented by literally copying all the writable sections (e.g., stack, data) of the parent process address space into the child process address space. After doing all this work, most (but not all) of the time, the first thing the child process would do is to call exec(), throwing away the entire contents of the address space that were just copied. It's bad enough when the shell does this, but even worse when a large program (e.g. Chrome) tries to execute a small program (e.g. /bin/ls) in a child process.

We've already seen how to share read-only data, but can we do anything about writable data? In particular, data which is writable, but isn't actually going to be written?

A quick inspection of several Firefox and Safari instances (using pmap on Linux and vmmap on OS X) indicates that a browser with two or three open tabs can easily have over 300MB of writable address space[17]. When fork is executed these writable pages can't just be given writable mappings in the child process, or changes made in one process would be visible in the other. In certain cases (i.e., the stack) this mutual over-writing of memory would almost certainly be disastrous.

[16]In fact the system call returns twice, once in the parent and once in the child

[17]This measurement was made in 2012; more recent versions use more memory.

However in practice, most of these writable pages *won't* be written to again. In fact, if the child process only executes a few lines of code and then calls `exec`, it may only modify a handful of pages before its virtual address space is destroyed and replaced with a new one.

Linux uses a technique called *copy-on-write* to eliminate the need to copy most of this memory. When a child process is created in the `fork` system call, its address space shares not only the read-only pages from the parent process, but the writable pages as well. To prevent the two processes from interfering with each other, these pages are mapped read-only, resulting in a page fault whenever they are accessed by either process, but flagged as copy-on-write in the kernel memory map structures. This results in a page fault when either process tries to write to one of these pages; the page fault handler then "breaks" the sharing for that page, by allocating a new page, copying the old one, and mapping a separate page read-write in each of the processes.

Copy-on-write is in fact a widely-used strategy in computer systems. It is effectively a "lazy" copy, doing only the minimal amount of work needed and reducing both the cost of copying and the total space consumed. Similar copy-on-write mechanisms can be seen in file systems, storage devices, and some programming language runtime systems.

Review questions

4.10.1. Copy-on-write allows writable data pages to be shared: *True / False*

4.10.2. Copy-on-write performs copying during the fork system call: *True / False*

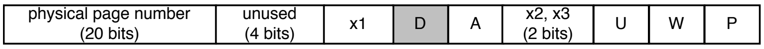


Figure 4.20: Page Table Entry with D (dirty) bit

Memory Over-Commitment and Paging

Page faults allow data to be dynamically fetched into memory when it is needed, in the same way that the CPU dynamically fetches data from memory into the cache. This allows the operating system to over-commit memory: the sum of all process address spaces can add up to more memory than is available, although the total amount of memory mapped at any point in time must fit into RAM. This means that when a page fault occurs and a page is allocated to a process, another page (from that or another process) may need to be evicted from memory.

Evicting a read-only page mapped from a file is simple: just forget the mapping and free the page; if a fault for that page occurs later, the page can be read back from disk. Occasionally pages are mapped read/write from a file, when a program explicitly requests it with `mmap`—in that case the OS can write any modified data back to the file and then evict the page; again it can be paged back from disk if needed again.

Types of Virtual Segments: There are two types of virtual segments: file-backed and anonymous. File-backed segments are what the name says; approximately 99.9% of them are read-only mappings of demand-paged executables. Anonymous mappings are called this because they don't correspond to a file; most of them contain writable program data or stacks.

Anonymous segments such as stack and heap are typically created in memory and do not need to be swapped; however if the system runs low on memory it may evict anonymous pages owned by idle processes, in order to give more memory to the currently-running ones. To do this the OS allocates a location in "swap space" on disk: typically a dedicated swap partition in Linux, and the `PAGEFILE.sys` and `/var/vm/swapfile` files in Windows and OSX respectively. The data must first be written out to that location, then the OS can store the page-to-location mapping and release the memory page.

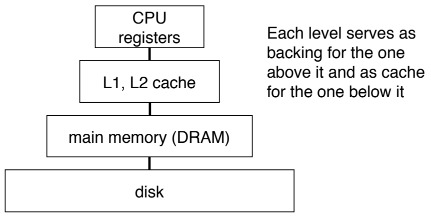

CPU registers
L1, L2 cache
main memory (DRAM)
disk
Each level serves as backing for the one above it and as cache for the one below it

Figure 4.21: Memory Hierarchy

Dirty and Clean Pages

How does the operating system determine whether a page has been modified and needs to be written to disk? It uses the D bit in the page table entry for this, as seen in Figure 4.20. When a page is mapped in the page table, the D bit in the PTE is set to zero; when the CPU writes to a page with D = 0, the MMU re-writes the page table entry with D = 1. When the OS decides to evict a page, the D bit tells it whether the page is "clean," i.e., it hasn't been modified, or whether it is "dirty" and has to be written back to disk.

When the OS is paging in from a file (e.g. executable code), it is straightforward to find the data to read in, as there is a direct mapping between a range of pages in a specific file and corresponding pages in the virtual memory space. This correspondence can easily be stored in the definition of that virtual address segment. When pages are saved to swap space this doesn't work, however, as the locations they are saved to are allocated dynamically and fairly arbitrarily.

This problem is solved by using the page table itself. After evicting a page, its page table entry is invalidated by setting P = 0; however, the other 31 bits of the entry are ignored by the MMU. These bits are used to store the location of the page in swap space, so it can be found later later at page fault time. Thus, the page table entry does dual duty: when the page is present it points to the physical page itself, and is interpreted by the MMU; otherwise, it points to a location in swap space, and is ignored by the MMU and used by the software page fault handler.

The Memory Hierarchy

Demand paging from files and from swap provides the mechanisms to create the traditional memory hierarchy, as shown in Figure 4.22.

To access address A:

- If it's not in the cache, then the old cache line is evicted, and A is loaded into the resulting empty cache line. This is done in hardware.
- If it's not in memory, then the old page is evicted, and the page containing A is loaded into the resulting empty page. This is done in software.

In general, this works because of *locality*: when a cache line is brought in from memory, a page is loaded into in memory from disk, etc., it tends to get accessed multiple times before eviction.

Decades ago this was used to run programs much bigger than physical memory—CPUs were slow and disks were almost as fast as they are today, so the relative overhead of paging infrequently-used data to disk was low. Today's CPUs are thousands of times faster, while disks are only a few times faster, and virtual memory doesn't seem like such a great idea anymore. However it still gets used, even on desktop and laptop systems, to "steal" memory from idle programs: if you leave a large program like Chrome or Microsoft Word idle for half an hour while you use another memory-hungry program, memory will be released from the idle process and given to the active one; if you switch back, the original program will run slowly for a while as it swaps these pages back in.

Review questions

4.10.1. When a value cannot be found in main memory, it must be fetched from: a) L2 or L1 cache b) Disk or other secondary storage

4.10.2. CPU caches and caches of disk data held in RAM both perform best when accesses are random: *True / False*

4.11 Page Replacement

If there's a limited amount of memory available, then every time a page is swapped in from disk, it will be necessary to remove, or evict, another page from memory. The choice of which page to evict is important: the best page to choose would be one that won't be needed anymore, while the worst page to evict would be one of the next to be used. (in that case, paging it back in would force another page to be evicted, and the work of paging it out and back in again would be wasted.) In fact, replacement of items in a cache is a general problem in computer systems; examples include:

- Cache line replacement in the hardware CPU cache
- Entry replacement in the TLB

- Buffer replacement in a file system buffer pool
- Page replacement in virtual memory

The page replacement problem can be stated in abstract form:

Given the following:

1. A disk holding d (virtual) pages, with virtual addresses $0, \ldots d-1$;
2. A memory M consisting of m (physical) pages, where each page is either empty or holds one of the d virtual pages, and
3. An access pattern $a_1, a_2, a_3, \cdots$ where each a_i is a virtual address in the range $(0, d-1)$:

a demand-paging strategy is an algorithm which for each access a_i does the following:

- If a_i is already in one of the m physical pages in M (i.e. a *hit*): do nothing
- Otherwise (a miss) it must:
- Select a physical page j in M (holding some virtual address M_j) and evict it, then
- Fetch virtual page a_i from disk into physical page j

In other words it only fetches page j *on demand*—i.e. in response to a request for it.

4.12 Page Replacement Strategies

In this class we consider the following page replacement strategies:

- FIFO: *first-in first-out.* The page evicted from memory is the first page to have been fetched into memory.
- LRU: *least-recently used.* Here, accesses to each page are tracked after it has been loaded into memory, and the least-recently-used page is evicted (unsurprisingly, given the name of the strategy).
- OPT: this is the optimal demand-paged strategy, which is simple but impossible to implement, since it requires knowledge of the future. It's examined because it provides a way of telling how well a real replacement strategy is performing—is it close to OPT, or is it far worse?

FIFO

This strategy is very simple to implement, as it only requires keeping track of the order in which pages were fetched into memory. Given 4 pages in physical memory, and the following access pattern:

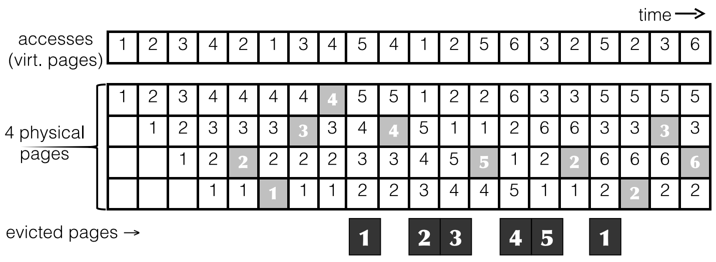


Figure 4.22: FIFO cleaning

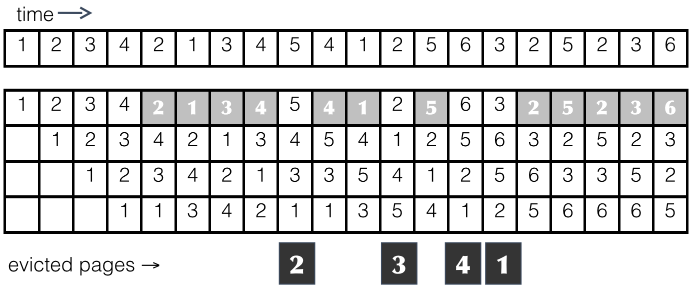


Figure 4.23: LRU cleaning

1 2 3 4 2 1 3 4 5 4 1 2 5 6 3 2 5 2 3 6

The contents of memory after each access is shown in Figure 4.22, with hits shown in light grey and pages evicted (when misses occur) shown in dark grey.

LRU

The idea behind LRU is that pages which have been accessed in the recent past are likely to be accessed in the near future, and pages which haven't, aren't. LRU replacement is shown in Figure 4.23.

To make the operation of the LRU algorithm more clear, on each hit, the accessed page is moved to the top of the column. (This is how LRU is typically implemented in software: elements are kept in a list, and on access, an element is removed and reinserted at the front of the list. The least-recently-used element may then be found by taking the tail of the list) Although this is a small example, a performance improvement is noted, with four misses compared to six for FIFO.

OPT

The optimal algorithm picks a page to evict by looking forward in time and finding the page which goes for the longest time without being accessed again. Except for seeing the future, OPT plays by the same rules as other demand-paging algorithms: in particular, it can't fetch a page until it is accessed. (That's why the OPT strategy still has misses.) OPT is shown in Figure 4.24, using the same access pattern as before. The first eviction decision is shown graphically: pages 4, 2, and 1 are accessed 1, 3, and 2 steps in the future, respectively, while page 3 isn't accessed for 6 steps and is thus chosen to be evicted.

FIFO with Second Chance (CLOCK)

LRU is simple and quite effective in many caching applications, and it's ideal that the operating system uses it to determine which pages to evict from memory. But there is one small problem in using it in a virtual memory system: in this case, a "miss" corresponds to a page fault and fetching a page from disk, while a "hit" is when the page is already mapped in memory and the access succeeds in hardware. This means that once a page is faulted into memory, any further use of that page is "invisible" to the operating system. If the OS doesn't know when a page was last used, it can't implement the Least-Recently-Used replacement strategy.

Despite this issue, it's still possible to do better than FIFO by using the A ("accessed") bit in the page table entry, which indicates whether the page has been accessed since the last time the bit was cleared[18]. In Figure 4.25 we see an algorithm called "FIFO with second chance," where the A bit is used to determine whether a page has been accessed while it was in

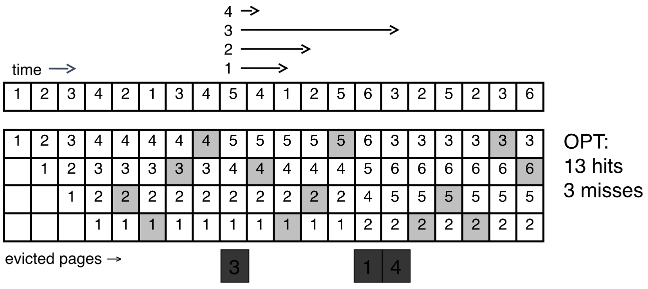


Figure 4.24: OPT (optimal) cleaning

[18]When the hardware reads a page table entry into the TLB it checks the A bit; if it is clear, then the hardware re-writes the entry with the A bit set.

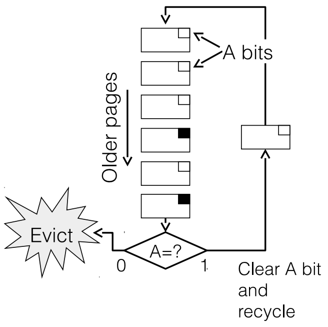


Figure 4.25: FIFO with second chance

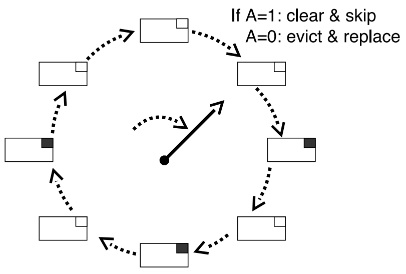


Figure 4.26: CLOCK Algorithm

the FIFO queue. If the A bit is 1, the replacement algorithm clears it and re-writes the page table entry, and the page is given “another chance,” i.e., it is cycled back to the head of the list. If the A bit is 0, then there have been no accesses to the page during its entire trip through the list, and so it is selected for replacement.

CLOCK

An alternate way of visualizing the FIFO with second chance algorithm is shown in Figure 4.26. Pages are arranged in a circle, with a “hand” advancing around the circle testing pages and determining whether to keep or evict them. This description is the origin of the widely-used name for this algorithm, CLOCK.

Review questions

4.12.1. Page replacement strategies are used to decide:

a) Which pages to load into memory from disk
b) Which pages to evict from memory

4.12.2. Which of these statements are true?

a) The OPT replacement strategy results in more misses (i.e. page faults) then LRU.
b) The OPT replacement strategy is easier to implement than LRU.
c) The CLOCK replacement strategy is easier to implement in a virtual memory system than LRU.

Answers to Review questions

4.3.1 *(translating 0x00C001C0)* 1, `0x00C00`. The top 20 bits (or 5 hex digits, at 4 bits each) form the page number. The bottom 12 bits (or *offset*) are `0x1C0`, and the top 10 bits (taken as a 10-bit binary number) are `0x008`.

4.3.2 *(top-level page table entry)* (3), (P=1, PPN=00003), as this is the entry at index 003 in the top-level page directory.

4.7.1 *(physical addresses read in page table walk)* (1), (00000080, 00002124). Remember that the address of the i^{th} 4-byte element in a table is $4 \cdot i$ bytes after the beginning, not i bytes.

4.8.1 False. Each page accessed in loading and executing an instruction can result in a page fault.

4.8.2 4 page faults - 2 for instruction fetch (in the case where the first bytes of an instruction are on one page, and the remainder is on the next page) and 2 for the memory access if it crosses a page boundary as well.

4.8.3 (b), the MMU. The page fault handler calculates virtual-to-physical mappings and installs them in the page table, but the MMU performs the actual translation when an address is used.

4.8.1 (2), specified in the executable file header. (or mostly so - the stack and heap are typically determined at runtime.) The CPU hardware puts very few restraints on the address space layout, and the command-line arguments are not used by the operating system but are instead passed directly to the program.

4.8.2 False. The CPU only switches address spaces when the OS explicitly loads the address of a new page table into the page table base register (CR3).

4.8.3 False. A single instruction can safely give rise to multiple page faults, one fault (or two, if page boundaries are straddled) for the instruction itself, and one or two for each memory address referenced by the instruction. (Note that this is different from a "double fault," which occurs if there is a page fault while executing the page fault handler.)

4.10.1 (1) and (3). Memory can't be shared between different programs and libraries, as shared pages will have the same contents in each address space.

4.10.2 True. As an example, different processes can share the memory pages used to map the code section of a particular program, so that no matter how many copies of the same program are running, only a single copy of the program code is needed in memory.

4.10.1 By copying pages before they are written to, COW allows sharing of writable pages without risk of interference or information leakage.

4.10.2 False. Shared mappings are created in `fork`, but actual copying is performed in the page fault handler.

4.10.1 (2), Disk / secondary storage. Data in L1/L2 cache is a subset of data in memory, which is a subset of data on disk.

4.10.2 False. Cache performance relies on non-randomness—i.e. that some values (hopefully the ones in cache) are used more than others.

4.12.1 (2). That's why it's called a page replacement strategy.

4.12.2 (1): False: no demand-paging strategy is more efficient than OPT. (2) False: OPT is impossible to implement. (3) True: CLOCK is easier to implement because it does not require precise knowledge of when pages are used.

Chapter 5

I/O, Drivers, and DMA

This chapter covers (a) the memory and I/O bus architecture of modern computers, (b) programmed I/O and direct-memory access, (c) disk drive components and how they influence performance, and (d) logical block addressing and the SATA and SCSI buses.

5.1 Introduction

Input/Output (I/O) devices are crucial to the operation of a computer. The data that a program processes — as well as the program binary itself — must be loaded into memory from some I/O device such as a disk, network, or keyboard. Similarly, without a way to output the results of a computation to the user or to storage, those results would be lost. One of the primary functions of the operating system is to manage these I/O devices. It should control access to them, as well as providing a consistent programming interface across a wide range of hardware devices with similar functionality but differing details. This chapter describes how I/O devices fit within the architecture of modern computer systems, and the role of programmed I/O, interrupts, direct memory access (DMA), and device drivers in interacting with them. In addition, you will examine one device, the hard disk drive and its corresponding controller, which is the source and destination of most I/O on typical systems.

5.2 PC architecture and buses

In Figure 5.1 you see the architecture of a typical Intel-architecture computer from a few years ago. Different parts of the system are connected by buses, or communication channels, operating at various speeds. The

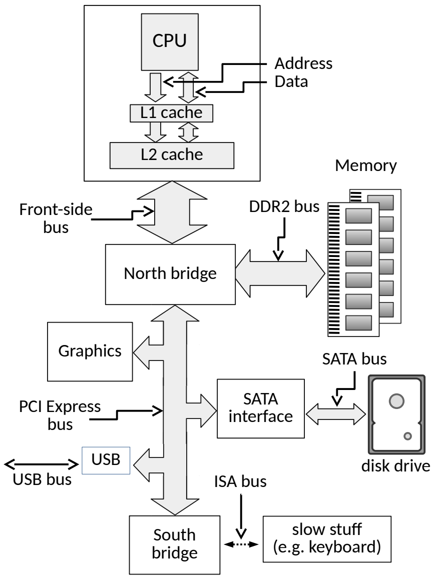


Figure 5.1: Standard Intel PC Architecture

Front-Side Bus carries all memory transactions which miss in L1 and L2 cache, and the North Bridge directs these transactions to memory (DDR2 bus) or I/O devices (PCIe bus) based on their address. The PCI Express (PCIe) is somewhat slower than the front-side bus, but can be extended farther; it connects all the I/O devices on the system. In some cases (like USB and SATA), a controller connected to the PCIe bus (although typically located on the motherboard itself) may interface to a yet slower external interface. Finally, the ISA bus is a vestige of the original IBM PC; for some reason, they've never moved some crucial system functions off of it, so it's still needed.[1]

Simple I/O bus and devices

The fictional computer system described in earlier chapters included a number of memory-mapped I/O devices, which are accessible at particular physical memory addresses. On early computers such as the Apple II and the original IBM PC this was done via a simple I/O bus as shown in Figure 5.2 and Figure 5.3. Address and data lines were extended across

[1]The primary difference between this figure and contemporary (2016) systems is that (a) the memory bus is DDR3 or DDR4, and (b) the north bridge is located on the CPU chip, with no external front-side bus.

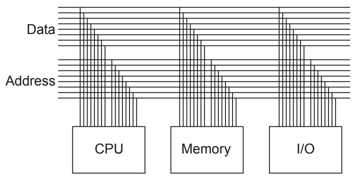


Figure 5.2: Simple memory/IO bus using shared address and data lines

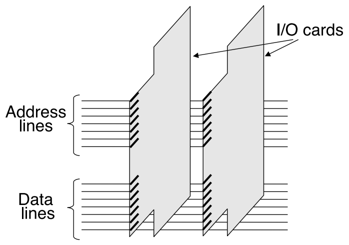


Figure 5.3: Simple memory/IO bus with extension cards

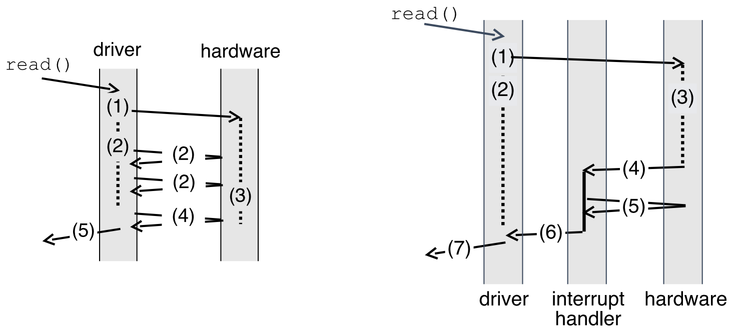
a series of connectors, allowing hardware on a card plugged into one of these slots to respond to read and write requests in much the same way as memory chips on the motherboard would. (This required each card to respond to a different address, no matter what combination of cards were plugged in, typically requiring the user to manually configure card addresses with DIP switches.)

The term "bus" was taken from electrical engineering; in high-power electric systems a *bus bar* is a copper bar used to distribute power to multiple pieces of equipment. A simple bus like this one distributes address and data signals in much the same way.

I/O vs. memory-mapped access: Certain CPUs, including Intel architecture, contain support for a secondary I/O bus, with a smaller address width and accessed via special instructions. (e.g. "IN 0x100" to read a byte from I/O location 0x100, which has nothing to do with reading a byte from memory location 0x100)

Memory-mapped I/O: like in our fictional computer, devices can be mapped in the physical memory space and accessed via standard load and store instructions. In either case, I/O devices will have access to an interrupt line, allowing interrupts to be raised for events like I/O completion.

Device selection: Depending on the system architecture, the device may be responsible for decoding the full address and determining when it has been selected, or a select signal may indicate when a particular slot on the bus is being accessed. Almost all computers today use a version of the PCI bus, which uses memory-mapped access, and at boot time, assigns each I/O device a physical address range to which it should respond.



(a) polled (b) interrupt-driven

Figure 5.4: Polled and interrupt-driven I/O

Polled vs. Interrupt-driven I/O

Polled (or “programmed”) I/O: As described in earlier chapters, the simplest way to control an I/O device is for the CPU to issue commands and then wait, polling a device status register until the operation is complete. In Figure 5.4(a) an application requests I/O via e.g. a `read` system call; the OS (step 1) then writes to the device command register to start an operation, after which (step 2) it begins to poll the status register to detect completion. Meanwhile (step 3) the device carries out the operation, after which (step 4) polling by the OS detects that it is complete, and finally (step 5) the original request (e.g. `read`) can return to the application.

Interrupt-driven I/O: The alternate is interrupt-driven I/O, as shown in Figure 5.4(b). After (step 1) issuing a request to the hardware, the OS (step 2) puts the calling process to sleep and switches to another process while (step 3) the hardware handles the request. When the I/O is complete, the device (step 4) raises an interrupt. The interrupt handler then finishes the request. In the illustrated example, the interrupt handler (step 5) reads data that has become available, and then (step 6) wakes the waiting process, which returns from the I/O call (step 7) and continues.

Latency and Programmed I/O

On our fictional computer the CPU is responsible for copying data between I/O devices and memory, using normal memory load and store instructions. Such an approach works well on computers such as the Apple II or the original IBM PC which run at a few MHz, where the address and data buses can be extended at full speed to external I/O cards. A modern CPU

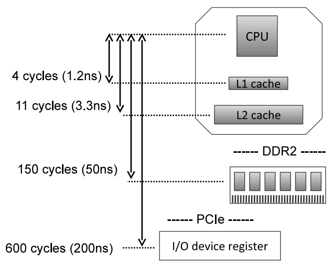


Figure 5.5: Latency between CPU and various levels of memory/IO hierarchy

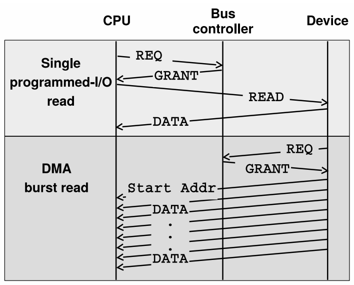


Figure 5.6: DMA access for high-speed data transfer

runs at over 3 GHz, however; during a single clock cycle light can only travel about 4 inches, and electrical signals even less. Figure 5.5 shows example latencies for a modern CPU (in this case an Intel i5, with L3 cache omitted) to read a data value from L1 and L2 cache, a random location in memory (sequential access is faster), and a register on a device on the PCIe bus. (e.g. the disk or ethernet controller) In such a system, reading data from a device in 4-byte words would result in a throughput of 5 words every microsecond, or 20MB/s — far slower than a modern network adapter or disk controller.

Review questions

5.2.1. Buses which extend farther from the CPU are usually:

a) Faster than those closer to the CPU
b) Slower than those closer to the CPU

5.2.2. Memory-mapped I/O is when the CPU reads from RAM: *True / False*

As CPU speeds have become faster and faster, RAM and I/O devices have only slowly increased in speed. The strategies for coping with the high relative latency of RAM and I/O are very different, however—caching works quite well with RAM, which stores data generated by the CPU, while I/O (at least the input side) involves reading new data; here latency is overcome by pipelining, instead.

The PCIe Bus and Direct Memory Access (DMA)

Almost all computers today use the PCIe bus. Transactions on the PCIe bus require a negotiation stage, when the CPU (or a device) requests access to bus resources, and then is able to perform a transaction after being granted access. In addition to basic read and write requests, the bus also supports Direct Memory Access (DMA), where I/O devices are able to read or write memory directly without CPU intervention. Figure 5.6 shows a single programmed-I/O read (top) compared to a DMA burst transfer (bottom). While the read request requires a round trip to read each and every 4-byte word, once the DMA transfer is started it is able to transfer data at a rate limited by the maximum bus speed. (For an 8 or 16-lane PCIe card this limit is many GB/s)

DMA Descriptors

A device typically requires multiple parameters to perform an operation and transfer the data to or from memory. In the case of a disk controller, for instance, these parameters would include the type of access (read or write), the disk locations to be accessed, and the memory address where data will be stored or retrieved from. Rather than writing each of these parameters individually to device registers, the parameters are typically combined in memory in what is called a *DMA descriptor*, such as the one shown in Figure 5.7. A single write is then used to tell the device the address of this descriptor, and the device can read the entire descriptor in a single DMA read burst. In addition to being more efficient than multiple programmed I/O writes, this approach also allows multiple requests to be queued for a device. (In the case of queued disk commands, the device may even process multiple such requests simultaneously.) When an I/O completes, the device notifies the CPU via an interrupt, and writes status information (such as success/failure) into a field in the DMA descriptor. (or sometimes in a device register, for simple devices which do not allow multiple outstanding requests.) The interrupt handler can then determine which operations have completed, free their DMA descriptors, and notify any waiting processes.

Cache-coherent I/O: The PCIe bus is *cache-consistent*; many earlier I/O buses weren't. Consider what would happen if the CPU wrote a value to location 1000 (say that's the command/status field of a DMA descriptor), then the device wrote a new value to that same location, and finally the CPU tried to read it back?

Device Driver Architecture

Figure 5.8 illustrates the I/O process for a typical device from user-space application request through the driver, hardware I/O operation, interrupt, and finally back to user space.

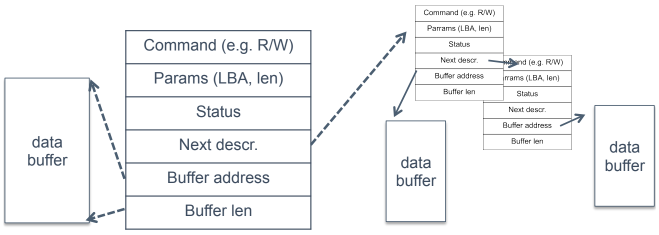


Figure 5.7: List of typical DMA descriptors

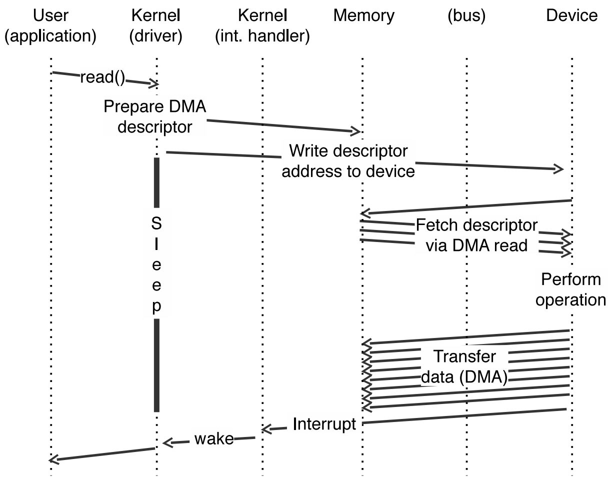


Figure 5.8: Driver Architecture

In more detail:

- The user process executes a `read` system call, which in turn invokes the driver `read` operation, found via the `read` method of the file operations structure.
- The driver fills in a DMA descriptor (in motherboard RAM), writes the physical address of the descriptor to a device register (generating a Memory Write operation across the PCIe bus), and then goes to sleep.
- The device issues a PCIe Memory Read Multiple command to read the DMA descriptor from RAM.
- The device does some sort of I/O. (e.g. read from a disk, or receive a network packet)
- A Memory Write and Invalidate operation is used to write the received data back across the PCIe bus to the motherboard RAM, and

to tell the CPU to invalidate any cached copies of those addresses.

- A hardware interrupt from the device causes the device driver interrupt handler to run.
- The interrupt handler wakes up the original process, which is currently in kernel space in the device driver read method, in a call to something like `interruptible_sleep_on`. After waking up, the read method copies the data to the user buffer and returns.

Review questions

5.2.1. High I/O latency can be compensated for by the use of CPU caches, so that almost all accesses complete at cache speeds instead of going over the bus: *True / False*

5.2.2. Direct Memory Access (DMA) refers to a class of CPU instructions which bypass the cache and access memory directly:
True / False

5.2.3. A device driver:

a) Is software which is part of the application
b) Is software which is part of the kernel
c) Is part of the hardware device

5.3 Hard Disk Drives

The most widely used storage technology today is the hard disk drive, which records data magnetically on one or more spinning platters, in concentric circular tracks. The performance of a hard drive is primarily determined by physical factors: the size and geometry of its components and the speeds at which they move:

Platter: the platter rotates at a constant speed, typically one of the following:

Speed	Rotations/sec	ms/rotation
5400 RPM	90	11
7200 RPM	120	8.3
10,000 RPM	167	6
15,000 RPM	250	4

Head and actuator arm: these take between 1 and 10 ms to move from one track to another on consumer disk drives, depending on the distance between tracks, and between 1 and 4 ms on high-end enterprise drives. (at the cost of higher power consumption and noise)

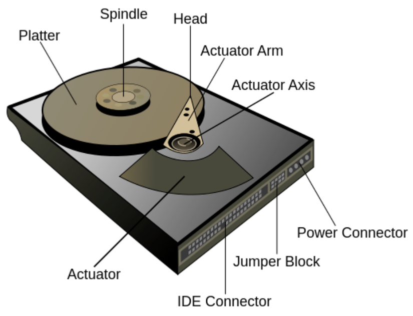


Figure 5.9: Hard Disk Drive (HDD) components

Bits and tracks: on modern drives each track is about 3 micro-inches (75nm) wide, and bits are about 1 micro-inch (25nm) long; with a bit of effort and knowing that the disk is 3.5 inches at its outer diameter you could calculate the maximum speed at which bits pass under the head.

Electronics and interface: the drive electronics are responsible for controlling the actuator and transferring data to and from the host. On a consumer drive this occurs over a SATA interface, which has a top speed of 150, 300, or 600MB/s for SATA 1, 2, or 3.

Hard Drive Performance

Data on a drive can be identified by the platter surface it is on, the track on that surface, and finally the position on that track. Reading data from a disk (or writing to it) requires the following steps:

- Switching the electronics to communicate with the appropriate head. (we'll ignore this, as it's fast)
- Moving the head assembly until the head is positioned over the target track. (*seek time*)
- Waiting for the platter to rotate until the first bit of the target data is passing under the head (*rotational latency*)
- Reading or writing until the last bit has passed under the head. (*transfer time*)

Geometric disk addressing

Unlike memory, data on a disk drive is read and written in fixed-sized units, or sectors, of either 512 or 4096 bytes. Thus small changes (e.g. a single byte) require what is known as a read/modify/write operation — a full sector is read from disk into memory, modified, and then written back to disk. These sectors are arranged in concentric tracks on each platter surface; a sector may thus be identified by its geometric coordinates:

- **Cylinder:** this is the track number; for historical reasons the group formed by the same track on all disk platters has been called a cylinder, as shown in the figure. Early disk drives could switch rapidly between accesses to tracks in the same cylinder; however this is no longer the case with modern drives.
- **Head:** this identifies one of the platter surfaces, as there is a separate head per surface and the drive electronics switches electrically between them in order to access the different surfaces.
- **Sector:** the sector within the track.

Performance examples

The overhead of seek and rotational delay has a major effect on disk performance. To give an example, consider randomly accessing a data

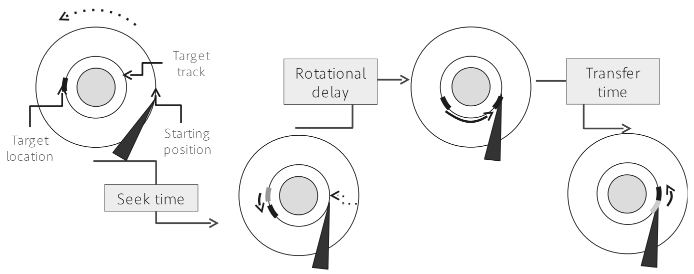


Figure 5.10: Hard disk latency

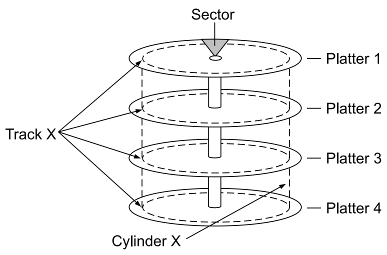


Figure 5.11: Why a track is also called a *cylinder* — the same track on each surface forms a “virtual” cylinder.

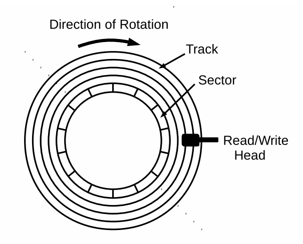


Figure 5.12: Disk access diagram

block on a 7200 RPM disk with the following parameters:

- Average seek time: 8 ms.
- Average rotational delay: 4 ms. (i.e., 1/2 rotation — after seeking to a track, the rotational delay for sectors on that track will be uniformly distributed between 0 and 1 rotation)
- Transfer rate: 200 MB/s. (outer tracks on disks available in 2017)

On average, reading a random 4KB block (i.e. one not immediately following the previous read) requires:

$$8 + 4 + 0.02 = 12ms$$

for an average throughput of 34 KB/s. (0.02 is 4KB / 200KB per ms) Random access to a 5 MB block, or over 1000 times more data, requires:

$$8 + 4 + 25 = 37ms$$

for an average throughput of 134MB/s. (25ms is obtained by dividing 5000KB by a rate of 200KB/ms)

In other words, although disks are random-access devices, random access is expensive. To achieve anywhere near full bandwidth on a modern disk drive you need to read or write data in large contiguous blocks; in our random access example, for instance, a 2 MB transfer would require 22 ms, or less than twice as long[2]as the smallest transfer.

[2]For system operations such as this where performance has a fixed and a variable component, you can think of the point where the two costs are equal as the “knee” in the curve, where you switch from the region where performance is dominated by the fixed cost to where it is dominated by the variable cost. To get high throughput you want to be firmly in the variable-cost region, where the fixed-cost effects are relatively minor.

Disk scheduling

A number of strategies are used to avoid the full penalties of seek and rotational delay in disks. One of these strategies is that of optimizing the order in which requests are performed—for instance reading sectors 10 and 11 on a single track, in that order, would require a seek, followed by a rotational delay until sector 10 was available, and then two sectors of transfer time. However reading 11 first would require the same seek and about the same rotational delay (waiting until sector 11 was under the head), followed by a full rotation to get from section 12 all the way back to sector 10.

Changing the order in which disk reads and writes are performed in order to minimize disk rotations is known as *disk scheduling*, and relies on the fact that multitasking operating systems frequently generate multiple disk requests in parallel, which do not have to be completed in strict order. Although a single process may wait for a read or write to complete before continuing, when multiple processes are running they can each issue requests and go to sleep, and then be woken in the order that requests complete.

Primary Disk Scheduling Algorithms

The primary algorithms used for disk scheduling are:

- **first-come first-served (FCFS):** in other words no scheduling, with requests handled in the order that they are received.
- **Shortest seek time first (SSTF):** this is the throughput-optimal strategy; however it is prone to starvation, as a stream of requests to nearby sections of the disk can prevent another request from being serviced for a long time.
- **SCAN:** this (and variants) are what is termed the *elevator algorithm* — pending requests are served from the inside to the outside of the disk, then from the outside back in, etc., much like an elevator goes from the first floor to the highest requested one before going back down again. It is nearly as efficient as SSTF, while avoiding starvation. (With SSTF one process can keep sending requests which will require less seek time than another waiting request, "starving" the waiting one.)

More sophisticated disk head scheduling algorithms exist, and could no doubt be found by a scan of the patent literature; however they are mostly of interest to hard drive designers.

Implementing Disk Scheduling

Disk scheduling can be implemented in two ways — in the operating system, or in the device itself. OS-level scheduling is performed by keeping a queue of requests which can be re-ordered before they are sent to the disk. On-disk scheduling requires the ability to send multiple commands to the disk before the first one completes, so that the disk is given a choice of which to complete first. This is supported as Command Queuing in SCSI, and in SATA as Native Command Queuing (NCQ).

Note that OS-level I/O scheduling is of limited use today for improving overall disk performance, as the OS has little or no visibility into the internal geometry of a drive. (OS scheduling is still used to merge adjacent requests into larger ones and to allocate performance fairly to different processes, however.)

On-Disk Cache

In addition to scheduling, the other strategy used to improve disk performance is caching, which takes two forms—*read caching* (also called track buffering) and *write buffering*. Disk drives typically have a small amount of RAM used for caching data[3]. Although this is very small in comparison the the amount of RAM typically dedicated to caching on the host, if used properly it can make a significant difference in performance.

At read time, after seeking to a track it is common practice for the disk to store the entire track in the on-disk cache, in case the host requests this data in the near future. Consider, for example, the case when the host requests sector 10 on a track, then almost (but not quite) immediately requests sector 11. Without the track buffer it would have missed the chance to read 11, and would have to wait an entire revolution for it to come back around; with the track buffer, small sequential requests such as this can be handled efficiently.

Write buffering is a different matter entirely, and refers to a feature where a disk drive may acknowledge a write request while the data is still in RAM, before it has been written to disk. This can risk loss of data, as there is a period of time during which the application thinks that data has been safely written, while it would in fact be lost if power failed.

Although in theory most or all of the performance benefit of write buffering could be achieved in a safer fashion via proper use of command queuing, this feature was not available (or poorly implemented) in consumer drives

[3]8-16MB two or three years ago; 128 MB is common today, probably in part because 128 MB chips are now cheaper than the old 16 MB ones.

until recently; as a result write buffering is enabled in SATA drives by default. Although write buffering can be disabled on a per-drive basis, modern file systems typically issue commands[4] to flush the cache when necessary to ensure file system data is not lost.

SATA and SCSI

Almost all disk drives today use one of two interfaces: SATA (or its precursor, IDE) or SCSI. The SATA and IDE interfaces are derived from an ancient disk controller for the PC, the ST-506, introduced in about 1980. This controller was similar to—but even cruder than—the disk interface in our fictional computer, with registers for the command to execute (read/write/other) and address (cylinder/head/sector), and a single register which the CPU read from or wrote to repeatedly to transfer data. What is called the ATA (AT bus-attached) or IDE (integrated drive electronics) disk was created by putting this controller on the drive itself, and using an extender cable to connect it back to the bus, so that the same software could still access the control registers. Over the years many extensions were made, including DMA support, logical block addressing, and a high-speed serial connection instead of a multi-wire cable; however the protocol is still based on the idea of the CPU writing to and reading from a set of remote, disk-resident registers.

Logical vs. CHS addressing: For CHS addressing to work the OS (and bootloader, e.g. BIOS) has to know the geometry of the drive, so it can tell e.g. whether the sector following (cyl=1,head=1,sector=51) is (1,1,52) or (2,1,0). For large computers sold with a small selection of vendor-approved disks this was not a problem, but it was a major hassle with PCs—you had to read a label on the disk and set BIOS options. Then drive manufacturers started using "fake" geometries because there weren't enough bits in the cylinder and sector fields, making drives that claimed to have 255 heads, giving the worst features of both logical and CHS addressing.

In contrast, SCSI was developed around 1980 as a high-level, device-independent protocol with the following features:

- Packet-based. The initiator (i.e. host) sends a command packet (e.g. READ or WRITE) over the bus to the target; DATA packets are then sent in the appropriate direction followed by a status indication. SCSI specifies these packets over the bus; how the CPU interacts with the disk controller to generate them is up to the maker of the disk controller. (often called an HBA, or host bus adapter)

[4]In SATA the FLUSH command or the FUA (force unit attention) flag. Don't ask me what "force unit attention" means - I have no idea.

- Logical block addressing. SCSI does not support C/H/S addressing — instead the disk sectors are numbered starting from 0, and the disk is responsible for translating this logical block address (LBA) into a location on a particular platter. In recent years logical addressing has been adopted by IDE and SATA, as well.

SCSI over everything

SCSI (like e.g. TCP/IP) is defined in a way that allows it to be carried across many different transport layers. Thus today it is found in:

- USB drives. The USB storage protocol transports SCSI command and data packets.
- CD and DVD drives. The first CD-ROM and CD-R drives were SCSI drives, and when IDE CDROM drives were introduced, rather than invent a new set of commands for CD-specific functions (e.g. eject) the drive makers defined a way to tunnel existing SCSI commands over IDE/ATA (and now SATA).
- Firewire, as used in some Apple systems.
- Fibre Channel, used in enterprise Storage Area Networks.
- iSCSI, which carries SCSI over TCP/IP, typically over Ethernet

and no doubt several other protocols as well. By using SCSI instead of defining another block protocol, the device makers gained SCSI features like the following:

- Standard commands (“Mode pages”) for discovering drive properties and parameters.
- Command queuing, allowing multiple requests to be processed by the drive at once. (also offered by SATA, but not earlier IDE drives)
- Tagged command queuing, which allows a host to place constraints on the re-ordering of outstanding requests.

Review questions

5.3.1. Since the platter spins while the head is seeking, rotational latency and seek time happen in parallel and the time until data can be accessed is the maximum of the two: *True / False*

5.3.2. Command queuing in SATA and SCSI will make which of the following workloads run faster:

a) Very large sequential reads and writes
b) A single process performing random reads and waiting for each read to complete before issuing the next one

c) Multiple processes performing random reads.

5.4 RAID and other re-mappings

In the previous section you learned about:

- Disk drives: how they work, and how that determines their performance
- SCSI and SATA buses, which carry block I/O commands between host controllers and disk drives
- The PCI bus, DMA, and device drivers which communicate between host controllers and the operating system

This section is about about *disk-like* devices, which behave like disks but aren't; this includes multi-disk arrays, solid-state drives (SSDs), and other block devices.

Early disk drives used cylinder/head/sector addressing, required the operating system to be aware of the exact parameters of each disk so that it could store and retrieve data from valid locations. The development of logical block addressing, first in SCSI, then in IDE and SATA drives, allowed drives to be interchangeable: with logical block addressing the operating system only needs to know how big a disk is, and can ignore its internal details.

This model is more powerful than that, however, as there is no need for the device on the other end of the SCSI (or SATA) bus to actually *be* a disk drive. (You can do this with C/H/S addressing, as well, but it requires creating a fake drive geometry, and then hoping that the operating system won't assume that it's the real geometry when it schedules I/O requests)

Instead the device on the other end of the wire can be an array of disk drives, a solid-state drive, or any other device which stores and retrieves blocks of data in response to write and read commands. Such disk-like devices are found in many of today's computer systems, both on the desktop and especially in enterprise and data center systems, and include:

- Partitions and logical volume management, for flexible division of disk space
- Disk arrays, especially RAID (redundant arrays of inexpensive disks), for performance and reliability
- Solid-state drives, which use flash memory instead of magnetic disks
- Storage-area networks (SANs)
- De-duplication, to compress multiple copies of the same data

Almost all of these systems look exactly like a disk to the operating system. Their function, however, is typically (at least in the case of disk arrays) an attempt to overcome one or more deficiencies of disk drives, which include:

- Performance: Disk transfer speed is determined by (a) how small bits can be made, and (b) how fast the disk can spin under the head. Rotational latency is determined by (b again) how fast the disk spins. Seek time is determined by (c) how fast the head assembly can move and settle to a final position. For enough money, you can make (b) and (c) about twice as fast as in a desktop drive, although you may need to make the tracks wider, resulting in a lower-capacity drive. To go any faster requires using more disks, or a different technology, like SSDs.
- Reliability: Although disks are surprisingly reliable, they fail from time to time. If your data is worth a lot (like the records from the Bank of Lost Funds), you will be willing to pay for a system which doesn't lose data, even if one (or more) of the disks fails.
- Size: The maximum disk size is determined by the available technology at any time—if they could build them bigger for an affordable price, they would. If you want to store more data, you need to either wait until they can build larger disks, or use more than one. Conversely, in some cases (like dual-booting) a single disk may be more than big enough, but you may need to split it into multiple logical parts.

In the rest of this section we will look at drive *re-mappings*, where a *logical volume* is created which is a different size or has different properties than the disk or disks it is built from. These mappings are not complex—in most cases a simple mathematical operation on a logical block address (LBA) within the logical volume will determine which disk or disks the operation will be directed to, and to what LBA on that disk. This translation may be done on an external device (a *RAID array*), within a host bus adaptor, transparently to the host (a *RAID adapter*), or within the operating system itself (*software RAID*), but the translations performed are the same in each case.

Partitioning

The first remapping strategy, partitioning, is familiar to most advanced computer users. A desktop or laptop computer typically has a single disk drive; however it is frequently useful to split that device into multiple logical devices via partitioning. An example is shown in Figure 5.1, where a single 250GB disk (named `sda`, SCSI disk a) has been split into three sections for a Linux installation. A small partition ('sda1') is used by the boot loader, followed by a swap partition used for virtual memory, and then the remainder ('sda3') is used for the root file system. Another common use for partitioning is for dual-booting a machine, where e.g. Windows might be installed into one partition and Linux or Apple OS X installed in another. Note that unlike some of the other remappings we will examine, partitioning is almost always handled in the operating system itself, rather than in an external device.

```
   Device Boot     Start        End      Blocks  Id  System
/dev/sda1   *         63     208844      104391  83  Linux
/dev/sda2         208845    4401809    2096482+  82  Linux swap
/dev/sda3        4401810  488392064  241995127+  83  Linux
```

Listing 5.1: Example Linux partition map

There are two parts to disk partitioning: (a) a method for recording partition information in a *partition table* to be read by the operating system, and (b) translating in-partition logical block addresses (LBAs) into *absolute* LBAs (i.e. counting from the beginning of the entire disk) at runtime.

The first step is done via a *partition table* on the disk, which gives the starting logical block address (LBA), length, and type of each partition. On boot the operating system reads this table, and then creates virtual block devices (each with an LBA range starting at 0) for each partition. There are two partition table formats in wide use today — Master Boot Record (MBR) boot tables based on the original IBM PC disk format, and GUID Partition Table (GPT) tables used in new systems; for more detail see the following Wikipedia entries: `http://en.wikipedia.org/wiki/Master_Boot_Record`, `http://en.wikipedia.org/wiki/GUID_Partition_Table`

Address translation: Figure 5.13 shows a logical view of the translation between logical block addresses within a partition and physical addresses on the actual device.

Given a partition with start address S and length L and a block address A within that partition, the actual on-disk address A0 can be determined as follows:

```
if A > L:
   error
else:
   A0 = A + S
```

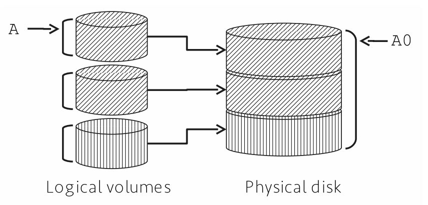


Figure 5.13: Partition layout and formula

Review questions

5.4.1. Which one of the following statements best describes what a disk partition is?

a) A set of files on a disk reserved for a specific purpose
b) A portion of the disk address space (LBA or logical block address space) which is treated as a separate virtual device
c) A type of disk drive

Concatenation

Concatenation means joining two things (like strings) end-to-end; a concatenated volume is the opposite of a partitioned disk, joining the LBA spaces of each disk, one after the other, into a single logical volume which is the sum of multiple physical disks.

Why would you do this? After all, you can just create separate file systems on multiple disks and use the mount command to join them into a single file system hierarchy, as shown in Figure 5.14.

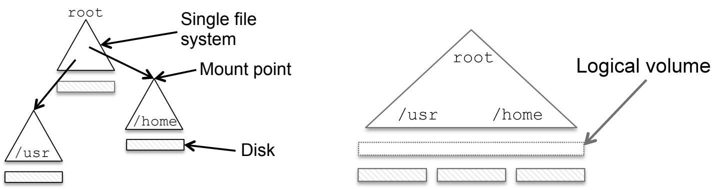


Figure 5.14: Multiple mounted file systems vs. single concatenated volume

This has disadvantages, though. What if you have 3 100GB disks, but 200GB of home directories? Now you're stuck with home directories that look like /home/disk1/joe and /home/disk2/jane, and no matter how you assign accounts, one of the disks is likely to fill up while there is still a lot of free space on the other one.

If you can paste all three disks together and create a single large volume, however, with a single file system on top, then you have a single large, flexible volume, and you don't need to guess how much space to allocate for different directories. (the most modern file systems — ZFS and Btrfs — will handle this for you, but widely-used file systems like NTFS and ext3 do not.)

In Figure 5.15 we see concatenation with three disks, D_1, D_2, D_3, of size S_1, S_2, S_3. The address A in the concatenated volume is translated to a physical disk D_0 and an address on that disk A_0, and (as for partitioning) the translation is very simple:

```
if A < S1 then
   D0 = D1
   A0 = A
else if A < S2 then
   D0 = D2
   A0 = A-S1
else if A < S3 then
   D0 = D3
   A0 = A-S1-S2
else
   error
```

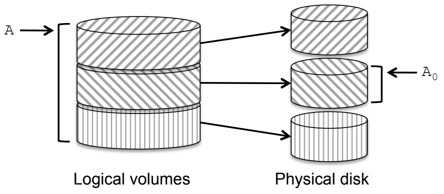


Figure 5.15: Concatenation layout and formula

Concatenation may be implemented in the OS (via the Logical Volume Manager in Linux, or as a type of "software RAID" in Windows) or in an external storage device. With the right tools for modifying the file system, it can even be used to add another disk to an existing file system.

Striping — faster concatenation

Although the size of a concatenated volume is the sum of the individual disk sizes, the performance is typically not. For instance, if you create a single large file, it will probably be placed on contiguous blocks on one of the disks, limiting read and write throughput to that of a single disk. If you've paid for more than one disk, it would be nice to actually get more than one disk's performance, if you can.

Isn't that RAID0? The term "RAID" was coined in a 1988 paper by Paterson, Gibson, and Katz, titled "A case for redundant arrays of inexpensive disks (RAID)", where they defined RAID levels 0 through 5—it turns out RAID0 and RAID1 were what everyone had been calling "striping" and "mirroring" for years, but no one had a name for the newer parity-based systems. RAID2 and 3 are weird and obsolete; no one talks about them.

If the file was instead split into small chunks, and each chunk placed on a different disk than the chunk before it, it would be possible to read and write to all disks in parallel. This is called *striping*, as the data is split into stripes which are spread across the set of drives.

In Figure 5.16 we see individual *strips*, or chunks of data, layed out in horizontal rows (called *stripes*) across three disks. In the figure, when writing strips 0 through 5, strips 0, 1, and 2 would be written first at the same time to the three different disks, followed by writes to strips 3, 4, and 5. Thus, writing six strips would take the same amount of time it takes to write two strips to a single disk.

How big is a strip? It depends, as this value is typically configurable—

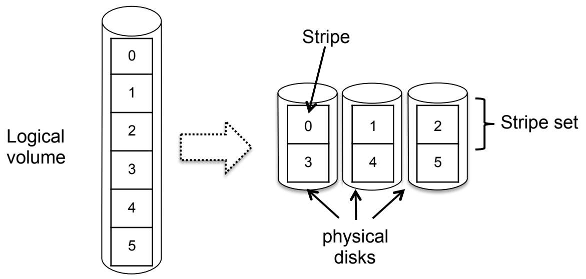


Figure 5.16: Striping across three disks

the RAID algorithms work with any strip size, although for convenience everyone uses a power of 2. If it's too small, the large number of I/Os may result in overhead for the host (software RAID) or for the RAID adapter; if it's too large, then large I/Os will read or write from individual disks one at a time, rather than in parallel. Typical values are 16 KB to 512 KB. (the last one is kind of large, but it's the default built into the `mdadm` utility for creating software RAID volumes on Linux. And the `mdadm` man page calls them "chunks" instead of "strips", which seems like a much more reasonable name.)

Striping data across multiple drives requires translating an address within the striped volume to an address on one of the physical disks making up the volume, using these steps:

1. Find the stripe set that the address is located in - this will give the stripe number within an individual disk.
2. Calculate the stripe number within that stripe set, which tells you the physical disk the stripe is located on.
3. Calculate the address offset within the stripe.

Note that—unlike concatenation—each disk must be of the same size for striping to work. (Well, if any disks are bigger than the smallest one, that extra space will be wasted.)

Given 3 disks d1, d2, d3 of the same size, with a strip size of N sectors, an address A in the striped volume is translated to a physical disk D_0 and an address on that disk A_0 as follows, assuming integer arithmetic:

Review questions

5.4.1. Which one of the following statements best describes the total storage capacity of a **striped** volume of equal-sized disks?

```
S = A / N
- strip # in volume
O = A % N
- offset in strip
case S % 3:
- disk is n1 mod 3
  0:  $D_0$= d1
  1:  $D_0$= d2
  2:  $D_0$= d3
$S_d$ = S / 3
- stripe # in disk
$A_0$ = $S_d$*N + O
```

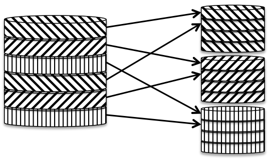


Figure 5.17: Striping layout and formula

a) the same as one of the disks in the volume
b) the sum of the capacity of the disks in the volume

5.4.2. The disks within a striped volume (or at least the portion used of each disk) must be the same size: *True / False*

Mirroring

Disks fail, and if you don't have a copy of the data on that disk, it's lost. A lot of effort has been spent on creating multi-disk systems which are more reliable than single-disk ones, by adding *redundancy*—i.e. additional copies of data so that even if one disk fails completely there is still a copy of each piece of your data stored safely somewhere. (Note that striping is actually a step in the wrong direction - if *any one* of the disks in a striped volume fail, which is more likely than failure of a single disk, then you will almost certainly lose all the data in that volume.)

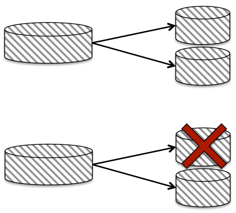


Figure 5.18: Failure of one disk in mirrored volume.

The simplest redundant configuration is *mirroring*, where two identical ("mirror image") copies of the entire volume are kept on two identical disks. In Figure 5.18 we see a mirrored volume comprising two physical disks; writes are sent to both disks, and reads may be sent to either one. If one disk fails, reads (and writes) will go to the remaining disk, and data is not lost. After the failed disk is replaced, the mirrored volume must be rebuilt (sometimes termed "re-silvering") by copying its contents from the other drive. If you wait too long to replace the failed drive, you risk having the second drive crash, losing your data.

Address translation in a mirrored volume is trivial: address A in the logical volume corresponds to the same address A on each of the physical disks. As with striping, both disks must be of the same size. (or any extra sectors in the larger drive must be ignored.)

Mirroring and Consistency

A mirrored volume can be temporarily inconsistent during writing. Consider the following case, illustrated in Figure 5.19:

1. a block in the logical volume contains the value X, and a write is issued changing it to Y, and
2. Y is successfully written to one disk but not the other, and then
3. the power fails

Now, when the system comes back up (step 4 in the figure) the value of this block will depend on which disk the request is sent to, and may change if a disk fails.

High-end storage systems typically solve this problem by storing a temporary copy of written data to non-volatile memory (NVRAM), either battery-backed RAM or flash. If power fails, on startup the system can check that all recent writes completed to each disk. Without hardware support, the OS can check on startup to see if it was cleanly shut down, and if not it may need to check both sides of the mirror and ensure they are consistent. (a lengthy process with modern disks)

When recovering an inconsistent mirrored volume, the value from either disk may be used. Why is this OK? (it helps to remember that from the point of view of the file system or application, a write to a mirrored volume does not complete until both sides have been successfully written to.)

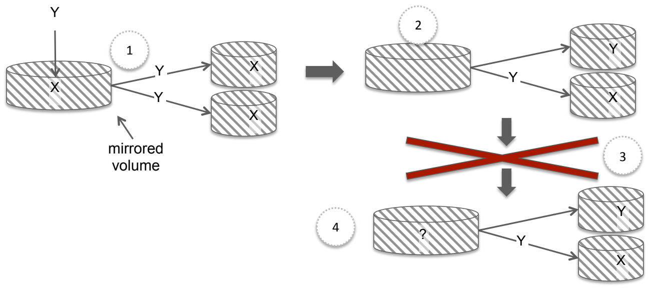
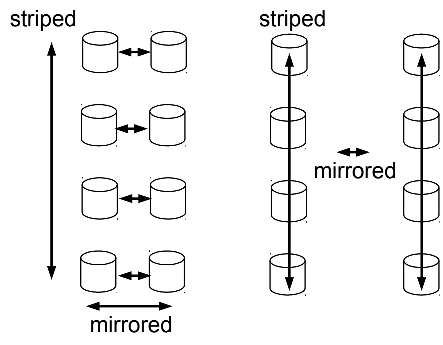


Figure 5.19: Failure during mirror write causing inconsistency



RAID 1+0
(stripe of mirrors)

RAID 0+1
(mirror of stripes)

Striping + Mirroring (RAID 0+1, RAID 1+0)

Mirroring and striping can also be used to construct a logical volume out of other logical volumes, so you can create a mirrored volume consisting of two striped volumes, or a striped volume consisting of two mirrored volumes. In either case, a volume holding N drives worth of data will take 2N drives to hold (in this figure, that works out to eight drives) and will give N times the performance of a single disk.

Since striping is also known as RAID 0 and mirroring as RAID 1, these configurations are called RAID 0+1 and RAID 1+0, respectively. RAID 0+1 is less reliable, as if one disk fails in each of the two striped volumes the whole volume will fail. Interestingly enough, the disks contain exactly the same data in both cases; however, in the RAID 0+1 case the controller doesn't try as hard to recover it.

Review questions

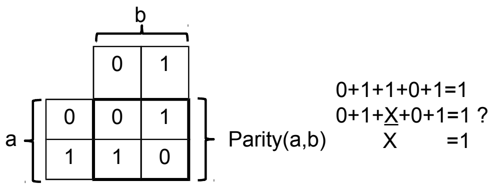
5.4.1. Which one of the following statements best describes the storage capacity of a mirrored volume?

a) It is the same as that of one of the disks making up the volume
b) It is equal to the sum of the capacities of the disks making it up
c) It is equal to the sum of the capacities of all disks, minus the capacity of the parity drive

RAID 4

Although mirroring and RAID 1+0 are good for constructing highly reliable storage systems, sometimes you don't want reliability bad enough to be willing to devote half of your disk space to redundant copies of data. This is where RAID 4 (and the related RAID 5) come in.

For the 8-disk RAID 1+0 volume described previously to fail, somewhere between 2 and 5 disks would have to fail (3.66 on average). If you plan on replacing disks as soon as they fail, this may be more reliability than you need or are willing to pay for. RAID 4 provides a high degree of reliability with much less overhead than mirroring or RAID 1+0.



RAID 4 takes N drives and adds a single parity drive, creating an array that can tolerate the failure of any single disk without loss of data. It does this by using the parity function (also known as exclusive-OR, or addition modulo 2), which has the truth table seen in the figure to the right. As you can see in the equation, given the parity calculated over a set of bits, if one bit is lost, it can be re-created given the other bits and the parity. In the case of a disk drive, instead of computing parity over N bits, you compute it over N disk blocks, as shown here where the parity of two blocks is computed:

```
  001010011101010010001 ... 001101010101 +
  011010100111010100100 ... 011000101010

= 010000111010000110101 ... 010101111111
```

RAID 4 - Organization: RAID 4 is organized almost exactly like a striped (RAID 0) volume, except for the parity drive. We can see this in Figure 5.20 — each data block is located in the same place as in the striped volume, and then the corresponding parity block is located on a separate disk.

Writing to a RAID 4 Volume: How you write to a RAID 4 volume depends on whether it is a small or large write. For large writes you can over-write a complete stripe set at a time, letting you calculate the parity before you write. Small writes are less efficient: you have to read back some amount of data in order to re-calculate the parity. There are two

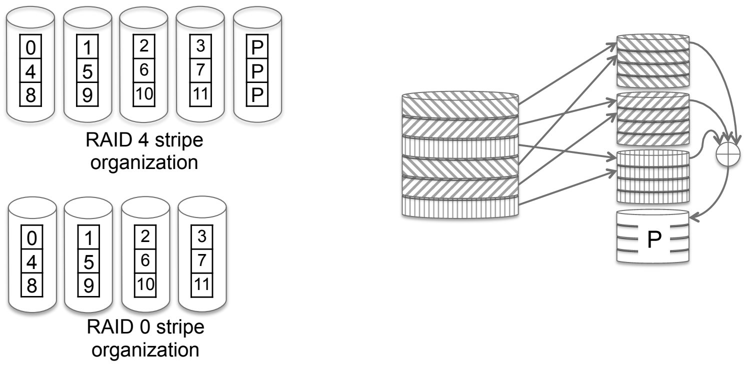
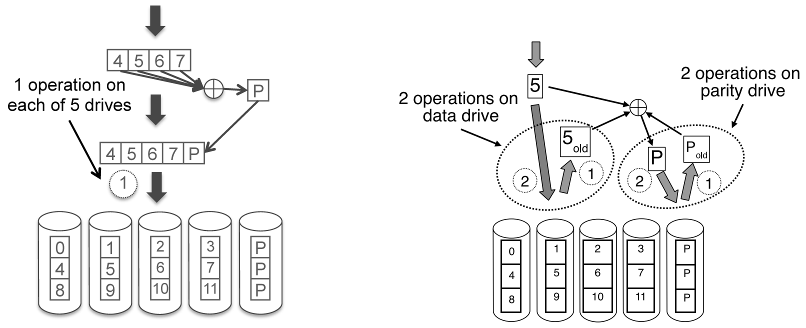


Figure 5.20: RAID 4 organization



Large write:

- Calculate parity (no I/O needed)
- (1) Write stripe set to disk

Small write:

- (1) read old data, parity
- Calculate new parity (no I/O)
- (2) write new data, parity

Figure 5.21: Large and small writes to RAID 4

options: you can either read the entire stripe set and calculate its parity after modifying it, or you can read the old data and parity, subtract the old data, and add in the new data, which is more efficient for larger RAID volumes (i.e. with more than 4 drives).

In Figure 5.21 you can see that a small write can take twice as long and require four times as many operations as the corresponding write to a striped volume, where no parity recalculation is needed.

A question for the reader: why does a small write to RAID 4 take twice as long, rather than four times as long, as a single disk write?

Reading from a RAID 4 Volume: There are two cases when reading from a RAID 4 volume: normal mode and *degraded mode*. In normal mode the data is available on the disk(s) it was written to, which is the case when no disks have failed, and for data on the remaining disks after one has failed. In *degraded* mode the data being read was written to the failed drive, and must be reconstructed from the remaining data and parity in the stripe set. (The actual reconstruction is quite simple, as the missing data stripe is just the exclusive OR of all the remaining data and parity in the stripe set.)

To write in degraded mode, parity is calculated and stripes are written to all but the failed disk. When the disk is replaced, its contents will be reconstructed from the other drives.

Review questions

5.4.1. A RAID 4 volume with five data drives and one parity drive can tolerate two disk failures without data loss: *True / False*

5.4.2. A RAID 4 volume with five data drives and one parity drive holds more data than three mirrored disk pairs (six disks total) assuming the disks are the same size in the two cases: *True / False*

5.4.3. After a disk fails on a RAID 4 volume, which statement is more correct?

a) It should be replaced quickly
b) It doesn't need to be replaced immediately, as the RAID controller will prevent data loss if another disk fails

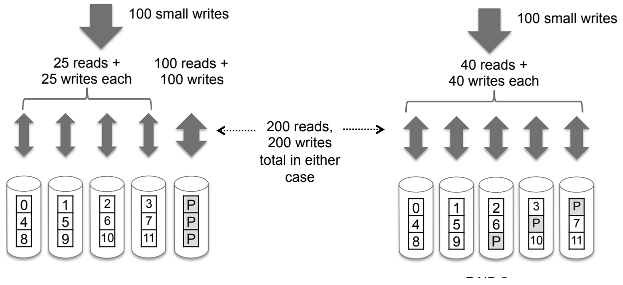
5.4.4. Which one of the following statements best describes the efficiency of small writes on RAID 4?

a) They are more efficient than large writes
b) They are less efficient than large writes

RAID 5

Small writes to RAID 4 require four operations: one read each for the old data and parity, and one write for each of the new data and parity. Two of these four operations go to the parity drive, no matter what LBA is being written, creating a bottleneck. If one drive can handle 200 random operations per second, the entire array will be limited to a total throughput of 100 random small writes per second, no matter how many disks are in the array.

By distributing the parity across drives in RAID 5, the parity bottleneck is eliminated. It still takes four operations to perform a single small write, but those operations are distributed evenly across all the drives. (Because of the distribution algorithm, it's technically possible for all the writes to go to the same drive; however it's highly unlikely.) In the five-drive case shown here, if a disk can complete 200 operations a second, the RAID 4 array would be limited to 100 small writes per second, while the RAID 5 array could perform 250. (5 disks = 1000 requests/second, and 4 requests per small write)



RAID 6 - more reliability

RAID level 1 (including 1+0 and 0+1), and levels 4 and 5 are designed to protect against the total failure of any single disk, assuming that the remaining disks operate perfectly. However, there is another failure mode known as a *latent sector error*, in which the disk continues to operate but one or more sectors are corrupted and cannot be read back. As disks become larger these errors become more problematic: for instance, one vendor specifies their current desktop drives to have no more than 1 unrecoverable read error per 10^{14} bits of data read, or 12.5 TB. In other words, there might be in the worst case a 1 in 4 chance of an unrecoverable read error while reading the entire contents of a 3TB disk. (Luckily, actual error rates are typically much lower, but not low enough.)

If a disk in a RAID 5 array fails and is replaced, the "rebuild" process requires reading the entire contents of each remaining disk in order to reconstruct the contents of the failed disk. If any block in the remaining drives is unreadable, data will be lost. (Worse yet, some RAID adapters and software will abandon the whole rebuild, causing the entire volume to be lost.)

RAID 6 refers to a number of RAID mechanisms which add additional redundancy, using a second parity drive with a more complex error-correcting code[5]. If a read failure occurs during a RAID rebuild, this additional protection may be used to recover the contents of the lost block, preventing data loss. Details of RAID 6 implementation will not be covered in this class, due to the complexity of the codes used.

Review questions

5.4.1. RAID 5 is less likely to lose data from disk failure than RAID 4: *True / False*

5.4.2. RAID 5 is faster for very large writes than RAID 4: *True / False*

5.4.3. RAID 5 is faster for small writes than RAID 4: *True / False*

5.4.4. Which one of the following statements best describes why RAID 6 has become important recently?

a) Because total failure is more common in modern disks
b) Because modern disks are bigger

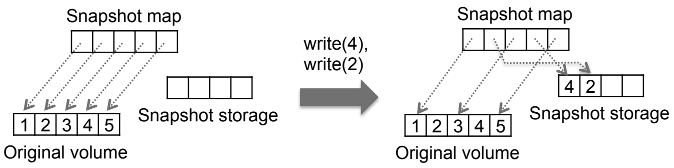
Logical Volume Management

If you have managed a Linux system (especially Fedora or Red Hat) you may have used the Logical Volume Manager (LVM), which allows disks on the system to be flexibly combined and split into different volumes; similar functionality is available on other operating systems, as well as on high-end storage arrays.

The volume types which can be created under LVM are those which have been described in this section: partitioned, concatenated, and the various RAID levels. In addition, however, logical volume managers typically offer functions to migrate storage contents and to create snapshots of a volume.

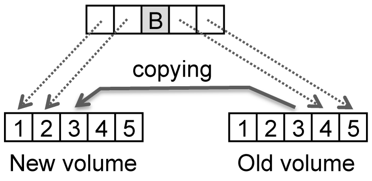
Volume snapshots rely on a copy-on-write mechanism almost identical to that used in virtual memory:

[5]Commonly a Reed-Solomon code; see Wikipedia if you want to find out what that is.



A snapshot is a "lazy copy" of a volume—it preserves the contents without immediately consuming any additional disk space, instead consuming space as the volume is written to. (It's also much faster than copying all the data) Why would you want to make a snapshot? Maybe you want to save the state of your machine before you make major changes, like installing new software and drivers, or upgrading the OS. If things don't work out, you can revert back to the snapshot and try again.

Snapshots are also frequently used for backing up a computer, because it takes so long to copy all the data from a modern disk. If you merely copied all the files off of the disk, the backed-up version of one file might be hours older than another file; this can be avoided by backing up a snapshot instead of the volume itself.



Live migration is a sort of magical operation, allowing you to switch from one disk drive to another while the machine continues to run. It works by using a map to direct individual requests to either the old volume or the new volume, with the dividing line moving as data is copied from one to the other. What happens if you try to write to the small section being copied in the middle? The write gets stalled until the copy is done, and then is directed to the new location.

Solid State Drives

Solid-state drives (SSDs) store data on semiconductor-based flash memory instead of magnetic disk; however by using the same block-based interface (e.g. SATA) to connect to the host they are able to directly replace disk drives.

SSDs rely on flash memory, which stores data electrically: a high programming voltage is used to inject a charge onto a circuit element (a *floating gate*—ask your EE friends if you want an explanation) that is isolated by insulating layers, and the presence or absence of such a stored charge can be detected in order to read the contents of the cell. Flash memory has several advantages over magnetic disk, including:

- Random access performance: since flash memory is addressed electrically, instead of mechanically, random access can be very fast.
- Throughput: by using many flash chips in parallel, a consumer SSD (in 2018) can read speeds of 1-2 GB/s, while the fastest disks are limited to a bit more than 200MB/s.

Flash is organized in pages of 4KB to 16KB, which must be read or written as a unit. These pages may be written only once before they are erased in blocks of 128 to 256 pages, making it impossible to directly modify a single page. Instead, the same copy-on-write algorithm used in LVM snapshots is used internally in an SSD: a new write is written to a page in one of a small number of spare blocks, and a map is updated to point to the new location; the old page is now invalid and is not needed. When not enough spare blocks are left, a garbage collection process finds a block with many invalid pages, copies any remaining valid pages to another spare block, and erases the block.

When data is written sequentially, this process will be efficient, as the garbage collector will almost always find an entirely invalid block which can be erased without any copying. For very random workloads, especially on cheap drives with few spare blocks and less sophisticated garbage collection, this process can involve huge amounts of copying (called write amplification) and run very slowly.

SSD Wear-out: Flash can only be written and erased a certain number of times before it begins to degrade and will not hold data reliably: most flash today is rated for 3000 write/erase operations before it becomes unreliable. The internal SSD algorithms distribute writes evenly to all blocks in the device, so in theory you can safely write 3000 times the capacity of a current SSD, or the entire drive capacity every day for 8 years. (Note that 3000 refers to *internal* writes; random writes with high write amplification will wear out an SSD more than the same volume of sequential writes.)

For a laptop or desktop this would be an impossibly high workload, especially since they are typically used only half the hours in a day or less. For some server applications, however, this is a valid concern. Special-purpose SSDs are available (using what is called Single-Level Cell, or SLC, flash) which are much more expensive but are rated for as many as 100,000 write/erase cycles. (This capacity is the equivalent of overwriting an entire drive every 30 minutes for 5 years. For a 128GB drive, this would require continuously writing at over 70MB/s, 24 hours a day.)

New Disk Technologies

The capacity of a disk drive is determined by how many bits there are on a track (i.e. how *short* the bits are), how many tracks fit on each side of a platter (how *narrow* the bits are), and how many platters and sssociated heads fit into a drive enclosure. Since sometime around the late 90s most of the increase in drive density has come from making the tracks narrower; however this has hit a stumbling block recently. The narrower you make the write head, the weaker its magnetic field, until eventually it becomes too weak to magnetize bits on the platter. You can fix this for a while by making the platter easier to magnetize (lower *coercivity*), but if you go too far in that direction, the bits will flip spontaneously due to thermal noise. (There's a cure for that—make the bits bigger—but it obviously won't help.)

In the last few years disks have come perilously close to this limit. Much of the capacity growth in the last couple of years (2018) and most in coming years is expected to come from the following technologies:

- **Helium:** Filling the drive with helium[6] reduces the air turbulence around the heads and platters, allowing them to be thinner so you can cram more of them into a disk. (The highest capacity air-filled drives typically had 4 platters and 8 heads; the largest helium-filled drives today have 9 platters.)
- **Shingled magnetic recording (SMR):** Narrow tracks can be written with a wide (and thus high magnetic field) head by overlapping the wide tracks, like rows of shingles[7], and read back by a narrower read head. Unfortunately, overwriting a sector on an SMR disk will damage the neighboring sector, requiring a translation layer (much like a flash translation layer) in order to be used by a normal file system.
- **Heat-assisted Magnetic Recording (HAMR):** If you heat a magnetic material it becomes easier to magnetize. HAMR relies on narrow, weak write heads that shouldn't be able to write to the platter, and heats the surface with a laser just before writing to it.

Although the impending death of hard disk drives has been predicted many times—Google "bubble memory" for an example–technological breakthroughs have come through each time to keep them in the position

[6]Which is harder than it sounds, since helium will leak through cast aluminum, which is the preferred material for HDD enclosures.

[7]Really more like clapboards, but "clapboarded" just doesn't have the same ring to it.

of the most cost-effective bulk storage medium available. It remains to be seen whether high-density SSDs based on low-performance NAND flash are able to catch up to disk in cost per terabyte, or whether some technological advance will keep disk ahead for yet another decade.

Storage-area Networks

In enterprise environments it is typical to separate storage systems from the servers that use the storage. This allows tasks such as backup to be centralized, as well as simplifying the task of replacing or servicing hardware. (In fact, in a virtualized environment (covered in a later chapter) external storage allows running servers to be moved from one piece of hardware to another without interruption.)

Storage-Area Networks, or SANs, typically use the SCSI protocol and a transport which can be routed or switched as a network. The most common SAN technologies are Fibre Channel and iSCSI:

- Fibre Channel is a bizarre networking protocol used only in SANs; for historic reasons it is typically used with optical fiber cabling, which is expensive and unreliable for short connections.
- iSCSI is an encapsulation of SCSI within TCP/IP; it uses traditional ethernet cabling, switching, and IP routing, although an iSCSI deployment may use a separate network for storage.

"Disks" on a SAN are identified by a transport address (either an IP address or DNS name, for iSCSI, or a 64-bit World Wide Name (WWN) for Fibre Channel) plus a logical unit number (LUN), which identifies a specific volume on a target. In other words, an individual block of data on a SAN can be identified by address + LUN + LBA.

One of the key administrative features in a SAN is LUN masking, which determines which resources (LUNs) on the network may be seen by which hosts. This lets each server see only the LUNs which have been assigned to it, so that a misconfigured host cannot access or corrupt storage which it is not supposed to have access to. In addition to source-based access control, iSCSI also offers several authentication protocols, to prevent access to disk volumes from unauthorized hosts or applications.

De-duplication

Large enterprise storage systems typically store large amounts of similar data. As an example, your CCIS account stores your home directory on a central server; if you log onto a college Windows or Linux machine almost all the files you create and edit will be located on this server.

In a corporate environment this approach is frequently used with desktop machines, resulting in many copies of the same data (items like a spreadsheet or document sent to several people will be copied into each users' email inbox) In addition in such environments data is backed up frequently, creating even more copies. Very high compression ratios can be achieved by saving only a single copy of such data, using a process called (not surprisingly) *deduplication*. We see this in the figure below, where the data to be stored is 26 long, but only contains 9 unique blocks, giving a nearly 3:1 compression ratio if the 26 blocks of data are replaced by pointers to unique data blocks:

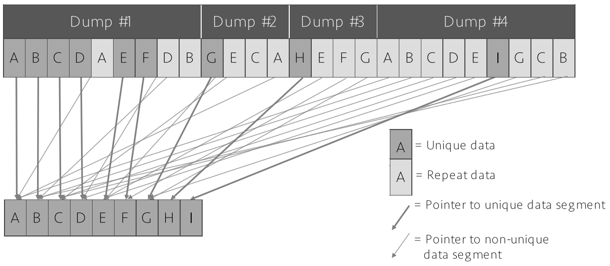

Dump #1
Dump #2
Dump #3
Dump #4
A B C D A E F D B G E C A H E F G A B C D E I G C B
A B C D E F G H I
A = Unique data
A = Repeat data
= Pointer to unique data segment
= Pointer to non-unique data segment

Figure 5.22: De-duplication

To perform deduplication, a cryptographic hash (a form of checksum) is calculated over each block to be written, and checked against a database. If the hash is found, then a block containing the same bits has already been written to storage, and we store a pointer to that block. If not—i.e. it is the first time we saw that particular data pattern—it is written to a new location on disk, and a pointer to that location is stored. By using this map we can then (somewhat slowly) retrieve the data later.

De-duplication is widely used for storing backups and retaining data for legal purposes, as it achieves very high compression (and thus lower cost) in many such cases. However, due to the overhead and non-sequential reads involved in retrieving data, it is typically much slower than normal storage.

5.5 Putting it all together

In our `ls` example the block layer and disk drive get used extensively by the file system. When the new process is created the kernel must read the first page from disk, to identify the type of executable, and then after the sections are mapped into memory, page faults will cause block read requests to be sent through the file system to the underlying device. Additional disk requests will come in response to the `readdir` system call, as the file system reads directory and inode data to list the files in a directory.

We'll ignore the file system for now, as it is described in more detail in later chapters, and focus on the role of the Linux block layer, which sits between file systems and the physical devices[8]. The block layer is organized around the `struct bio` object, a typical Linux kernel object which is fantastically complicated in order to track lots of things we don't really care about. We'll ignore most of this complexity; the fields that we're concerned with are the command flag (indicating read or write), data pointer (points to one or more pages), a callback function and private data field provided by the subsystem which submitted the I/O (more on this below), and a pointer to the device to which the I/O has been issued. (actually a pointer to a `struct block_device`)

First, a note about the private data pointer and callbacks, which are a common design pattern in C. (at least in the Linux kernel) In a proper object-oriented language, if you want to specialize a class (e.g. a block I/O descriptor) by adding additional fields (e.g. for details like timers or queues needed by your device driver), you can create a derived class with these additional fields. You can't do that in C—you can allocate two structures, or embed an instance of the general structure within the specialized one, but there will be cases (like callback functions) where a function handling the general class will in turn invoke another function which needs to access the specialized structure.

The most straightforward way to do this is via a "private data" field in a object; this is a generic pointer which is set to point to a separate structure holding the specialized data. An example shown in the listing below is the bio callback function (called `bi_end_io`): this is a function pointer which is invoked when the I/O operation completes, which is given a pointer to the bio itself as an argument.

```
struct my_data {
  ... specific data ...
};

void my_end_io(struct bio *b)
{
    struct my_data *md = b->bi_private;
    ...
}

...
{
```

[8]For a more detailed description of the Linux block layer, see https://lwn.net/Articles/736534/.

```
        struct bio *b = ...
        struct my_data *priv = ...
        b->bi_private = priv;
        b->bi_end_io = my_end_io;
        submit_bio(b);
    }
```

Listing 5.2: Using the `bi_private` field to pass information to a callback function

Note that `struct bio` has no way to indicate the *type* of attached data; instead we need to be sure that functions which interpret `bi_private` as a pointer to `struct my_data` are only ever called on bios where the attached object actually *is* of that type. (e.g. in this case `bi_end_io` will only be set to `my_end_io` in cases where the attached object is of type `struct my_data`)

Turning our attention back to the block layer, let's trace the case where a file system submits a single page read or write to a old-fashioned programmed-IO IDE drive. If you remember the IDE drive is similar to the disk controller described earlier in the text, with a few registers to indicate the disk sector, command (read / write), and the number of sectors to transfer, as well as a register which the CPU reads or writes to transfer the data. For a write you push the command and data, then wait for an interrupt to indicate that it's done; for a read you wait until the interrupt before transferring the data.

Here we see the path for submitting a read request in ext2[9]:

```
fs/ext2/inode.c:
793  int ext2_readpage(struct file *file, struct page *page) {
795         return mpage_readpage(page, ext2_get_block);

fs/mpage.c:
398 int mpage_readpage(struct page *page, get_block_t get_block) {
408        bio = do_mpage_readpage(bio, page, 1, &last_block_in_bio,
411               mpage_bio_submit(REQ_OP_READ, 0, bio);

143 struct bio *
144 do_mpage_readpage(struct bio *bio, struct page *page, ...) {
284               bio = mpage_alloc(bdev, blocks[0] << (blkbits - 9),

68  struct bio *
69  mpage_alloc(struct block_device *bdev, ...) {
77         bio = bio_alloc(gfp_flags, nr_vecs);
85               bio->bi_bdev = bdev;


59  struct bio *mpage_bio_submit(int op, int op_flags, ...
```

[9]Line numbers from Linux kernel 4.8.0

```
61        bio->bi_end_io = mpage_end_io;
64        submit_bio(bio);
```

Listing 5.3: Ext2 read bio submission

Ignoring all sorts of bookkeeping and optimizations, we have: a `bio` is allocated (`mpage_alloc` line 77) and a pointer is stored to the destination device (line 85), then a callback function is set (`mpage_bio_submit` line 61) and the I/O enters the block system via `submit_bio`.

From this point the block system generates a *request*[10] to the underlying device:

```
block/blk-core.c:
2067 blk_qc_t submit_bio(struct bio *bio) {
2099        return generic_make_request(bio);

1995 blk_qc_t generic_make_request(struct bio *bio) {
2036            struct request_queue *q = bdev_get_queue(bio->bi_bdev);
2039                        ret = q->make_request_fn(q, bio);
```

Listing 5.4: Submit_bio logic

We'll skip over the details of how I figured out what value `q->make_request_fn` has here; just trust me that in our case it's `blk_queue_bio`:

```
block/blk-core.c:
1663 blk_qc_t blk_queue_bio(struct request_queue *q, struct bio *bio)

1704        el_ret = elv_merge(q, &req, bio);
1705        if (el_ret == ELEVATOR_BACK_MERGE) {
1706              if (bio_attempt_back_merge(q, req, bio)) {
1710                      goto out_unlock;

1739        req = get_request(q, bio_data_dir(bio), rw_flags, bio, ...
1752        init_request_from_bio(req, bio);

1775               add_acct_request(q, req, where);
1776               __blk_run_queue(q);
```

Listing 5.5: `block/block-core.c`, `blk_queue_bio`

It first calls the "elevator" merge function (a reference to the classic disk scheduling algorithm) which tries to merge it with an existing queued I/O; if it can, then we return via `goto`[11] (lines 1704-1710). If not, we allocate a

[10]Unix block devices have always been different from normal files in that they have a single submission function for both reads and writes.

[11]The use of gotos to jump to cleanup code is a common design pattern in kernel coding, replacing the try/finally pattern in more civilized programming languages.

request structure (basically a bunch of information for queueing, hashing, accounting, sleeping, and stuff like that) and set up all its fields (lines 1739, 1752). Then we add the request to the elevator queue (line 1775, which in turn calls `__elv_add_request`, which has a lot of very complicated logic to figure out where to put the request in the queue) and then run a request from the queue:

```
block/block-core.c
311  inline void __blk_run_queue_uncond(struct request_queue *q)

324          q->request_fn(q);
```

Listing 5.6: Running a request from the queue

For a "legacy" (i.e. really old) IDE device the request function is `do_ide_request`. If you're looking at the code yourself, note that anything with `_pm_` in it is power management, that while `start_request` is important, `blk_start_request` doesn't do anything interesting, and that "plugging" refers to a complicated mechanism of delaying I/Os a short time to see if they'll be followed by additional requests that can be merged into one big request. You can skip over those parts; I did.

```
drivers/ide/ide-io.c:
456  void do_ide_request(struct request_queue *q)
517                      rq = blk_fetch_request(drive->queue);
551                 startstop = start_request(drive, rq);

block/blk-core.c:
2506 struct request *blk_fetch_request(struct request_queue *q)
2510         rq = blk_peek_request(q);

2349 struct request *blk_peek_request(struct request_queue *q)
2354          ... rq = __elv_next_request(q) ...
2399               ret = q->prep_rq_fn(q, rq);

drivers/ide/ide-io.c
306  ide_startstop_t start_request (ide_drive_t *drive, ... *rq)

343                 if (rq->cmd_type == REQ_TYPE_ATA_TASKFILE)
344                      return execute_drive_cmd(drive, rq);
```

So in order of execution, we grab a request from the queue (`blk-core.c` 2354) and call the queue prep function (`idedisk_prep_fn`, which sets `rq->cmd_type` to `REQ_TYPE_ATA_TASKFILE` and does a lot of other things we ignore), and then we call `start_request` (`ide-io.c` line 551) which calls `execute_drive_cmd` (line 344).

```
drivers/ide/ide-io.c
253  ide_startstop_t execute_drive_cmd (ide_drive_t *drive, ... *rq)
```

```
259          if (cmd->protocol == ATA_PROT_PIO) {
260                 ide_init_sg_cmd(cmd, blk_rq_sectors(rq) << 9);
261                 ide_map_sg(drive, cmd);
264          return do_rw_taskfile(drive, cmd);
```

If the drive controller is in programmed I/O mode (PIO), `ide_init_sg_cmd` creates a "taskfile", the bytes that have to be written to the control registers of the device; `ide_map_sg` gets pointers to all the memory regions to transfer. *Now* we're finally ready to send a command to the disk controller.

We'll trace a write operation, since it's easier:

```
drivers/ide/ide-taskfile.c:
78  ide_startstop_t do_rw_taskfile(ide_drive_t *drive, ...

118          tp_ops->tf_load(drive, &cmd->hob, cmd->valid.out.hob);
119          tp_ops->tf_load(drive, &cmd->tf,
cmd->valid.out.tf);

122      switch (cmd->protocol) {
123      case ATA_PROT_PIO:
123           if (cmd->tf_flags & IDE_TFLAG_WRITE) {
125                  tp_ops->exec_command(hwif, tf->command);
126                  ndelay(400);   /* FIXME */
127                  return pre_task_out_intr(drive, cmd);
```

(Fun fact: that FIXME comment was there in kernel 2.4.31 in 2005. I don't think it will get fixed.)

First the taskfile (and extended taskfile, known as the HOB since it's valid when the High Order Bit is set somewhere in the basic taskfile) to the controller, using `ide_tf_load`, which uses the `outb` instruction to write the bytes to the appropriate control registers; e.g. the 3 bytes of LBA in each get written as so:

```
            ...
        if (valid & IDE_VALID_LBAL)
               tf_outb(tf->lbal, io_ports->lbal_addr);
        if (valid & IDE_VALID_LBAM)
               tf_outb(tf->lbam, io_ports->lbam_addr);
        if (valid & IDE_VALID_LBAH)
               tf_outb(tf->lbah, io_ports->lbah_addr);
            ...
```

Then `ide_exec_command` writes the command byte to the appropriate register, and calls `pre_task_out_intr`:

```
drivers/ide/ide-taskfile.c:
403  ide_startstop_t pre_task_out_intr(ide_drive_t *drive, ... cmd)
```

```
419        ide_set_handler(drive, &task_pio_intr, WAIT_WORSTCASE);
421        ide_pio_datablock(drive, cmd, 1);
```

which sets a handler (saved in `hwif->handler`, with a timer in case the disk hangs) to be called when the request completes, and then actually copies the data to the data register.

We're almost done; bear with me. When the drive finishes writing its data, the IDE interrupt handler is called, which invokes the handler we just registered above, and then through a long, complicated chain of calls invokes `bio->bi_end_io`, which is the `mpage_end_io` that we stuck in the `bio` structure way back up at the top:

```
drivers/ide/ide-io.c:
892  irqreturn_t ide_intr (int irq, void *dev_id)
793        handler = hwif->handler;
849        startstop = handler(drive);

drivers/ide/ide-taskfile.c:
344  ide_startstop_t task_pio_intr(ide_drive_t *drive)
348        u8 stat = hwif->tp_ops->read_status(hwif);
  ... handle partial transfers; if done:
396        ide_complete_rq(drive, 0, blk_rq_sectors(cmd->rq) << 9);

drivers/ide/ide-io.c:
115  int ide_complete_rq(ide_drive_t *drive, int error, ...
128        rc = ide_end_rq(drive, rq, error, nr_bytes);

57  int ide_end_rq(ide_drive_t *drive, struct request *rq, ...
70        return blk_end_request(rq, error, nr_bytes);

block/blk-core.c
2796 bool blk_end_request(struct request *rq, int error, ...
2798      return blk_end_bidi_request(rq, error, nr_bytes, 0);

2740 bool blk_end_bidi_request(struct request *rq, int error,
2746      if (blk_update_bidi_request(rq, error, nr_bytes, ...

2654 bool blk_update_bidi_request(struct request *rq, int error,
2658       if (blk_update_request(rq, error, nr_bytes))

2539 bool blk_update_request(struct request *req, int error, ...
2604               req_bio_endio(req, bio, bio_bytes, error);

142  void req_bio_endio(struct request *rq, struct bio *bio, ...
155              bio_endio(bio);

block/bio.c:
1742 void bio_endio(struct bio *bio)
1761       if (bio->bi_end_io)
1762             bio->bi_end_io(bio);
```

Listing 5.7: The home stretch: from IDE interrupt to invoking bio->bi_end_io

Review questions

5.5.1. SSDs wear out faster if you repeatedly write to the same file or logical block address: *True / False*

5.5.2. Which one of the following statements is correct?

a) Deduplication is faster than traditional RAID arrays, but requires more disk space to hold the same amount of data
b) Deduplication is slower than traditional RAID arrays, but can hold more data with the same amount of disk space

Answers to Review Questions

5.2.1 (2) In general, connections which span longer distances and connect more devices (such as those far from the CPU) will be slower.
5.2.2 False. RAM and I/O devices (even memory-mapped I/O devices) are separate parts of the system.
5.2.1 False. The whole idea of an I/O (input/output) device is that the CPU doesn't know what value will be returned when it reads it.
5.2.2 False. DMA is when a device on the PCIe (or similar) bus accesses memory directly, without CPU intervention.
5.2.3 (2), software in the kernel. A device driver is that part of the kernel code which reads from, writes to, and handles interrupts from one or more specific hardware devices.
5.3.1 False. Since the platter is constantly spinning, when the head reaches the right track it may still have to wait as much as a full rotation for the target block to come beneath the head.
5.3.2 (3), multiple processes performing simultaneous random reads. In this case the OS can issue multiple read commands which are queued by the drive and completed in the most efficient order.
5.4.1 (2), a portion of the disk LBA space. The partition boundary is specified in a partition table in the beginning of the disk, and the operating system treats each partition as if it were a separate device.
5.4.1 (2), the storage capacity of a striped volume is the sum of the capacity of the disks in the volume, since only one copy of data is stored.
5.4.2 True. Stripes from each disk are interleaved at a fine granularity, so when one disk comes to an end, the entire volume has to end.
5.4.1 (1), each disk holds a copy of each byte written to the volume. (and no, there's no parity drive in a mirrored volume.)

5.4.1 False. The RAID 4 parity code can only recover from one missing drive, no matter how many drives are in the volume.

5.4.2 True. Three mirrored pairs hold three disks worth of data, while the six-disk RAID volume contains five disks of data.

5.4.3 (1) Once a single disk fails in a RAID 4 (or 5) volume, the data is unprotected and will be lost if a second disk fails. The sooner the disk is replaced, the less likely this is to happen.

5.4.4 (2) Small writes require reading old data and parity, and then writing data and parity, requiring four operations for a one-block write. Writing a full stripe set allows parity to be calculated without reading any information from disk, adding only a single operation to the parity drive.

5.4.1 False. RAID 5 and RAID 4 can both tolerate only a single disk failure without data loss.

5.4.2 False. If an entire stripe set is written at once, the parity can be calculated and written with it, resulting in one write operation for each drive in the array, regardless of whether it is RAID 4 or RAID 5.

5.4.3 True. A small write requires four operations: read (1) old data, (2) old parity, write (3) new data, (4) new parity. In RAID 4, two of these always go to the same (parity) drive, which becomes a bottleneck.

5.4.4 (2) Modern disks do not seem to fail more or less frequently than those of several years ago. Similarly, the probability of losing a single block of data to an unrecoverable read error has stayed roughly the same (as of 2015); however, the number of data blocks on a single disk has grown hugely, making it far more likely that one of the data blocks on a disk will be lost.

5.5.1 False. SSD algorithms distribute writes evenly over the internal flash, whether writes are to the same or different block addresses.

5.5.2 (2) Writing to a de-duplicated volume is slower due to the need to search for possible duplicates. Reading is typically much slower, as well, because the fragments making up a file will not be sequential on the underlying disk. For many workloads, however, it may be possible to store 10 times as much data on the same number of disks.

Chapter 6

File Systems

General-purpose operating systems typically provide access to block storage (i.e. disks) via a *file system*, which provides a much more application- and user-friendly interface to storage. From the point of view of the user, a file system contains the following elements:

- a *name space*, the set of names identifying objects;
- *objects* such as the files themselves as well as directories and other supporting objects;
- *operations* on these objects.

Hierarchical namespace: File systems have traditionally used a tree-structured namespace[1], as shown Figure 6.1. This tree is constructed via the use of *directories*, or objects in the namespace which map strings to further file system objects. A full filename thus specifies a *path* from the root, through the tree, to the object (a file or directory) itself. (Hence the use of the term "path" to mean "filename" in Unix documentation)

File: Early operating systems supported many different file types—binary executables, text files, and record-structured files, and others. The Unix operating system is the earliest I know of that restricted files to sequences of 8-bit bytes; it is probably not a coincidence that Unix arrived at the same time as computers which dealt only with multiples of 8-bit bytes (e.g. 16 and 32-bit words), replacing older systems which frequently used odd word sizes such as 36 bits. (Note that a machine with 36-bit instructions already needs two incompatible types of files, one for text and one for executable code)

[1]Very early file systems sometimes had a single flat directory per user, or like MS-DOS 1.0, a single directory per floppy disk.

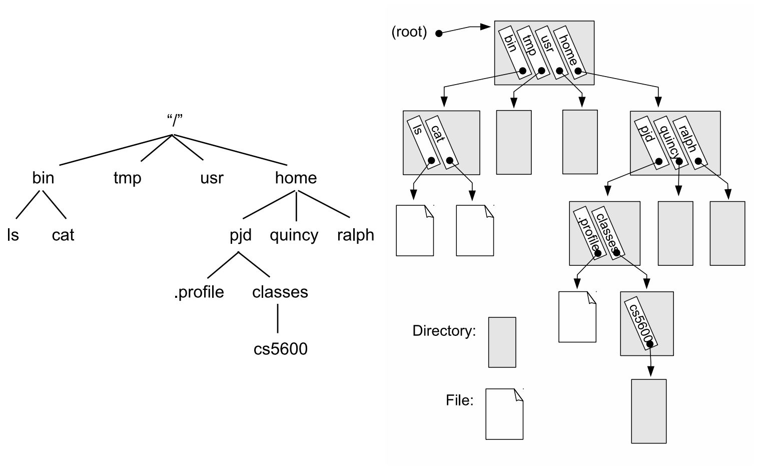


Figure 6.1: Logical view (left) and implementation (right) of a hierarchical file system name space.

Modern operating systems follow the UNIX model, which imposes no structure on a file—a file is merely a sequence of bytes.[2] Any structure to the file (such as a JPEG image, an executable program, or a database) is the responsibility of applications which read and write the file. The file format is commonly indicated by a file extension like .jpg or .xml, but this is just a convention followed by applications and users. You can do things like rename file.pdf to file.jpg, which will confuse some applications and users, but have no effect on the file contents.

Data in a byte-sequence file is identified by the combination of the file and its offset (in bytes) within the file. Unlike in-memory objects in an application, where a reference (pointer) to a component of an object may be passed around independently, a portion of a file cannot be named without identifying the file it is contained in. Data in a file can be created by a write which appends more data to the end of a shorter file, and modified by over-writing in the middle of a file. However, it can't be "moved" from one offset to another: if you use a text editor to add or delete text in the middle of a file, the editor must re-write the entire file (or at least from the modified part to the end).

Unix file name translation: each process has an associated *current di-*

[2]Almost. Apple OSX uses *resource forks* to store information associated with a file (HFS and HFS+ file systems only), Windows NTFS provides for multiple data streams in single file, although they were never put to use, and several file systems support *file attributes*, small tags associated with a file..

rectory, which may be changed via the `chdir` system call. File names beginning in '`/`' are termed *absolute* names, and are interpreted relative to the root of the naming tree, while *relative* names are interpreted beginning at the current directory. (In addition, `d/..` always points to the parent directory of `d`, and `d/.` points to `d` itself.) Thus in the file system in Figure 6.1, if the current directory were `/home`, the the paths `pjd/.profile` and `/home/pjd/.profile` refer to the same file, and `../bin/cat` and `/bin/cat` refer to the same file.

6.1 File System Operations:

There are several common types of file operations supported by Linux (and with slight differences, Windows). They can be classified into three main categories: open/close, read/write, and naming and directories.

Open/close: In order to access a file in Linux (or most operating systems) you first need to open the file, passing the file name and other parameters and receiving a *handle* (called a *file descriptor* in Unix) which may be used for further operations. The corresponding system calls are:

- `int desc = open(name, O_READ)` - Verify that file `name` exists and may be read, and then return a *descriptor* which may be used to refer to that file when reading it.
- `int desc = open(name, O_WRITE | flags, mode)` - Verify permissions and open `name` for writing, creating it (or erasing existing contents) if necessary as specified in `flags`. Returns a descriptor which may be used for writing to that file.
- `close(desc)` - stop using this descriptor, and free any resources allocated for it.

Note that application programs rarely use the system calls themselves to access files, but instead use higher-level frameworks, ranging from Unix Standard I/O to high-level application frameworks.

Read/Write operations: To get a file with data in it, you need to write it; to use that data you need to read it. To allow reading and writing in units of less than an entire file, or tedius calculations of the current file offset, UNIX uses the concept of a *current position* associated with a file descriptor. When you read 100 bytes (i.e. bytes 0 to 99) from a file this pointer advances by 100 bytes, so that the next read will start at byte 100, and similarly for write. When a file is opened for reading the pointer starts at 0; when open for writing the application writer can choose to start at the beginning (default) and overwrite old data, or start at the end (`O_APPEND` flag) to append new data to the file.

System calls for reading and writing are:

- `n = read(desc, buffer, max)` - Read `max` bytes (or fewer if the end of the file is reached) into `buffer`, starting at the current position, and returning the actual number of bytes `n` read; the current position is then incremented by `n`.
- `n = write(desc, buffer, len)` - write `len` bytes from `buffer` into the file, starting at the current position, and incrementing the current position by `len`.
- `lseek(desc, offset, flag)` Set an open file's current position to that specified by `offset` and `flag`, which specifies whether `offset` is relative to the beginning, end, or current position in the file.

Note that in the basic Unix interface (unlike e.g. Windows) there is no way to specify a particular location in a file to read or write from[3]. Programs like databases (e.g. SQLite, MySQL) which need to write to and read from arbitrary file locations must instead move the current position by using `lseek` before a read or write. However most programs either read or write a file from the beginning to the end (especially when written for an OS that makes it easier to do things that way), and thus don't really need to perform seeks. Because most Unix programs use simple "stream" input and output, these may be re-directed so that the same program can—without any special programming—read from or write to a terminal, a network connection, a file, or a pipe from or to another program.

Naming and Directories: In Unix there is a difference between a name (a directory entry) and the object (file or directory) that the name points to. The naming and directories operations are:

- `rename(path1, path2)` - Rename an object (i.e. file or directory) by either changing the name in its directory entry (if the destination is in the same directory) or creating a new entry and deleting the old one (if moving into a new directory).
- `link(path1, path2)` Add a *hard link* to a file[4].

[3]On Linux the `pread` and `pwrite` system calls allow specifying an offset for the read or write; other UNIX-derived operating systems have their own extensions for this purpose.

[4]A hard link is an additional directory entry pointing to the same file, giving the file two (or more) names. Hard links are peculiar to Unix, and in modern systems have mostly been replaced with symbolic links (covered next); however Apple's Time Machine makes very good use of them: multiple backups can point to the same single copy of an un-modified file using hard links.

- `unlink(path)` - Delete a file.[5]
- `desc = opendir(path)`
  `readdir(desc, dirent*), dirent=(name,type,length)`
  This interface allows a program to enumerate names in a directory, and determine their type. (i.e. file, directory, symbolic link, or special-purpose file)
- `stat(file, statbuf)`
  `fstat(desc, statbuf)` - returns file attributes - size, owner, permissions, modification time, etc. In Unix these are attributes of the file itself, residing in the i-node, and can't be found in the directory entry - otherwise it would be necessary to keep multiple copies consistent.
- `mkdir(path)`
  `rmdir(path)` - directory operations: create a new, empty directory, or delete an empty directory.

Review Questions

6.1.1. Directories in most file systems only contain pointers to files, not to other directories: *True / False*

6.1.2. Which one or more of the following scenarios could cause the contents of the 1000th byte in a file to either change or cease to exist?

a) The file is renamed
b) The file is deleted
c) Bytes 500 through 600 in the file are over-written
d) Bytes 900 through 1200 are over-written

6.1.3. For the read operation `read(handle, buffer, max)`, the range of bytes to be read from the file (e.g. bytes 100 through 199) is determined by which of the following? (more than one may apply)

a) The 'buffer' and 'max' arguments
b) The file handle current position and file length
c) The 'max' argument
d) bytes 0 through 'max'

[5] Sort of. If there are multiple hard links to a file, then this just removes one of them; the file isn't deleted until the last link is removed. Even then it might not be removed yet - on Unix, if you delete an open file it won't actually be removed until all open file handles are closed.. In general, deleting open files is a problem: while Unix solves the problem by deferring the actual delete, Windows solves it by protecting open files so that they cannot be deleted

Symbolic links

An alternative to hard links to allow multiple names for a file is a third file system object (in addition to files and directories), a *symbolic link*. This holds a text string which is interpreted as a “pointer” to another location in the file system. When the kernel is searching for a file and encounters a symbolic link, it substitutes this text into the current portion of the path, and continues the translation process.

Thus if we have:

```
directory: /usr/program-1.0.1
file:      /usr/program-1.0.1/file.txt
sym link: /usr/program-current -> "program-1.0.1"
```

and if the OS is looking up the file `/usr/program-current/file.txt`, it will:

1. look up `usr` in the root directory, finding a pointer to the `/usr` directory
2. look up `program-current` in `/usr`, finding the link with contents `program-1.0.1`
3. look up `program-1.0.1` and use this result instead of the result from looking up `program-current`, getting a pointer to the `/usr/program-1.0.1` directory.
4. look up `file.txt` in this directory, and find it.

Note that unlike hard links, a symbolic link may be “broken”—i.e. if the file it points to does not exist. This can happen if the link was created in error, or the file or directory it points to is deleted later. In that case path translation will fail with an error:

```
pjd-1:tmp pjd$ ln -s /bad/file/name bad-link
pjd-1:tmp pjd$ ls -l bad-link
lrwxr-xr-x 1 pjd wheel 22 Aug 2 00:07 bad-link -> /bad/file/name
pjd-1:tmp pjd$ cat bad-link
cat: bad-link: No such file or directory
```

Finally, to prevent loops there is a limit on how many levels of symbolic link may be traversed in a single path translation:

```
pjd@pjd-fx:/tmp$ ln -s loopy loopy
pjd@pjd-fx:/tmp$ ls -l loopy
lrwxrwxrwx 1 pjd pjd 5 Aug 24 04:25 loopy -> loopy
pjd@pjd-fx:/tmp$ cat loopy
cat: loopy: Too many levels of symbolic links
pjd@pjd-fx:/tmp$
```

In early versions of Linux (pre-2.6.18) the link translation code was recursive, and this limit was set to 5 to avoid stack overflow. Current versions use an iterative algorithm, and the limit is set to 40.

Device Names vs. Mounting: A typical system may provide access to several file systems at once, e.g. a local disk and an external USB drive or network volume. In order to unambiguously specify a file we thus need to both identify the file within possibly nested directories in a single file system, as well as identifying the file system itself. (in Unix this name is called an *absolute pathname*, providing an unambiguous "path" to the file.) There are two common approaches to identifying file systems:

- Explicitly: each file system is given a name, so that a full pathname looks like e.g. `C:\MyDirectory\file.txt` (Windows[6]) or `DISK1:[MYDIR]file.txt` (VMS).
- Implicitly: a file system is transparently *mounted* onto a directory in another file system, giving a single uniform namespace; thus on a Linux system with a separate disk for user directories, the file "/etc/passwd" would be on one file system (e.g. "disk1"), while "/home/pjd/file.txt" would be on another (e.g. "disk2").

The actual implementation of mounting in Linux and other Unix-like systems is implemented via a *mount table*, a small table in the kernel mapping directories to directories on other file systems. In the example above, one entry would map "/home" on disk1 to ("disk2", "/"). As the kernel translates a pathname it checks each directory in this table; if found, it substitutes the mapped file system and directory before searching for an entry. Thus before searching "/home" on disk1 (which is probably empty) for the entry "pjd", the kernel will substitute the top-level directory on disk2,and then search for "pjd".

For a more thorough explanation of path translation in Linux and other Unix systems see the `path_resolution(7)` man page, which may be accessed with the command `man path_resolution`.

Review Questions

6.1.1. Creating, modifying, and deleting directories is performed by different system calls than creating and deleting files. Which of the following are possible reasons for this?

a) When deleting a directory, the OS must check to be sure that it is empty
b) Directories use a different kind of name from files
c) To prevent users from modifying directory data which is accessed by kernel code.

[6]Modern Windows systems actually use a mount-like naming convention internally; e.g. the `C:` drive actually corresponds to the name `\DosDevices\C:` in this internal namespace.

6.1.2. Which one of the following statements best describes the Unix mount table?

a) Used at startup to determine how to name different filesystems
b) A table in the kernel used to recognize where one filesystem is “attached” to another

6.2 File System Layout

To store a file system on a real disk, the high-level objects (directories, files, symbolic links) must be translated into fixed-sized blocks identified by logical block addresses.

Note that instead of 512-byte sectors, file systems traditionally use *disk blocks*, which are some small power-of-two multiple of the sector size, typically 1KB, 2KB, or 4KB. Reading and writing is performed in units of complete blocks, and addresses are stored as disk block numbers rather than LBAs, and are then multiplied by the appropriate value before being passed to the disk. Since modern disk drives have an internal sector size of 4 KB (despite pretending to support 512-byte sectors) and the virtual memory page size is 4 KB on most systems today, that has become a very common file system block size.

Designing on-disk data structures is complicated by the fact that for various reasons (virtual memory, disk controller restrictions, etc.) the data in a file needs to be stored in full disk blocks — e.g. bytes 0 through 4095 of a file should be stored in a single 4096-byte block. (This is unlike in-memory structures, where odd-sized allocations usually aren’t a problem.)

In this section we examine a number of different file systems; we can categorize them by the different solutions their designers have come up with for the following three problems:

1. How to find objects (files, directories): file identification.
2. How to find the data within a file: file organization.
3. How to allocate free space for creating new files.

CD-ROM File System

In Figure 6.2 we see an example of an extremely simple file system, similar to early versions of the ISO-9660 file system for CD-ROM disks. Objects on disk are either files or directories, each composed of one or more 2048-

byte[7] *blocks*; all pointers in the file system are in terms of *block numbers*, with blocks numbered from block 0 at the beginning of the disk.

There are no links—each object has exactly one name—and the type of an object is indicated in its directory entry. (The only exception is the root directory, which has no name; however it is always found at the beginning of the disk) Finally, all objects are *contiguous*, allowing them to be identified by a starting block number and a length.

This organization is both compact and fairly efficient. As in almost all file systems, an object is located by using linear search to find each path component in the corresponding directory. Once a file is located, access to any position is straightforward and can be calculated from the starting block address of the file, as all files are contiguous.

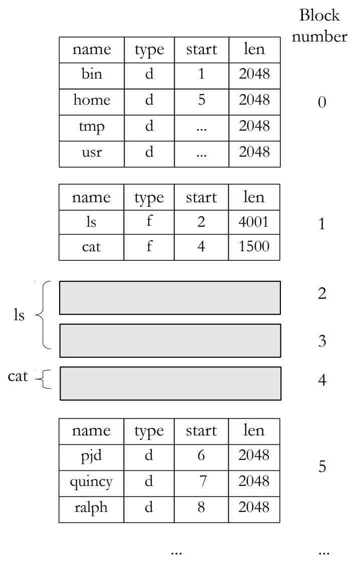


Figure 6.2: Simplified ISO-9660 (CD-ROM) file layout for tree in Figure 6.1, 2KB blocks

Contiguous organization works fine for a read-only file system, where all files (and their sizes) are available when the file system is created. It works poorly for writable file systems, however, as space would quickly fragment making it impossible to create large files. (Also the CDROM file system has no method for tracking free space, so allocation would be very inefficient.)

In the simple CD-ROM file system, what were the solutions to the three design problems?

1. File identification: files are identified by their starting block number
2. File organization: blocks in a file are contiguous, so an offset in the file can be found by adding to the starting block number.
3. Free space allocation: since it's a read-only file system, there is no free space to worry about.

[7]Why 2048? Because the designers of the CDROM file system defined it that way. Data is stored on CD in 2048-byte blocks plus error correction, making use of smaller block sizes difficult, and the authors evidently didn't see any need to allow larger block sizes, either.

Review Questions

6.2.1. On-disk structures must be constructed of disk blocks, rather than arbitrary-sized regions: *True / False*

6.2.2. A file system can use large blocks for the large files in a directory and small blocks for the small files: *True / False*

6.2.3. Not counting blocks used for the directory, how much space would be required to store 20 files, each 100 bytes long, in the CD-ROM file system described?

a) It would require 20 2048-byte blocks
b) It would require a single 2048-byte block

6.2.4. The CD-ROM file system described in this chapter tracks free space in its directories: *True / False*

MS-DOS file system

The next file system we consider is the MS-DOS (or FAT, File Allocation Table) file system. Here blocks within a file are organized in a linked list; however implementation of this list is somewhat restricted by the requirement that all access to the disk be done in multiples of a fixed block size.[8] Instead these pointers are kept in a separate array, with an entry corresponding to each disk block, in what is called the File Allocation Table.

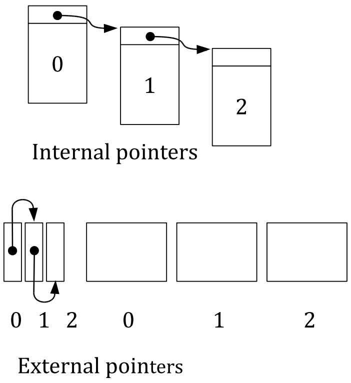


Figure 6.3: Linked list organization with in-object pointers (typical for in-memory structures) and external pointers, as used in MS-DOS File Allocation Table.

Entries in this table can indicate (a) the number of the next block in the file or directory, (b) that the block is the last one in a file or directory, or (c) the block is free. The FAT is thus used for free space management as well as file organization; when a block is needed the table may be searched for a free entry which can then be allocated.

Again, what were the solutions to the three design problems?

1. File identification - Files and directories are identified by their starting block number
2. File organization - blocks within a file are linked by pointers in the FAT
3. Free space allocation - free blocks are marked in the FAT, and linear search is used to find free space

Directories are similar to the CD-ROM file system - each entry has a name, the object type (file or directory), its length, and the starting address of the file contents. Note that although the last block of a file can be identified by a flag in the FAT, the length field is not redundant as it is still needed to know how much of the last block is valid. (E.g. a 5-byte file will require

[8]The astute reader will note that the pointer could use bytes within a block, causing each block to store slightly less than a full block of data. This would pose difficulties for operating systems such as Linux which tightly couple the virtual memory and file systems, and assume that each 4 KB virtual memory page corresponds to one (or maybe 2 or 4) file system blocks.

an entire block, but will only use 5 bytes in that block.) Sequential access to a file incurs overhead to fetch file allocation table entries, although since these are frequently used they may be cached; random access to a file, however, requires walking the linked list to find the corresponding entry, which can be slow even when cached in memory. (Consider random I/O within a 1 GB virtual disk image with 4 KB blocks—the linked list will be 256K long, and on average each I/O will require searching halfway through the list[9]).

Directories in the MS-DOS file system are similar to those in ISO-9660. Each directory entry is a fixed size and has a field indicating whether it is valid; to delete a file, this field is set to invalid and the blocks in that file are marked as free in the file allocation table. Only a single name per file is supported, and all file metadata (e.g. timestamps, permissions) is stored in the directory entry along with the size and first block number.

Like most file systems, linear search is used to locate a file in a directory. This is usually reasonably efficient (it's used by most Unix file systems, too) but works poorly for very large directories. (That's why your browser cache has filenames that look like `ab/xy/abxy123x.dat`, instead of putting all its files in the same directory.)

A note for the reader - the original MS-DOS file system only supported 8-byte upper-case names with 3-byte extensions, with (seemingly) no way to get around this restriction, since the size of a directory entry is fixed. A crazy mechanism was devised that is still used today: multiple directory entries are used for each file, with the extra entries filled with up to 13 2-character Unicode filename characters in not only the filename field, but also the space that would have otherwise been used for timestamp, size, starting block number, etc., and marked in a way that would be ignored by older versions of MS-DOS.

Review Questions

6.2.1. The MS-DOS file system identifies the blocks in a file through which of the following processes?

a) By marking them with the file ID in the file allocation table
b) By linking them with pointers at the beginning of each block
c) By linking them with pointers in the FAT

6.2.2. Which of these are differences between ext2 and the MS-DOS file system described previously?

[9] A benchmark run on login.ccs.neu.edu indicates that "pointer chasing" on a high-end Xeon takes about 200 ns when data is not in cache; each such random I/O would thus take about 25 ms of CPU time.

a) ext2 allows multiple names for the same file, while MS-DOS only allows one.
b) ext2 has less overhead than the MS-DOS file system.

Unix file systems (e.g. ext2)

File systems derived from the original Unix file system (e.g. Linux ext2 and ext3) use a per-file structure called an inode ("indirect node") designed with three goals in mind: (a) low overhead for small files, in terms of both disk seeks and allocated blocks[10], (b) ability to represent sufficiently large files without excessive storage space or performance overhead, and (c) crash resiliency—crashing while the file is growing should not endanger existing data.

Why not use e.g. a balanced binary tree? The in-memory tree structures from your algorithms class aren't appropriate for a file system, for several reasons: (a) the minimum allocation unit is a disk block, typically 4 KB, (b) disk seeks are really expensive, and (c) we want to avoid re-arranging existing data on disk as the file grows, so that we don't lose it if the system crashes mid-operation.

To do this, the inode uses an asymmetric tree, or actually a series of trees of increasing height with the root of each tree stored in the inode. As seen in Figure 6.4 the inode contains N *direct* block pointers (12 in ext2/ext3), so that files of N blocks or less need no indirect blocks. A single *indirect pointer* specifies an *indirect block*, holding pointers to blocks $N, N+1, ...N+N_1-1$ where N_1 is the number of block numbers that fit in a file system block (1024 for ext2 with a 4 KB blocksize). If necessary, the *double-indirect pointer* specifies a block holding pointers to N_1 indirect blocks, which in turn hold pointers to blocks $N+N_1...N+N_1+N_1^2-1$—i.e. an N_1-ary tree of height 2; a triple indirect block in turn points to a tree of height 3. For ext2 with 4-byte block numbers, if we use 4K blocks this gives a maximum file size of $(4096/4)^3$ 4 KiB[11] blocks, or 4.004 TiB. This organization allows random access within a file with overhead $O(logN)$ where N is the file size, which is vastly better than the $O(N)$ overhead of the MS-DOS File Access Table system.

In addition to the block pointers, the inode holds file *metadata* such as the owner, permissions, size, and timestamps. The separation of name (i.e. directory entry) and object (the inode and the blocks it points to) also allows files to have multiple names, which for historical reasons are called

[10]The median file size in a recent study was 4 KB, or one block

[11]When we're being really precise, we'll use KiB, MiB, GiB etc. to mean 2^{10}, 2^{20}, 2^{30} and KB, MB, GB to mean 10^3, 10^6 and 10^9.

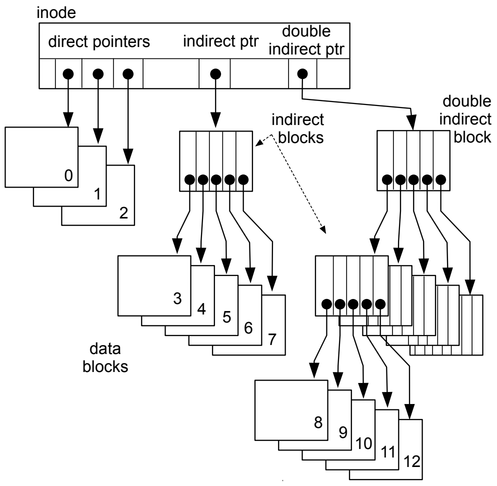


Figure 6.4: Inode-type file organization as found in many Unix file systems (e.g. Linux ext2, ext3). Note that the degree of branching is far lower than in real file systems, and the triple-indirect pointer is missing.

hard links. For the longest time hard links were a little-used capability of Unix-style file systems; however Apple Time Machine for the HFS+ file system makes good use of them to create multiple backup snapshots which share identical files to save space.

Since files can have multiple names, the inode also contains a reference count; as each name is deleted (via the `unlink` system call) the count is decremented, and when the count goes to zero the file is deleted. This also allows a file to have *zero* names—when a file is opened the reference count (in memory, not on disk) is incremented, and decremented when it is closed, so if you unlink a file which is in use, it is not actually deleted until the last open file descriptor is closed[12].

Ext2 space allocation: The original Unix file system used a free list to store a list of unused blocks; blocks were allocated from the head of this list for new files, and returned to the head when freed. As files were created and deleted this list became randomized, so that blocks allocated for a file were rarely sequential and disk seeks were needed for nearly every block read from or written to disk. This wasn't a significant problem, because

[12]Deleting open files is a tricky problem, as there's no good way to handle operations on those open handles after the file is deleted. Unix solves it by postponing the actual deletion until the file descriptor is closed; Windows instead locks the file against deletion until any open file handles are closed.

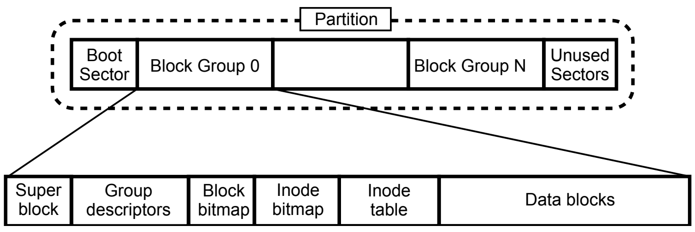


Figure 6.5: Ext2 on-disk layout

early Unix systems ran on machines with fast disks and excruciatingly slow CPUs. As computers got faster and users started noticing that the file system was horribly slow, the Fast File System (FFS) from Berkeley Unix replaced the free list with a more efficient mechanism, the *allocation bitmap*.

Ext2 is essentially a copy of FFS, and uses this bitmap mechanism. It keeeps a boolean array with one bit for each disk block; if the block is allocated the corresponding bit is set to '1', and cleared to '0' if it is freed. To allocate a block you read a portion of this bitmap into memory and scan for a '0' bit, changing it to '1' and writing it back. When you extend a file you begin the search at the bit corresponding to the last block in the file; in this way if a sequential block is available it will be allocated. This method results in files being allocated in a mostly sequential fashion, reducing disk seeks and greatly improving performance. (An additional bitmap is used for allocating inodes; in this case we don't care about sequential allocation, but it's a compact representation, and we can re-use some of the code written for block allocation.)

Block groups, as shown in Figure 6.5, are an additional optimization from FFS. Each block group is a miniature file system, with block and inode bitmaps, inodes, and data blocks. The file system tries to keep the inode and data blocks of a file in the same block group, as well as a directory and its contents. In this way common operations (e.g. open and read a file, or 'ls -l') will typically access blocks within a single block group, avoiding long disk seeks.

Long file names: Ext2 supports long file names using the mechanism used in FFS. Rather than treating the directory as an array of fixed-sized structures, it is instead organized as a sequence of length/value-encoded entries, with free space treated as just another type of entry. Directory search is performed using linear search.

Ext2 solutions to the three design problems?

1. File identification - files and directories are identified by inode number, and the location of the fixed-sized inode can be calculated from inode number and the inode table location.
2. File organization - blocks within a file are located via pointers from the inode
3. Free space allocation - free blocks are tracked in a free-space bitmap, and block groups are used to keep blocks from the same file near to each other, their inode, and their directory.

Note the difference here between the data structure (a bitmap) and strategies used such as trying to allocate the block immediately after the previous one written. The MS-DOS file system organizes its free list in an array, as well, and most of the allocation techniques introduced in the Berkeley Unix file system could be used with it. In practice, however, the MS-DOS file system was typically implemented with simple allocation strategies that resulted in significant file system fragmentation.

An additional anti-fragmentation strategy used by many modern operating systems is the enforcement of a maximum utilization, typically 90% or 95%, as when a file system is almost full, it is likely that any free space will be found in small fragments scattered throughout the disk. By limiting utilization to e.g. 90%—i.e. one block out of ten is free—we significantly increase the chance of finding multiple contiguous blocks when writing to a file, while greatly decreasing the fraction of the bitmap we may need to search to find a free block.

6.3 Superblock

Before a disk can be used in most systems it needs to be *initialized* or *formatted*—the basic file system structures need to be put in place, describing a file system with a single directory and no files. A key structure written in this process is the *superblock*, written at a well-known location on the disk. (This is often block 1, allowing block 0 to be used by the boot loader.) The superblock specifies various file system parameters, such as:

- Block size - most file systems can be formatted with different block sizes, and the OS needs to know this size before it can interpret any pointers given in terms of disk blocks. Historically larger blocks were used for performance and to allow larger file systems, and smaller blocks for space efficiency. In recent years disk drives have transitioned to using an internal block size of 4KB, while keeping the traditional 512-byte sector addressing, so any file system should use a block size of at least 4KB.

- Version - including a version number allows backwards compatibility as a file system evolves. That way you can upgrade your OS, for instance, without reformatting your disk.
- Other parameters - in the MS-DOS file system the OS needs to know how large the FAT table is, so that it doesn't accidently go off the end and start looking at the first data block. In ext2 you need to know the sizes of the block groups, as well as the bitmap sizes, how many inodes are in each group, etc.
- Dirty flag - when a file system is mounted, this flag is set; as part of a clean shutdown the flag is cleared again. If the system crashes without clearing the flag, at the next boot this indicates that additional error checks are needed before mounting the file system.

6.4 Extents, NTFS, and Ext4

The ext2 and MS-DOS file systems use separate pointers to every data block in a file, located in inodes and indirect blocks in the case of ext2, and in the file allocation table in MS-DOS. But the values stored in these pointers are often very predictable, because the file system attempts to allocate blocks sequentially to avoid disk seeks—if the first block in a file is block 100, it's highly likely that the second will be 101, the third 102, etc.

We can take advantage of this to greatly compress the information needed to identify the blocks in a file - rather than having separate pointers to blocks 100,101,…120 we just need to identify the starting block (100) and the length (21 blocks). This is shown in Figure 6.6, where five data blocks are identified by inodes or indirect block pointers; to the right, the same five data blocks are identified by a single extent. Why would we want to compress the information needed to organize the blocks in a file? Mostly for performance—although the code is more complicated, it will require fewer disk seeks to read from disk.

This organization is the basis of *extent-based* file systems, where blocks in a file are identified via one or more *extents*, or (start,length) pairs. The inode (or equivalent) can contain space for a small number of extents; if the file grows too big, then you add the equivalent of indirect blocks - extents pointing to blocks holding more extents. Both Microsoft NTFS and Linux ext4 use this sort of extent structure.

NTFS: Each NTFS file system has a Master File Table (MFT), which is somewhat like the inode table in ext2—each file or directory has an entry in this table which holds things like permissions, timestamps, and block information. (The superblock contains a pointer to the start of the MFT;

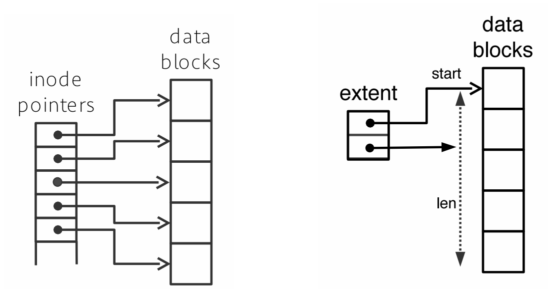


Figure 6.6: File structure—pointers vs. extents

the first entry in the MFT describes the MFT itself, so that it can grow as needed.) Each entry in the MFT is structured as a set of attributes, with a $Data attribute specifying the file contents. This attribute can be of two types: internal, where the attribute holds file data directly (for very small files), or external, in which case the $Data attribute holds a list of *extents*, or contiguous regions identified by a starting block and length.

If the number of extents grows too large to fit into the MFT entry, an $ATTRIBUTE_LIST field is added, holding a list of extents describing the blocks holding the list of extents describing the file. This can continue for one more level, which is enough to support files up to 16TB. Note that the amount of space taken by the $Data attribute depends not only on the size of the file, but its fragmentation; a very large file created on an empty file system might consist of only a few extents, while a modest-sized file created slowly (e.g. a log file) on a full file system might be composed of hundreds of extents.

Free space is handled similarly, as a list of extents sorted by starting block number; this allows the free space list to be easily compacted when storage is freed. (i.e. just by checking to see if it can be combined with its neighbors on either side) This organization makes it easy to minimize file fragmentation, reducing the number of disk seeks required to read a file or directory. It has the disadvantage that random file access is somewhat more complex, and appears to require reading the entire extent list to find which extent an offset may be found in. (A more complex organization could in fact reduce this overhead; however in practice it does not seem significant, as unless highly fragmented the extent lists tend to be fairly short and easily cached.)

NTFS solutions to the three design problems?

1. file identification - Master File Table entry

2. file organization - (possibly multi-level) extent list
3. free space management - sorted extent list.

Ext4: Ext4 supports extent-based file organization with minimal change to the inode structure in ext2/ext3: an extent tree is used, with each node explicitly marked as an interior or leaf node, as shown in Figure 6.7. The inode holds a four-entry extent tree node, allowing small files to be accessed without additional lookup steps, while for moderate-sized files only a single level of the tree (a "leaf node" in the figure) is needed.

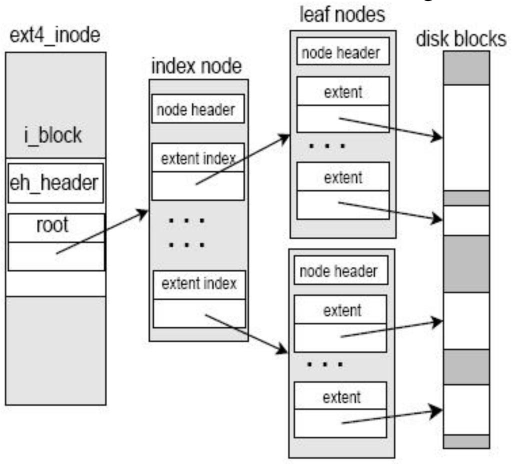


Figure 6.7: Ext4 on-disk structure

6.5 Smarter Directories

In the CD-ROM, MS-DOS, and ext2 file systems, a directory is just an array of directory entries, in unsorted order. To find a file, you search through the directory linearly; to delete a file, you mark its entry as unused; finally, to create a new entry, you find any entry that's free. (It's a bit more complicated for file systems like ext2 which have variable-length directory entries, but not much.)

From your data structures class you should realize that linear search isn't an optimal data structure for searching, but it's simple, robust, and fast enough for small directories, where the primary cost is retrieving a block of data from the disk. As an example, one of my Linux machines has 94944 directories that use a single 4KB block, another 957 that use 2 to 5 blocks, and only 125 larger than 5 blocks. In other words, for the 99% of the directories that fit within a single 4 KB block, a more complex algorithm would not reduce the amount of data read from disk, and the difference between $O(N)$ and $O(logN)$ algorithms when searching a single block is negligible.

However the largest directories are actually quite big: the largest on this machine, for example, has 13,748 entries; another system I measured had a database directory containing about 64,000 files with long file names, or roughly 4000 blocks (16 MB) of directory data. Since directories tend to grow slowly, these blocks were probably allocated a few at a time, resulting in hundreds or thousands of disk seeks to read the entire directory into memory. At 15 ms per seek, this could require 10-30 seconds or more, and once the data was cached in memory, linear search in a 16 MB array will probably take a millisecond or two.

To allow directories with tens of thousands of files or more, modern file systems tend to use more advanced data structures for their directories. NTFS (and Linux Btrfs) use B-trees, a form of a balanced tree. Other file systems, like Sun ZFS, use hash tables for their directories, while ext4 uses a hybrid hash/tree structure. If you're really interested, you can look these up in Google.

6.6 The B-tree

The B-tree is one of those widely-used data structures that you never see in your data structures course. It's not a file system— the B-tree is a disk-optimized search structure, optimized for the case where accessing a block of information is much more expensive (e.g. requiring a disk seek) than searching through that block after it has been accessed. It has been used for file systems, databases, and similar purposes since the 1970s, along with various extensions (e.g. B^+-trees) which are not described here.

B-tries are balanced trees made up of large blocks, with a high branching factor, in order to reduce the number of block accesses needed for an operation. Interior and leaf nodes are identical; each contains a sorted list of key/-value pairs, and (in non-leaf nodes) pointers between pairs of keys, pointing to subtrees holding keys which are between those two values. The tree grows from the bottom up: if a block overflows, you split it, dividing the contents between two blocks, and add a pointer to the new block in the correct position in the parent; if the parent overflows it is split, and so on. If the root node splits, a new root is allocated with pointers to the two pieces.

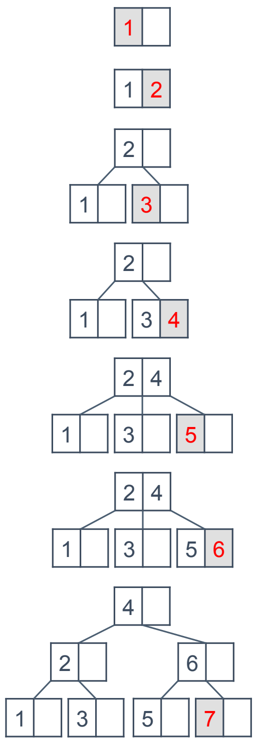


Figure 6.8: B-tree growth

If the branching factor of a B-tree is m, then each block (except for the root) holds between m/2 and m entries. In the example shown in Figure 6.8, m=2; in a real system each node would contain many more entries.

In Figure 6.8 we see seven values being added to the tree, which grows “from the bottom up”:

1. The first value goes in the root
2. Since the root isn't full, the second value goes here too
3. Now it's full - split the block. Since the block doesn't have a parent (it's the root) we add one, which becomes the new root
4. ’4’ fits into one of the leaf nodes where there's room
5. ’5’ doesn't fit, so we split the node. There's room in the parent to hold another pointer
6. ’6’ fits in the leaf node

7. '7' doesn't, so we split the leaf node, but that causes the parent node to overflow, so we split it, and have to add a new parent node which becomes the new root.

6.7 Consistency and Journaling

Unlike in-memory structures, data structures on disk must survive system crashes, whether due to hardware reasons (e.g. power failure) or software failures. This is a different problem than the consistency issues we dealt with for in-memory structures, where data corruption could only occur due to the action of other threads, and could be prevented by the proper use of mutexes and similar mechanisms. Unfortunately there is no mutex which will prevent a system from crashing before the mutex is unlocked, or file system designers would use it liberally. The problem is compounded by the fact that operating systems typically cache reads and writes to increase performance, so that writes to the disk may occur in a much different order than that in which they were issued by the file system code.

In its simplest form the problem is that file system operations often involve writing to multiple disk blocks—for example, moving a file from one directory to another requires writing to blocks in the source and destination directories, while creating a file writes to the block and inode allocation bitmaps, the new inode, the directory block, and the file data block or blocks[13]. If some but not all of these writes occur before a crash, the file system may become *inconsistent*—i.e. in a state not achievable through any legal sequence of file system operations, where some operations may return improper data or cause data loss.

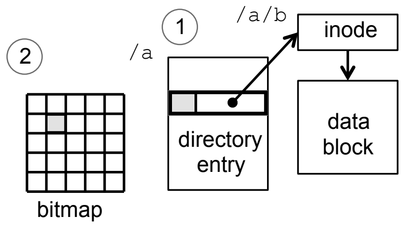


Figure 6.9: File, directory, bitmap

For a particularly vicious example, consider deleting the file /a/b as shown in Figure 6.9, which requires the following actions:

1. Clear the directory entry for /a/b. This is done by marking the entry as unused and writing its block back to the directory.
2. Free the file data block, by clearing the corresponding entry in the block allocation bitmap

[13]These steps ignore inode writes to update file or directory modification times.

This results in two disk blocks being modified and written back to disk; if the blocks are cached and written back at a later point in time they may be written to disk in any order. (this doesn't matter for running programs, as when they access the file system the OS will check cached data before going to disk)

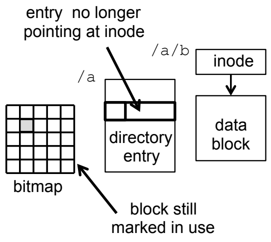


Figure 6.10: Directory block written before crash

If the system crashes (e.g. due to a power failure) after one of these blocks has been written to disk, but not the other, two case are possible:

1. The directory block is written, but not the bitmap. The file is no longer accessible, but the block is still marked as in use. This is a disk space leak (like a memory leak), resulting in a small loss of disk space but no serious problems.
2. The bitmap block is written, but not the directory. Applications are still able to find the file, open it, and write to it, but the block is also available to be allocated to a new file or directory. This is much more serious.

If the same block is now re-allocated for a new file (`/a/c` in this case) we now have two files sharing the same data block, which is obviously a problem. If an application writes to `/a/b` it will also overwrite any data in `/a/c`, and vice versa. If `/a/c` is a directory rather than a file things are even worse - a write to `/a/b` will wipe out directory entries, causing files pointed to by those entries to be lost. (The files themselves won't be erased, but without directory entries pointing to them there won't be any way for a program to access them.)

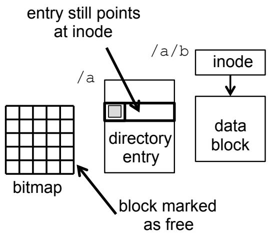


Figure 6.11: Bitmap block written before crash

This can be prevented by writing blocks in a specific order—for instance in this case the directory entry could always be cleared before the block is marked as free, so that in the worst case a crash might cause a few data blocks to become unusable. Unfortunately this is very slow, as these writes must be done synchronously, waiting for each write to complete before issuing the next one.

Fsck / chkdsk: One way to prevent this is to run a disk checking routine every time the system boots after a crash. The dirty flag in the file system superblock was described in the section above; when a machine boots, if the file system is marked dirty, (`fsck`, or `chkdsk` in Windows) is run to repair any problems.

In particular, the Unix file system checker performs the following checks and corrections:

1. Blocks and sizes. Each allocated inode is checked to see that (a) the number of blocks reachable through direct and indirect pointers is consistent with the file size in the inode, (b) all block pointers are within the valid range for the volume, and (c) no blocks are referenced by more than one inode.
2. Pathnames. The directory tree is traversed from the root, and each entry is checked to make sure that it points to a valid inode of the same type (directory / file / device) as indicated in the entry.
3. Connectivity. Verifies that all directory inodes are reachable from the root.
4. Reference counts. Each inode holds a count of how many directory entries (hard links) are pointing to it. This step validates that count against the count determined by traversing the directory tree, and fixes it if necessary.
5. "Cylinder Groups" The block and inode bitmaps are checked for consistency. In particular, are all blocks and inodes reachable from the root marked in use, and all unreachable ones marked free?
6. "Salvage Cylinder Groups" Free inode and block bitmaps are updated to fix any discrepancies.

This is a lot of work, and involves a huge number of disk seeks. On a large volume it can take hours to run. Note that full recovery may involve a lot of manual work; for instance, if fsck finds any files without matching directory entries, it puts them into a `lost+found` directory with numeric names, leaving a human (i.e. you) to figure out what they are and where they belong.

Checking disks at startup worked fine when disks were small, but as they got larger (and seek times didn't get faster) it started taking longer and longer to check a file system after a crash. Uninterruptible power supplies help, but not completely, since many crashes are due to software faults in the operating system. The corruption problem you saw was due to inconsistency in the on-disk file system state. In this example, the free space bitmap did not agree with the directory entry and inode. If the file system can ensure that the on-disk data is always in a consistent state, then it should be possible to prevent losing any data except that being written at the exact moment of the crash.

Performing disk operations synchronously (and carefully ordering them in the code) will prevent inconsistency, but as described above imposes excessive performance costs. Instead a newer generation of file systems, termed *journaling* file systems, has incorporated mechanisms which add additional information which can be used for recovery, allowing caching and efficient use of the disk, while maintaining a consistent on-disk state.

6.8 Journaling

Most modern file systems (NTFS, ext3, ext4, and various others) use *journaling*, a variant of the database technique of *write-ahead logging*. The idea is to keep a log which records the changes that are going to be made to the file system, *before those changes are made*. After an entry is written to the log, the changes can be written back in any order; after they are all written, the section of log recording those changes can be freed.

When recovering from a crash, the OS goes through the log and checks that all the changes recorded there have been performed on the file system itself[14]. Some thought should convince you that if a log entry is written, then the modification is guaranteed to happen, either before or after a crash; if the log entry isn't written completely then the modification never happened. (There are several ways to detect a half-written log entry, including using an explicit end marker or a checksum; we'll just assume that it's possible.)

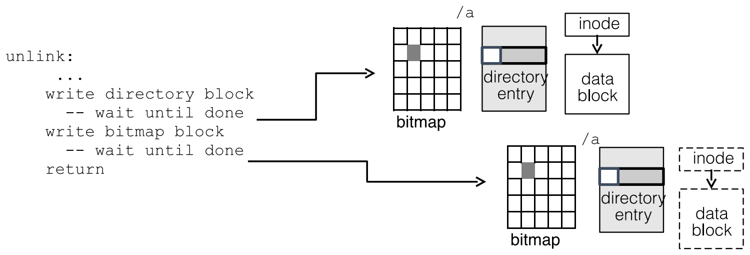
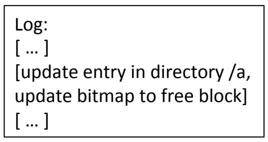
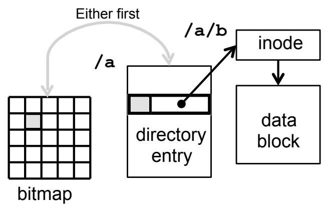


Figure 6.12: Synchronous disk writes for ext2 consistency.

[14]Actually it doesn't check, but rather "replays" all the changes recorded in the log.


Log:
[…]
[update entry in directory /a,
update bitmap to free block]
[…]

Step 1: record action in log



Step 2: write blocks in any order

Ext3 Journaling: The ext3 file system uses physical block logging: each log entry contains a header identifying the disk blocks which are modified (in the example you saw earlier, the bitmap and the directory entry) and a copy of the disk blocks themselves. After a crash the log is replayed by writing each block from the log to the location where it belongs. If a block is written multiple times in the log, it will get overwritten multiple times during replay, and after the last over-write it will have the correct value.

To avoid synchronous journal writes for every file operation, ext3 uses *batch commit*: journal writes are deferred, and multiple writes are combined into a single transaction. The log entries for the entire batch are written to the log in a single sequential write, called a *checkpoint*. In the event of a crash, any modifications since the last checkpoint will be lost, but since checkpoints are performed at least every few seconds, this typically isn't a problem. (If your program needs a guarantee that data is written to a file *right now*, you need to use the `fsync` system call to flush data to disk.)

Ext3 supports three different journaling modes:

- *Journaled*: In this mode, all changes (to file data, directories, inodes and bitmaps) are written to the log before any modifications are made to the main file system.
- *Ordered*: Here, data blocks are flushed to the main file system before a journal entry for any metadata changes (directories, free space bitmaps, inodes) is written to the log, after which the metadata changes may be made in the file system. This provides the same consistency guarantees as journaled mode, but is usually faster.
- *Writeback*: In this mode, metadata changes are always written to the log before being applied to the main file system, but data may be written at any time. It is faster than the other two modes, and will prevent the file system itself from becoming corrupted, but data within a file may be lost.

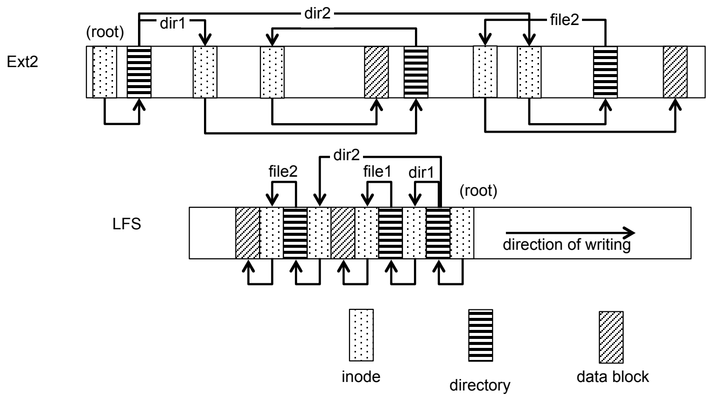


Figure 6.13: Ext2 vs Log-structured file system layout

6.9 Log-Structured File Systems

Log-structured file systems (like LFS in NetBSD, or NetApp WAFL) are an extreme version of a journaled file system: the journal is the entire file system. Data is never over-written; instead a form of copy-on-write is used: modified data is written sequentially to new locations in the log. This gives very high write speeds because all writes (even random ones) are written sequentially to the disk.

Figure 6.13 compares LFS to ext2, showing a simple file system with two directories (dir1, dir2) and two files (/dir1/file1, /dir2/file2). In ext2 the root directory inode is found in a fixed location, and its data blocks do not move after being allocated; in LFS both inode and data blocks move around—as they are modified, the new blocks get written to the head of the log rather than overwriting the old ones. The result can be seen graphically in the figure—in the LFS image, pointers only point to the left, pointing to data that is older than the block holding a pointer. Unlike ext2 there is no fixed location to find the root directory; this is solved by periodically storing its location in a small checkpoint record in a fixed location in the superblock. (This checkpoint is not shown in the figure, and would be the only arrow pointing to the right.)

When a data block is re-written, a new block with a new address is used. This means that the inode (or indirect block) pointing to the data block must be modified, which means that its address changes.

LFS uses a table mapping inodes to locations on disk, which is updated with the new inode address to complete the process; this table is itself

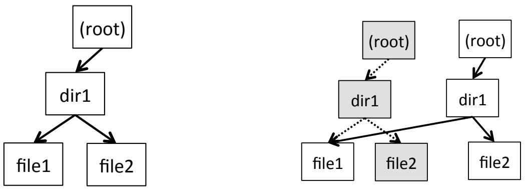


Figure 6.14: WAFL tree before and after update

stored as a file. (The astute reader may wonder why this update doesn't in fact trigger another update to the inode file, leading to an infinite loop. This is solved by buffering blocks in memory before they are written, so that multiple changes can be made.)

In WAFL these changes percolate all the way up through directory entries, directory inodes, etc., to the root of the file system, potentially causing a large number of writes for a small modification. (although they'll still be fairly fast since it's a single sequential write) To avoid this overhead, WAFL buffers a large number of changes before writing to disk; thus although any single write will modify the root directory, only a single modified copy of the root directory has to be written in each batch.

In Figure 6.14 a WAFL directory tree is shown before and after modifying /dir1/file2, with the out-of-date blocks shown in grey. If we keep a pointer to the old root node, then you can access a copy of the file system as it was at that point in time. When the disk fills up these out-of-date blocks are collected by a garbage collection process, and made available for new writes.

One of the advantages of a log-structured file system is the ability to easily keep snapshots of file system state—a pointer to an old version of the inode table or root directory will give you access to a copy of the file system at the point in time corresponding to that version. (e.g. look in your .snapshot directory on `login.ccs.neu.edu` - this data is stored on a NetApp filer using WAFL and its snapshot functionality.)

6.10 Kernel implementation

When applications access files they identify them by file and directory names, or by file descriptors (handles), and reads and writes may be performed in arbitrary lengths and alignments. These requests need to be translated into operations on the on-disk file system, where data is identified by its block number and all reads and writes must be in units of disk blocks.

The primary parts of this task are:

- Path translation - given a list of path components (e.g. “usr”, “local”, “bin”, “program”) perform the directory lookups necessary to find the file or directory named by that list.
- Read and write - translate operations on arbitrary offsets within a file to reads, writes, and allocations of complete disk blocks.

Path translation is a straightforward tree search - starting at the root directory, search for an entry for the first path component, find the location for that file or directory, and repeat until the last component of the list has been found, or an error has occurred. (not counting permissions, there are two possible errors here—either an entry of the path was not found, or a non-final component was found but was a file rather than a directory)

Reading requires finding the blocks which must be read, reading them in, and copying the requested data (which may not be all the data in the blocks, if the request does not start or end on a block boundary) to the appropriate locations in the user buffer.

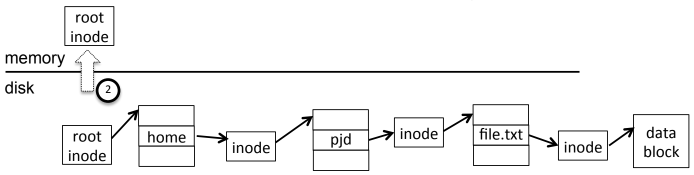
Writing is similar, with added complications: if a write starts in the middle of a block, that block must be read in, modified, and then written back so that existing data is not lost, and if a write extends beyond the end of the file new blocks must be allocated and added to the file.

As an example, to handle the system calls

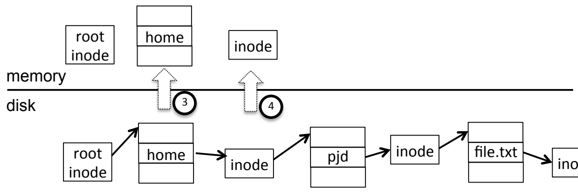
```
fd = open("/home/pjd/file.txt", O_RDONLY)
read(fd, buf, 1024)
```

the kernel has to perform the following steps:

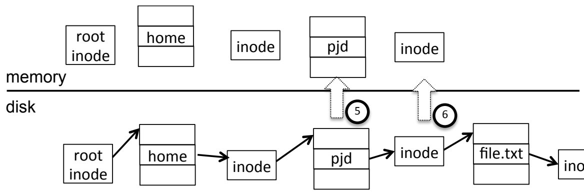
1. Split the string `/home/pjd/file.txt` into parts - `home`, `pjd`, `file.txt`
2. Read the root directory inode to find the location of the root directory data block. (let’s assume it’s a small directory, with one block)



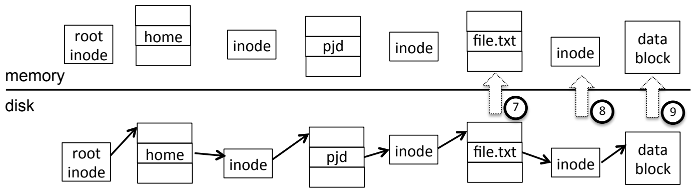
3. Read the root directory data block, search for `"home"`, and find the corresponding inode number
4. Read the inode for the directory `"home"` to get the data block pointer



5. Read the "`home`" directory data block, search for "`pjd`" to get the inode number
6. Read the "`pjd`" directory inode, get the data block pointer



7. Read the "`pjd`" directory block, and find the entry for `file.txt`
8. Read the "`file.txt`" inode and get the first data block pointer
9. Read the data block into the user buffer



Most of this work (steps 2 through 7) is path translation, or the process of traversing the directory tree to find the file itself. In doing this, the OS must handle the following possibilities:

1. The next entry in the path may not exist - the user may have typed `/hme/pjd/file.txt` or `/home/pjd/ffile.txt`
2. An intermediate entry in the path may be a file, rather than a directory - for instance `/home/pjd/file.txt/file.txt`
3. The user may not have permissions to access one of the entries in the path. On the CCIS systems, for instance, if a user other than pjd tries to access `/home/pjd/classes/file.txt`, the OS will notice that `/home/pjd/classes` is protected so that only user `pjd` may access it.

6.11 Caching

Disk accesses are slow, and multiple disk accesses are even slower. If every file operation required multiple disk accesses, your computer would run very slowly. Instead much of the information from the disk is cached in memory in various ways so that it can be used multiple times without going back to disk. Some of these ways are:

File descriptors: When an application opens a file the OS must translate the path to find the file's inode; the inode number and information from that inode can then be saved in a data structure associated with that open file (a *file descriptor* in Unix, or *file handle* in Windows), and freed when the file descriptor is closed.

Translation caching: An OS will typically maintain an in-memory translation cache (the dentry cache in Linux, holding individual directory entries) which holds frequently-used translations, such as root directory entries.

The directory entry cache differs from e.g. a CPU cache in that it holds both normal entries (e.g. directory+name to inode) and negative entries, indicating that directory+name does not exist[15]. If no entry is found the directory is searched, and the results added to the dentry cache.

Block caching: To accelerate reads of frequently-accessed blocks, rather than directly reading from the disk the OS can maintain a *block cache*. Before going to disk the OS checks to see whether a copy of the disk block is already present; if so the data can be copied directly, and if not it is read from disk and inserted into the cache before being returned. When data is modified it can be written to this cache and written back later to the disk.

Among other things, this allows small reads (smaller than a disk block) and small writes to be more efficient. The first small read will cause the block to be read into cache, while following reads from the same block will come from cache. Small writes will modify the same block in cache, and if a block is not flushed immediately to disk, it can be modified multiple times while only resulting in a single write.

Modern OSes like Linux use a combined buffer cache, where virtual memory pages and the file system cache come from the same pool. It's a bit complicated, and is not covered in this class.

[15]To be a bit formal about it, a CPU cache maps a *dense* address space, where every key has a value, while the translation cache maps a *sparse* address space.

6.12 VFS

In order to support multiple file systems such as Ext3, CD-ROMs, and others at the same time, Linux and other Unix variants use an interface called VFS, or the Virtual File System interface. (Windows uses a much different interface with the same purpose) The core of the OS does not know how to interpret individual file systems; instead it knows how to make requests across the VFS interface. Each file system registers an interface with VFS, and the methods in this interface implement the file system by talking to e.g. a disk or a network server.

VFS objects all exist in memory; any association between these structures and data on disk is the responsibility of the file system code. The important objects and methods in this interface are:

`superblock`. Not to be confused with the superblock on disk, this object corresponds to a mounted file system; in particular, the system *mount table* holds pointers to superblock structures. The important field in the superblock object is a pointer to the root directory `inode`.

`inode` - this corresponds to a file or directory. Its methods allow attributes (owner, timestamp, etc.) to be modified; in addition if the object corresponds to a directory, other methods allow creating, deleting, and renaming entries, as well as looking up a string to return a directory entry.

`dentry` - an object corresponding to a directory entry, as described earlier. It holds a name and a pointer to the corresponding `inode` object, and no interesting methods.

`file` - this corresponds to an open file. When it is created there is no associated "real" file; its open method is called with a `dentry` pointing to the file to open.

To open a file the OS will start with the root directory inode (from the superblock object) and call `lookup`, getting back a `dentry` with a pointer to the next directory, etc. When the dentry for the file is found, the OS will create a file object and pass the dentry to the file object's open method.

FUSE (File system in User Space) is a file system type in Linux which does not actually implement a file system itself, but instead forwards VFS requests to a user-space process, and then takes the responses from that process and passes them back to the kernel. This is seen in Figure 6.15, where a read call from the application results in kernel requests through VFS to FUSE, which are forwarded to a user-space file system process.

You will use FUSE to implement a file system in Homework 4, storing the file system in an image file accessed by the file system process.

- `getattr` - return attributes (size, owner, etc.) of a file or directory.
- `readdir` - list a directory
- `mkdir`, `rmdir`, `create`, `unlink` - create and remove directories and files
- `read`, `write` - note that these identify the offset in the file, as the kernel (not the file system) handles file positions.
- `rename` - change a name in a directory entry
- `truncate` - shorten a file
- ... and others, most of which are optional.

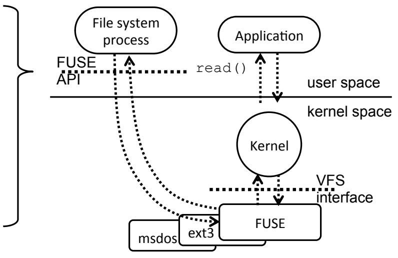


Figure 6.15: FUSE architecture and methods

Like VFS, the FUSE interface consists of a series of methods which you must implement, and if you implement them correctly and return consistent results, the kernel (and applications running on top of it) will see a file system. Unlike VFS, FUSE includes various levels of user-friendly support; we will use it in a mode where all objects are identified by human-readable path strings, rather than dentries and inodes.

6.13 Network File Systems

The file systems discussed so far are local file systems, where data is stored on local disk and is only directly accessible from the computer attached to that disk. Network file systems are used when we want to access to data from multiple machines - for instance, if you log in to a machine in the CCIS lab in room 102, your home directory will be the same on every machine, and is in fact stored on a NetApp file server in a machine room upstairs.

The two network file systems in most common use today are Unix NFS (Network File System) and Windows SMB (also known as CIFS). Each protocol provides operations somewhat similar to those in VFS (quite similar in the case of NFS, as the original VFS was designed for it), allowing the kernel to traverse and list directories, create and delete files, and read and write arbitrary offsets within a file.

The primary differences between the NFS (up through v3 - v4 is more complicated) and SMB are:

- State - NFS is designed to be stateless for reliability. Once you have obtained a file's unique ID (from the directory entry) you can just read from or write to a location in it, without opening the file. Operations are idempotent, which means they can be repeated multiple times without error. (e.g. writing page P to offset x can be repeated, while appending page P to the end of the file can't.) In contrast SMB is connection-oriented, and requires files and directories to be opened before they can be operated on. NFS tolerates server crashes and restarts more gracefully, but does not have some of the connection-related features in SMB such as authentication, described below.

- Identity - NFS acts like a local file system, trusting the client to authenticate users and pass numeric user IDs to the server. SMB handles authentication on the server side - each connection to the server begins with a handshake that authenticates to the server with a specific username, and all operations within that connection are done as that user.

Answers to Review Questions

6.1.1 False - otherwise there would be no subdirectories.

6.1.2 (2) and (4). If the file is deleted (2), all bytes (including the 1000th byte) will cease to exist; the 1000th byte is in the range being overwritten in (4). Renaming leaves the file otherwise unchanged, and modifying bytes 500 through 600 does not affect any other locations in the file.

6.1.3 (2) and (3). Bytes will be read starting at the current position until 'max' bytes have been read or the end of the file is reached, whichever comes first. The data itself is irrelevant, as is the 'buffer' argument. (as long as it points to enough valid memory)

6.1.1 (1) and (3). The OS will not allow non-empty directories to be deleted, as otherwise the files would be lost and their space would not be reclaimed. In addition it must prevent normal user writes to a directory, as user corruption of directory contents might be a security or crash risk.

6.1.2 (2) The mount table is internal to the kernel, and holds the current configuration of where filesystems are mounted, so it can be used when looking up a file. Programs external to the kernel are responsible for knowing where filesystems should be mounted, and doing so. (Typically the startup scripts in Linux read this information from the file `/etc/fstab`.)

6.2.1 True. Disks only support reading and writing in fixed-sized blocks; to modify a smaller region the OS must read a block, modify it, and write it back.

6.2.2 False. In almost all file systems, all blocks must be of the same size.

6.2.3 (1) Since the start of each file is indicated by only a block number (not by e.g. block number plus offset), each file must start at the beginning of a block.

6.2.4 False. It doesn't track free space at all, since being read-only it never needs to allocate space to create new files.

6.2.1 (3) Data blocks contain only data, and are linked via external pointers in the file allocation table.

6.2.2 (1) In ext2 multiple directory entries can point to the same inode. Like MS-DOS, ext2 requires (at least) one pointer per file block; these are in the inode and indirect blocks, while in the MS-DOS file system they are located in the FAT.

Chapter 7

Security

The term computer security covers a number of areas and goals. Most of them fall under the following categories:

- Confidentiality of data. As a user of a computer system, this allows you to prevent others from accessing information which you wish to keep private, such as email or passwords.
- Confidentiality of actions. This lets you prevent others from observing what programs you run and what files or external resources you access.
- Integrity of data. Your data will not be modified or deleted without your permission.
- Integrity of operations. Commands should do what they are supposed to do. For example, when you type ls you should get a directory listing, rather than a script that sends your passwords to a secret website in Russia.
- Availability. A system will not stop running when you need it to be operational.

With the rise of the Internet, security has become a much broader field, much of it related to either networking or the behavior of applications such as web browsers. This chapter will cover operating system features which enable computer security, and which reduce the risks from security flaws in application software; the field of computing security is much larger, however[1].

[1]E.g. see courses such as CS 5770, Software Vulnerabilities and Security, CS 6750, Cryptography and Communication Security, CS 6740, Network Security, or CS 6760, Privacy, Security and Usability

7.1 Protection

Much of security involves protection: deciding whether or not to allow an operation based on a series of rules. The purpose of protection is to ensure the security goals described above, by applying these rules to computing operations, allowing some operations and forbiding others. (This is not sufficient for full security, as seen in the discussion below of software vulnerabilities, but it helps.) These rules are typically based on a simple model, of actors, objects, and actions:

- Actors. These perform the actions. At the lowest level these are almost always processes, but they are typically identified by a text or numeric user ID, which is typically associated with either an actual person or a system service.
- Objects. These are the things which are being protected: usually files or directories, but sometimes processes, special devices, configurations, or other aspects of the system which can be modified.
- Actions. These are performed on objects. The most common actions are read and write, but others can include creating and deleting files, killing a process, or rebooting the system.

The goal of the operating system's protection or *access control* mechanisms is to express and enforce policies which determine whether a particular combination of an actor performing an action on an object is to be allowed or denied.

Identity and Authentication

In a Unix-like operating system, the actual actors are processes, which perform actions by issuing system calls. However specifying rules based on the processes themselves—e.g. process 10 may access file `"/home/pjd"`—will not work, because processes are created dynamically: rules could only be made for processes in existence at that time, and not for ones created in the future.

The solution used in almost all operating systems today is the concept of *user identity*: every process is associated with a user identity (e.g. a user name and ID in Unix) and rules are specified in terms of these identities. In its simplest form each of these identities is a login name associated with an individual person, and rules for that identity are used to permit or restrict access by that person[2]. User identity is inherited through the

[2]Additional identities are typically used for *system accounts*, like the `httpd` user associated with the webserver process on many systems. This allows the same mechanism to be used to grant or restrict access for various system operations.

`fork` system call, so that actions taken by a process either directly or indirectly (through children of that process) are bound by the same rules. Access rules specify actors in terms of these identities, and every process is associated with one of these identities, allowing a fixed mapping from identities to permissions.

In practice this requires a login process, or *authentication*, in which an external user proves that she has the right to take on a certain identity. Thus, the person Jane Smith may be given the right to use the operating system identity named `smith.j`, after *authenticating* that identity to the system by providing the correct password. Authentication is an important part of operating system security, as it forms the link between the higher-level goals of system administration (e.g. Jane is allowed to access *file.txt*, but Joe isn't) and the operating system-level features which implement this control.

These authentication mechanisms are frequently called *factors*; hence the term *multi-factor authentication*, where more than one factor (e.g. password, text message) are used. The Wikipedia entry on "Multi-factor authentication" is a good introduction.

Most authentication mechanisms can be classified as one of three types, based on the type of verification provided by the user:

1. *Something you know*: e.g. a password. This is the most common form of authentication, due to ease of implementation.
2. *Something you have*: like a key to a lock, an RSA SecurID token, etc. More complicated to administer, but more difficult for an adversary to obtain.
3. *Something you are*: often biometric data, such as a fingerprint.

You have undoubtedly used many password-authenticated processes; in addition you may have used other methods such as a SecureID token or fingerprint scanner. There are advantages and disadvantages to each type of authentication; however this class focuses on passwords as they are the most widely used.

Checking passwords

Securely storing and checking passwords is difficult, and various methods have been used over the years. The primary alternatives are as follows.

Unencrypted file: Passwords are stored In an unencrypted file, only accessible to the operating system or a privileged user, and a simple string compare is used to verify a user-entered password. Although in principle this should be secure, the result of any error or failure is catastrophic. (This issue is particularly problematic since people typically re-use passwords across systems.)

Hashed password file: Passwords are put through a one-way cryptographic hash function[3] before being stored. User-entered passwords are hashed by the same algorithm and compared with the stored password hash; if the two match, then the password is correct. Early UNIX systems used this mechanism, and made the password file publicly-readable, as the same file held other information (e.g. mapping between numeric user ID and text username) used by many unprivileged programs. (e.g. the **ls -l** command would need this mapping to show file ownership.) This was considered fairly secure on slow (and especially non-networked) machines, due to the length of time required to crack an individual password by *brute force*—i.e. trying all possible combinations until the correct password was found. However the discovery of *dictionary attacks* changed this, however.

A dictionary attack is based on the idea that in most cases breaking into *any* account is almost as good as breaking a particular account, and takes advantage of the fact that on shared machines (e.g. **login.ccs.neu.edu**) there are a large number of user accounts, increasing the chance of breaking into one of them. The attack consists of calculating the hash for every word in a dictionary[4], and then comparing this list with the hashes in the password file; if *any* of the accounts has a password in this list, the attack succeeds. (a more sophisticated attack, using pre-computed *rainbow tables*[5] is able to crack individual hashed passwords very quickly, as well.)

Password "salt": This is a variation on hashed passwords—when a password is stored, a random string S is chosen (for unknown reasons called the *salt*) and appended to the password before hashing ; the stored value is then [S, hash(S+password)]. Verifying a password is straightforward: the salt value is read from the password file and appended to the input password before it is hashed and compared. This protects against attacks which require pre-computed tables (e.g. dictionary and rainbow table attacks), as now tables would be needed for every possible salt value. (in early usage this was 12 bits long; in modern systems it is 32 bits or more.)

As machines (especially GPUs) become faster and faster, even this method has become less secure over time, as it is becoming feasible for attackers to evaluate billions of possible passwords per second. Counter-measures include making the hash function slower (e.g. running the password

[3]Typically, the password is used as a key to encrypt a known, constant message.

[4]Typically common words, plus personal names, plus variations on those words such as appending a digit.

[5]see `https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture24.pdf`

```
pjd@cs5600-vbox:~$ ls -l /etc/shadow /etc/passwd
-rw-r--r-- 1 root root 1061 Aug 24 2013 /etc/passwd
-rw-r----- 1 root shadow 867 Aug 24 2013 /etc/shadow
pjd@cs5600-vbox:~$ tail -2 /etc/passwd
student:x:1000:1000:A,,,:/home/student:/bin/bash
pjd:x:1001:1001:Peter Desnoyers,,,:/home/pjd:/bin/bash
pjd@cs5600-vbox:~$ sudo tail -2 /etc/shadow
student:$6$JjiTdyS2$cvbtxgVxMwMI5fL0If5Dc90JRuds9yolCKGHc/52ET1tLwksji/
          SN05pksqdwACztcvhIyCDRfAt9lrK133WA/:15935:0:99999:7:::
pjd:$6$wz5.BTqz$RXkmlCnbb0aoA7C67zf2zL7FokmdKLoc51MLdn7jcDe/JMHzs7iePBC
          NEy7O7ZGbVFIl4wTEbi5a8yhhQALnd1:15941:0:99999:7:::
```

Listing 7.1: Password and shadow password files in Linux

through the hash function 5000 times), and protecting the password file against public access, just like was done with early plain-text password files. (this is done in Linux— `/etc/passwd` contains username and UID information, and is publicly readable, while `/etc/shadow` contains password hashes and is protected.

Challenge-response: This is another variation on password checking, although it is typically used over a network rather than for direct login. In this case the server must keep the password in clear text; when a client requests authentication, the server sends a *challenge*—a random string—which the client adds to the password before hashing it and sending the result back to the server. In this way an attacker with access to the network is unable to learn the password, and (if the server never repeats challenges) is unable to replay previous responses.

Review Questions

7.1.1. In Unix, a password is used to determine if you have permission to access a file: *True / False*

7.1.2. Because the passwords in a password file are encrypted, it is safe to make the file publicly readable: *True / False*

Centralized authentication - LDAP and Kerberos

Modern computer systems frequently use centralized password administration: for instance, when you log in to a CCIS workstation your password is not checked locally, but rather against a central authentication server. The most common used mechanisms are LDAP and Kerberos, frequently used as part of Microsoft's Active Directory service. LDAP (lightweight directory access protocol) is a general-purpose directory protocol that can store information about people, machines, and just about anything

```
# pjd, people, ccs.neu.edu
dn: uid=pjd,ou=people,dc=ccs,dc=neu,dc=edu
displayName: Peter J. Desnoyers
cn: pjd
loginShell: /bin/bash
uidNumber: 11415
gidNumber: 65100
sn: Desnoyers
homeDirectory: /home/pjd
mail: pjd@ccs.neu.edu
givenName: Peter
 ...
```

Listing 7.2: Typical CCIS LDAP entry

else that a computer might want to name; an example entry is shown in Figure 7.2.

One of the attributes an LDAP entry can have is a password: a client can log in to the LDAP server by specifying this password, which will be checked by the server. A Linux or other system will use an LDAP server for authentication by attempting to login with the credentials supplied by the user; if this succeeds, then the local login is successful and user information (such as shell and home directory) is retrieved from the server[6].

Kerberos is a more general-purpose authentication mechanism that allows a server to supply unforgeable cryptographically-signed tickets. These allow untrusted machines, like personal computers, to securely access network services, such as a file server while only having to authenticate once, to the Kerberos server; after this initial authentication, the Kerberos tickets can be used for authentication without having to request additional passwords from the user.

Review Questions

7.1.1. LDAP is used for:

a) Storing user information on a central server
b) Storing (and checking) user passwords on a central server
c) Both of the above.

[6]In Linux and some other systems this is handled in practice by the PAM (pluggable authentication module) framework, developed at Sun Microsystems, which specifies one or more authentication sources for the system to try for various events such as login.

7.2 Unix Access Control

Basic security in an operating system is performed by access control: the process of determining whether each OS action will be allowed, based on the actor (determined by information like a user ID), the specific object (e.g. a file), and object permissions. The desired operation can be described by an access control matrix, such as this one:

	file1	file2	dir1	file3
user1	-	read	-	read/write
user2	read	read	read/write	read
user3	read	read	-	-
user4	read/write	read/write	-	-

Table 7.1: Simple access matrix for four users, 3 files and 1 directory

To be more specific, the Unix security model has the following parts:

Actors: Users. User identity (and file ownership) is described by two IDs:

1. User id (`uid`)
2. Group id (`gid`)

In addition there are permissions for *world*—i.e. any user.

Objects: Files and directories.

Actions: On files: *read*, *write*, and *execute* (i.e. run as a program). Directories: *list* (as in `ls`), *traverse* - i.e. accessing the file `/a/b/c` requires traversing the directories `/a` and `/a/b`, and *modify* - i.e. creating, deleting, or renaming files (or directories) within that directory. (note that list, traverse, and modify are encoded as read, execute, and write permissions)

Users may belong to more than one group: as an example, user pjd belongs to groups faculty, cs5600 and sssl, as shown here by the id command:

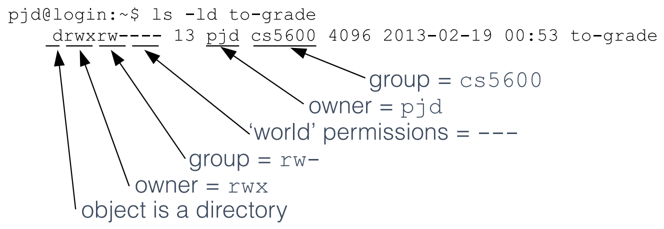
```
pjd@login:~$ id pjd
uid=11415(pjd) gid=65100(faculty) groups=1254(cs5600),1294(sssl),65100(faculty)
```

Listing 7.3: Id command output

Files and directories have an owner and a group, and three sets of permissions: one for the file owner, one for members of its group, and one for world, with the permissions typically encoded in a single string, as shown in Figure 7.1.

Finally, (a) only the owner of a file may change its permissions, and (b) each user may belong to some number of groups. (typically up to 32)

Permissions are interpreted as follows:



```
pjd@login:~$ ls -ld to-grade
    drwxrw---- 13 pjd cs5600 4096 2013-02-19 00:53 to-grade
```

Figure 7.1: Interpreting file ownership and permissions

```
check(process, action, file):
  if process.uid = file.uid:
     if action in file.perm.owner
        allow
     else deny
  if process.gid = file.gid:
     if action in file.perm.group
        allow
     else deny
  if action in file.perm.world:
     allow
  else deny
```

Listing 7.4: File access algorithm

As an example, the access control matrix from earlier: can be encoded in

	file1	file2	dir1	file3
user1	-	read	-	read/write
user2	read	read	read/write	read
user3	read	read	-	-
user4	read/write	read/write	-	-

Table 7.2: Simple access matrix (again)

the set of permissions shown in Figure 7.5.

```
group1 = {user2,user3,user4}
file1: owner = user4, group = group1
              permissions = {owner = 'rw-', group = 'r--', other = '---'}
file2: owner = user4, group = [doesn't matter]
       permissions = {owner = 'rw-', group = 'r--', other = 'r--'}
dir1:  owner = user2, group = [doesn't matter]
              permissions = {owner = 'rw-', group = '---', other = '---'}
group2 = {user1,user2}
file3: owner = user1, group = group2
              permissions = {owner = 'rw-', group = 'r--', other = '---'}
```

Listing 7.5: Permissions for access matrix in Table 7.2

Limitations of Unix permissions

However if we make minor changes to this access control list:

	file1	file2	dir1	file3
user1	—	r–	—	rw-
user2	r–	r–	rw–	r–
user3	rw-	r–	—	rwx
user4	rw-	rw-	—	—

Table 7.3: Complex access matrix (`ls -l` notation used for conciseness)

There are two problems here:

`file1`: Here two users have the highest level of privilege. If `user3` and `user4` are assigned to the same group, and the `file1` owner and group permission are both set to `rw-`, then the only permission left is "world". If that is set to `--` then `user2` will not have the read access specified in the access control matrix; however if it is set to `r-` then `user1` will improperly have access[7].

`file3`: Here there are 4 distinct combinations of permissions, while Unix permissions for a single file can only hold 3 combinations (owner, group, and world).

Review Questions

7.2.1. Given the following file permissions:

```
pjd@login:  ls -l file.txt
-rwxrw-r- 13 pjd faculty 1280 2013-10-19 00:01 file.txt
```

(A) which users can read the file? (B) Which users can write to the file? (C) which users can execute the file?

[7]Although it's possible to achieve this matrix with owner=[`user2 r-`], group=[`(user2,user3,user4) rw-`], and world=[`--`], it doesn't really make sense since the owner can gain write access just by changing permissions on the file.

a) Only user `pjd`
b) Only user `pjd` and any other user in group `faculty`
c) All users

7.2.2. In the following access control matrix:

	file1	file2	dir1	file3
user1	- w -	- - -	r - -	rw -
user2	r - -	r - -	rw -	r - -
user3	r - -	r - -	- - -	r - -
user4	rw -	rw -	rwx	- - -

which of the desired access for which files or directories cannot be implemented using simple Unix permissions?

a) file1 and dir1
b) file1 and file4
c) None: the entire access matrix can be expressed in Unix permissions
d) dir1

Access Control Lists

Access Control Lists (ACLs) are explicit rules granting or denying access to users, and are more powerful than simple permissions. The idea is straightforward: an ACL is a list of rules, where each rule specifies a user or group, an action, and whether to allow or deny permissions.

Using the same example, which could not be encoded in standard Unix permissions:

```
        file1  file2  dir1   file3
user1   ---    r--    ---    rw-
user2   r--    r--    rw-    r--
user3   rw-    r--    ---    rwx
user4   rw-    rw-    ---    ---
```

the desired access to `file1` can be expressed as:

```
file1:      owner = user4, group = {user4,user3}
            owner: rw-, group: rw-, user2: r--, other: ---
file3:      owner = user3, group = {user3, user1}
            owner: rwx, group: rw-, user2: r--, other: ---
```

Access Control List Examples

This example uses OSX access control lists; however, Linux and Windows have similar mechanisms. We start with three user IDs: `pjd`, `guest`, and `joe`.

First a file is created, made readable by all users, and an ACL rule is added denying access to user joe (using the chmod —a command).

```
pjd$ echo 'file 1 contents' > file.1
pjd$ chmod u=r,g=r,o=r file.1
pjd$ chmod +a 'joe deny read' file.1
pjd$ ls -le file.1
-r--r--r--+ 1 pjd wheel 16 Aug 28 14:20 file.1
 0: user:joe deny read
```

The file can now be read by both `pjd` and `guest`, but not `joe`:

```
pjd$ cat file.1
file 1 contents
joe$ cat file.1
cat: file.1: Permission denied
guest$ cat file.1
file 1 contents
```

Now we create a second file, set it owner read-only, and a rule is added giving read access to `joe` but no other user:

```
pjd$ echo 'file 2 contents' > file.2
pjd$ chmod u=r,g=,o= file.2
pjd$ ls -le file.2
-r--------+ 1 pjd wheel 16 Aug 28 14:20 file.2
 0: user:joe allow read
```

Now the file can be read by `pjd` and `joe` but not `guest`:

```
pjd$ cat file.2
file 2 contents
joe$ cat file.2
file 2 contents
guest$ cat file.2
cat: file.2: Permission denied
```

Other Privileged Operations

Most Linux security checking is handled by a combination of the following rules:

- File system permissions
- Signal permissions. A non-root process can only signal processes with the same user ID.
- Super-user. User ID 0 (traditionally named "root") may access any file or signal any process; dangerous kernel operations can only be performed if the current user id is 0.

- Set UID. This allows users to invoke functions with elevated privileges when appropriate.

Most security checking in Linux / Unix is handled by file system permissions. As an example, system utilities need direct access to the disk (e.g. to format a new file system); normal users are prevented from reformatting the disk by the permissions on the special file representing the disk device (e.g. `/dev/sd0`)

Unix signals are primarily used to kill a process, and so are only allowed between processes with the same user ID.

User ID number 0 (traditionally given the username "root") is allowed to bypass all user id-based permissions. In addition, certain system calls (e.g. mount a file system, reboot, install a kernel module) may only be performed by the super-user.

Finally, the setuid permission flag on a file tells the kernel that when the file is executed it should take on the identity of the file's owner, not the user who invoked it. This is a simple mechanism which allows programs to make finer distinctions than the kernel does. For instance the `mount` program is owned by user "root", and marked *setuid*, as under certain circumstances a normal user may be allowed to mount a filesystem (e.g. when it is a removable drive). When the program is run by a normal user, it checks configuration files to see whether the request is authorized; if so it is able to invoke the mount system call as user ID root.

7.3 SELinux

An alternate security mechanism available in Linux is called SELinux, or Security-Enhanced Linux. This is an enhancement to the normal Linux security model, which allows for exceptions to normal Linux rules. As an example, normal file permissions can still deny access to a file, but in cases where permissions allow access, SELinux rules might still forbid it.

SELinux is based on rules about *domains* and *types*. A domain is an execution environment that users run programs in; a set of rules for a domain determine which users can run what programs within that domain. Files have types, and rules determine which domains are able to access which types of files. Finally, users can change domains by running certain programs; when this occurs is again determined by (unsurprisingly) another set of rules.

Rules are loaded into the kernel by a user-space policy process, and file types are determined by an SELinux context associated with the file, stored in file *extended attributes* in the file inode.

As an example, the password file /etc/passwd contains usernames, groups, home directories, but not the hashed passwords themselves, which are in the shadow file, /etc/shadow; the shadow file may be modified when users run the /usr/bin/passwd program:

```
[root@localhost ~]# ls -Z /etc/passwd /etc/shadow /usr/bin/passwd
-rw-r--r-- root  root  system_u:object_r:etc_t          /etc/passwd
-r-------- root  root  system_u:object_r:shadow_t   /etc/shadow
-r-s--x--x root  root  system_u:object_r:passwd_exec_t /usr/bin/passwd
```

One SELinux policy rule states that a user enters the passwd_t domain when executing a file of type passwd_exec_t; another states that processes running in the passwd_t domain have read and write privileges to files of type shadow_t. If SELinux is enabled, then even the superuser will only be able to modify /etc/shadow (and update your password) by executing /usr/bin/passwd or another executable marked passwd_exec_t.

Actual SELinux policies are extremely complex, containing hundreds of rules; if you are interested in finding out more there are a number of tutorials on the Internet, including http://www.centos.org/docs/5/html/Deployment_Guide-en-US/ch-selinux.html

Control of Information Flow

File access control is (somewhat) straightforward for an operating system to provide, as it represents a simple decision. If access is allowed, then the requested operation proceeds without interference; if it is denied, then the request fails completely.

In many cases, however, the desired restrictions are more subtle. Perhaps the earliest published example was a simple computer guessing game; the program would need to access the data to be guessed, while preventing the user from accessing it directly. Simple file permissions would not work, as if the game program were able to access the data file, then other programs (e.g. an editor) would be able to as well, allowing users to cheat.

In general such a problem requires interposing higher-layer software between the user and the protected information. In Unix the *setuid* mechanism allows a user A to run a program as a different user B (e.g. root), by having the executable file owned by user B and setting the *setuid* permission on the file. This can be done in a way that user B has full access to the protected data, allowing the program to access the data on behalf of user A, but only in ways allowed by the program logic.

In modern systems it is more common for such a gatekeeper role to be played by a server with access to the data, which accepts requests from users via messages and makes decisions on what data to reveal. Some examples:

MySQL: This database accepts connections over a TCP socket; users log in and then are able to read and modify those tables they have been given permission to access, regardless of which files the data resides in. The MySQL process runs under a separate user ID, which is the only one able to access the underlying data.

Blackboard: Connections to Blackboard are web sessions, controlled directly by users, and isolation of the data itself is ensured by preventing user access to the system that Blackboard runs on. The application logic enforces a complex set of rules governing what information each user may access; thus an instructor may see all grades, while a student may only see aggregate information (e.g. averages) about other students' grades.

Review Questions

7.3.1. Which of the following most accurately describes the effect of the *setuid* attribute on a file?

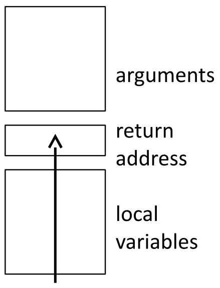
a) It causes the file owner (i.e., user ID) to be updated whenever a process accesses the file
b) It causes the file owner to be updated whenever a process executes the file
c) It causes the user ID of the process to be updated when a process executes the file

7.3.2. Which of the following statements are true? A server-based database such as MySQL:

a) protects the security of its data by putting it in files owned by a separate user ID
b) uses file permissions to prevent access to its data
c) uses application logic on a per-request basis to determine whether to provide access to a data item.

7.4 Attacks — Stack Overflow

In Figure 7.2 you can see a fragment of code that was attacked (among others) by the first piece of Internet malware, the 1988 Morris worm. The target program (`fingerd`) was run with a network connection for its standard input, and used the `gets` function to read a line of input into a buffer on the stack; it would normally return a simple reply based on that input and then exit. `Gets` reads a line from standard input, reading as many bytes as it takes before it finds a newline or reaches end-of-file. The buffer used, located at a lower address on the stack than the return address, was 512 bytes long, but the worm sent a 537-byte single-line message consisting of machine code, ending with a carefully chosen return address pointing at the beginning of the injected code. Since `fingerd` was run as the root user, the result was that when the function returned, the malware code had full control of the machine. In the years since, many lessons have been learned about preventing this sort of attack:



```
f() {
  char buf[512];
  gets(buf);
  ...
}
```

Figure 7.2: Stack frame and buffer overflow

- (application writers) Do not use `gets` or other functions which can overrun a fixed-length buffer. (e.g. `fgets` takes the buffer length as an argument)
- (OS writers) Randomize the location of the stack and libraries each time a program is run, to make it more difficult to guess where an attack should return to.
- (OS writers) Use the NX ("no execute") page table bit on modern processors to prevent code on the stack from being executed.

Unfortunately buffer overflow vulnerabilities are still alive and well, as more sophisticated attacks have been developed to counter these techniques, as long as there is an initial application bug to give access to the stack.[8]

[8]If you are interested in learning more about stack overflows, there is a good tutorial at `http://www.tenouk.com/cncplusplusbufferoverflow.html`

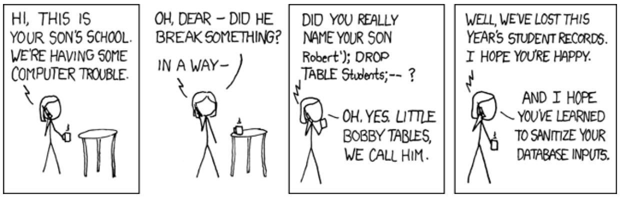

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Figure 7.3: Image by Randall Munroe, from Xkcd: http://xkcd.com/327/, licensed under creative commons non-commercial license 2.5

Security - SQL Injection

What happened to Bobby's school's student database? Let's assume they used code like this Visual Basic fragment, adapted from an example on an MSDN discussion board. ('&' concatenates strings in VB):

```
cmd.commandText = "INSERT INTO Students (Name) VALUES ('" \& studentName \& "');"
cmd.executeNonQuery()
```

So if studentName="Joey Smith", the following SQL command will be executed:

```
‘‘INSERT INTO Students (Name) VALUES('Joey Smith');"
```

But if studentName="Robert'); DROP TABLE Students;–", we get:

```
‘‘INSERT INTO Students (Name) VALUES('Robert'); DROP TABLE Students;--);’’
```

Semicolon (";") is the command separator in SQL, and "" is a comment marker causing the rest of the line to be ignored; after adding 'Robert' to the Students table, the DROP TABLE command will be executed, removing the entire table.

SSL and Connection Security

Secure Sockets Layer (SSL) allows two systems to establish a connection that cannot be intercepted, even by an adversary who observes every packet sent by both systems. Most importantly, it does not require any shared encryption key to be used by both systems[9]. SSL relies on a combination of private- and public-key encryption:

- Private-key encryption uses a private key to encrypt a message, which may then be decrypted using the same private key.

[9]In most cases using a shared private key doesn't solve the problem: before you use it, you have to figure out how to securely communicate the private key.

- Public-key encryption uses two keys: (a) A public key to encrypt the message, and (b) a separate private key which must be used to decrypt it. In one of today's public-key systems, the public key is the product of two large prime numbers, and the private key is the two numbers themselves. The private key can be derived from the public key by factoring it, but for large numbers this is believed to be prohibitively difficult to actually do.

The simplest use of public-key encryption to provide a private connection would be for the two systems to each have public/private key pairs, send each other their public keys, and then encrypt traffic with the other system's public key. Unfortunately, public-key encryption is very computationally expensive, so instead SSL uses the following steps, sometimes called the SSL handshake:

1. The client connects to the server
2. The server sends its public key to the client
3. The client chooses a random number, encrypts it with the public key, and sends it to the server, which then decrypts it.

Client and server both use this random number as the key to a private-key code — all outgoing messages are encrypted using this key, and all incoming messages are decrypted with it. There are additional aspects of the SSL protocol to guard against impersonation and "man-in-the-middle" attacks, which are somewhat more complicated and are not covered here.

Answers to Review Questions

7.1.1 False. The password authorizes you to log in as a specific user ID; file permissions determine whether that user ID has access to a particular file.

7.1.2 False. This used to be the case, but modern hardware can crack encrypted passwords very quickly.

7.1.1 (3) - LDAP handles both user information and passwords.

7.2.1 Read: (3), all users. ("world" permissions are `r-`) Write: (2), owner and group members. Execute: (1), owner `pjd` only.

7.2.2 dir1 has four separate sets of access, which cannot be encoded in three sets of permissions.

7.3.1 (3) the process ID is set to that of the owner of the file

7.3.2 All of these statements are true.

Chapter 8

Hardware Virtualization

Topics covered in this chapter include:

- Applications of virtualization, including server consolidation
- Software emulation, full binary translation, and classical virtualization
- Kernel binary translation, hardware virtualization, and paravirtualization
- Virtual machine migration
- Hosted vs. "bare-metal" hypervisors
- Containers and Docker (even though they don't use HW virtualization)

Hardware virtualization is a technique that allows multiple virtual machines (VMs) to run on the same physical machine, using either pure software or a combination of hardware and software techniques.

Previous chapters have described the differences between *threads*—separate flows of control sharing (almost) all resources such as memory and file descriptors—and *processes*, which are isolated from each other by the operating system, requiring the use of files, pipes, or similar mechanisms to communicate between two processes. A virtual machine is similar to a process, but is designed to run a full operating system and its applications, rather than a single program; communication between VMs is like that between real machines, and must take place over (possibly emulated) networks.

A virtual machine requires a much different interface—while a process runs in unprivileged mode, performing I/O and memory management operations by issuing system calls to the OS kernel, an operating system runs in supervisor mode and uses special instructions and other hardware

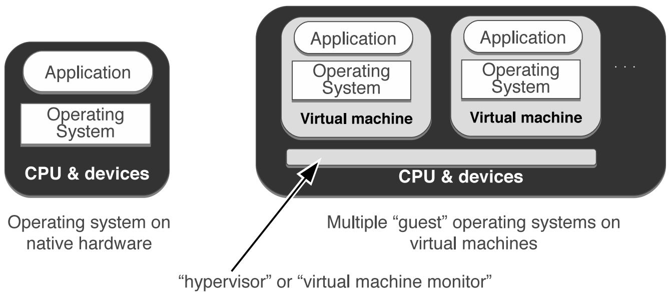


Figure 8.1: Virtual machine architecture

mechanisms to perform its operations. In a virtual machine, as shown in Figure 8.1, these mechanisms are performed by the *hypervisor*[1] which sits "underneath" the operating system.

Running multiple operating system instances on the same physical hardware serves a number of purposes:

Running multiple operating systems: Many applications are tied to a specific OS or even OS version; by using virtual machines it is possible to run instances of these other operating systems and make these applications availabe to a user without requiring extra hardware. (As an example, the laptop I am typing on runs Apple's OS X, but I have a virtual machine running Ubuntu Linux for Linux development.)

Multiple Configurations: Even applications which run on the same operating system may need to run on different machines, rather than just in separate processes. This may be because they require different, incompatible versions of system libraries, or different configuration options. In some cases (e.g. running an old and new version of the same application) they may need different versions of the same configuration files.

Supporting multiple configurations is frequently called *server consolidation*, as in the past an enterprise may have needed to use multiple physical machines to provide these configurations. Frequently the load on each service or configuration was much less than what could be handled by a single machine, and many of these services can instead be deployed as virtual machines on a single physical system.

Security: Many applications (e.g. webservers, databases) require administrative privileges (e.g. root on Unix) for configuration. In the past these applications were typically considered infrastructure services, maintained

[1]Early operating systems were often called *supervisors*; what do you call the program which supervises the supervisor? A hypervisor, I guess.

and configured by system administrators at the request of users. However in many recent cases (e.g. Amazon's Elastic Compute Cloud) the customer is expected to perform all configuration and management, and multiple untrusted customers may share the same physical hardware. Instead of being provided an unprivileged login on a shared machine, each customer is given a virtual machine which they can configure as they wish, with full root or administrative privileges, without posing a threat to customers on other virtual machines.

These uses for virtual machines are artifacts of how applications and operating systems have evolved, and a perfectly-designed OS would no doubt provide the security and manageability benefits described above using operating system-level protections. (This would of course eliminate the need to use any other less perfect operating system.) Virtual machines hold another security advantage, however: they have a smaller *attack surface* than general-purpose operating systems. Operating systems are very large, with millions of lines of code.[2] A *hypervisor*, the piece of software responsible for managing virtual machines, is typically far smaller in comparison, and has only a small number of external interfaces. In theory fewer lines of code (espcially the security-critical code which validates user inputs) means fewer bugs and thus fewer opportunities for security exploits; experience to date seems to support this theory.

Operating system *containers*, such as those used by Docker, provide many of the advantages of virtual machines while using a single operating system. Each container is a set of processes with a *namespace* of process IDs and network connections, and a separate file system tree, and (barring misconfiguration or kernel bugs) is unable to access resources belonging to other containers or to the host OS.

Review Questions

8.0.1. Which of these are reasons why it can be difficult to run multiple network servers on the same machine with a normal operating system?

a) Problems related to assigning separate network addresses to different servers

b) Conflicts between the software and OS requirements for different software packages

[2]The `kernel/` and `mm/` directories in the Linux source add up to about a third of a million lines of code; support for Intel CPUs in `arch/x86/` is another third of a million; the `drivers/` directory is over ten million LOC.

c) The need for administrative privileges to install software packages
d) All of the above

8.1 Implementing Hardware Virtualization

If you are used to running VirtualBox or VMware on your laptop, it may seem like it's just another program, maybe using more memory and CPU than most. But it isn't. To understand why, consider trying to run Linux (the "guest" operating system) on top of a "host" operating system, e.g. Windows. The linux kernel is an executable file, typically found in `/boot/vmlinux`, and could be readily translated into a Windows executable. However if you tried to do this[3] it would crash immediately. Some of the reasons an operating system kernel cannot run as a process are:

Privileged instructions: One of the first things the kernel does on startup is to initialize the virtual memory system, mapping virtual addresses to physical addresses. This configuration requires privileged-mode instructions, which are inaccessible to user-mode applications, as they could be used to bypass operating system protections. The first such instruction executed by the guest OS would cause an exception, killing the process.

Interference: The problem isn't just that the guest OS won't be allowed to modify virtual address mappings. If it actually could modify these mappings, then the underlying host operating system would almost certainly crash, as it assumes that it has complete control over them. The CPU only has a single address translation mechanism, and if two operating systems are going to make use of it, they must either deliberately share access, or it must be virtualized before being used by one or both OS.

Security: Secure isolation between virtual machines, including memory protection, is at least as important as isolation between processes in a normal operating system. But if a guest operating system has direct access to the CPU address translation mechanisms it can easily access physical memory allocated to another virtual machine (or to the host OS itself), bypassing any security mechanisms.

I/O: A process running under Linux or Windows uses *system calls* such as **open** and **read** to access *files*. In contrast, an operating system uses *drivers* to access *physical devices*.

[3]or, actually, running any OS on top of any OS including itself

```
char memory[EMULATED_MEM_SIZE];
int R1, R2, R3, ...;
int PC, SP, CR1, CR2, CR3, ...;
bool S; /* supervisor mode */

while (true):
   instr = memory[PC]
   PC += sizeof(instr)
   case (instr) in:
     "MOV R1 -> R2":
       R2 = R1
     case "JMP <arg>":
       PC = <arg>
     case "STORE Rx, <addr>":
         <paddr> = MMU_translate[<addr>]
                 - on error: emulate page fault
       if <paddr> is real memory:
         memory[paddr] = Rx
       else
         simulate_IO_access(Write, paddr, Rx)
     .... Etc. (for ~1000 more instructions)
```

Figure 8.2: Hypothetical software emulation

In the remainder of this chapter we discuss the following approaches to supporting virtual machines, arranged (roughly) in increasing order of both complexity and performance:

- Software emulation.
- Emulation with binary translation.
- Classical (direct execution + trap-and-emulate) virtualization
- Direct execution + binary translation
- Hardware-assisted virtualization
- Paravirtualization.

Software emulation

The most straightforward way to run a virtual machine is to emulate it entirely in software: in other words, to write a program that behaves exactly like the CPU, memory, and I/O devices of the real machine. The idea is simple: given a complete description of how the CPU behaves, create variables for the registers and a big array for memory, and write a program that repeatedly fetches instructions from the memory array, decodes them, and emulates their operation, much like the sample code in Figure 8.2.

Full software emulation is simple conceptually, although in practice the list of instructions to implement can get long (over 1000 on modern x86 CPUs) and complex. It has one major advantage, portability: once the code to emulate a specific CPU is written, it can be compiled and run on almost any host. This is especially useful in embedded development, where it is often necessary to develop and test software before the CPU (or at least the system incorporating that CPU) is ready to use.

The Java Virtual Machine (JVM) executes bytecode instructions, and can be considered a sort of CPU. Almost all JVMs are based on software emulation, typically with additional performance optimizations.

The primary disadvantage is performance—full software emulation is slow. It can be hundreds of times slower than native execution, making it unsuited for all but a few applications.

Emulation plus binary translation

Software interpreters can be sped up by what Java developers call Just In Time (JIT) compilation, and which CPU emulator developers call Binary Translation. The idea is to translate commonly-used fragments of code into actual machine code, which can usually run far faster than pure emulation. (In part, it eliminates the software-implemented instruction fetch and decode for each instruction, which is a significant overhead.) An example can be seen in Figure 8.3.

Other software systems which use binary translation techniques include:
JVMs: Almost all Java implementations use JIT compiling for performance.
Javascript: Recent browsers (Safari, Firefox, and others) use just-in-time compilation to improve Javascript performance.
Apple Rosetta: This allowed Intel-based Macintosh computers to execute programs compiled for PowerPC.

In other words, the following occurs when a section of binary-translated code is executed:

1. The real CPU registers are loaded from the virtual (i.e., software-emulated) registers
2. The translated instructions are executed
3. The virtual CPU state is updated with results from the real CPU

In most cases the translator will produce more than one instruction per emulated instruction. Memory accesses are particularly tricky, as the generated code must emulate address translation performed by the MMU,

The following code:

```
ADD  R1+R2 -> R2
ADD  R2+R3 -> R3
MUL  2,R3  -> R3
```

might be translated into the following fragment:

```
LOAD Rx <- &emulated_R1
LOAD Ry <- &emulated_R2
LOAD Rz <- &emulated_R3
ADD Rx,Ry -> Ry
ADD Ry,Rz -> Rz
MUL 2,Rz -> Rz
STORE Ry -> &emulated_R2
STORE Rz -> &emulated_R3
RET
```

Figure 8.3: Example of binary translation

and then check to see whether the resulting address is I/O or RAM before performing the operation. In practice it may be possible to arrange emulator memory (using e.g. the `mmap` system call) so that most memory accesses can be performed directly; in this case the overhead for most memory accesses can be reduced to a few instructions which check that an access falls within this typical range.

Trap and Emulate

Even with binary translation, software emulation is still slow—even the best binary translation systems incur a slowdown of 3x to 10x compared to running directly on the same hardware. This is much better than unoptimized software emulation, and may be the best that can be done when emulating one CPU on top of a CPU running a different instruction set. (e.g. running iPhone or Android applications on an Intel-based laptop or desktop, or the Rosetta emulator which Apple used to allow PowerPC applications to run seamlessly on early Intel-based Macintosh systems.)

However In many other cases—such as VirtualBox running on my laptop—the CPU emulated by the virtual machine is the same as the real, physical CPU. In this case we can improve performance greatly by using *direct execution* when possible: executing instructions directly on the physical CPU. The only reason we were emulating instructions in the first place was because the host OS could not allow a virtual machine to directly execute certain privileged instructions, so the goal here is to emulate only these privileged instructions while directly executing all others.

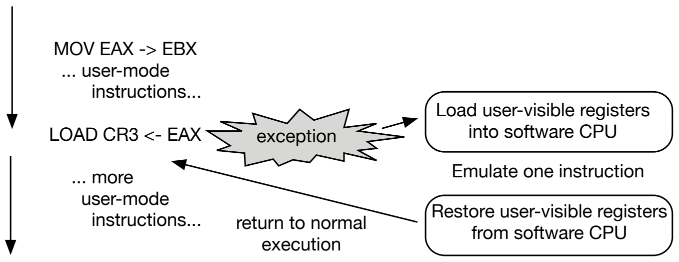

MOV EAX -> EBX
... user-mode instructions...
LOAD CR3 <- EAX
exception
... more user-mode instructions...
return to normal execution
Load user-visible registers into software CPU
Emulate one instruction
Restore user-visible registers from software CPU

Figure 8.4: Trap-and-emulate virtualization

This can be done[4] using a strategy that can be called *trap-and-emulate*. The guest OS is executed directly in user mode; when it executes a privileged instruction it causes an exception which is handled by a specialized OS called a *hypervisor* or *virtual machine monitor*. The hypervisor loads the user-visible CPU registers into the software CPU emulation, runs it for a single instruction, and then returns back to direct execution.

It is interesting to compare a hypervisor running a guest OS (and guest applications) with a traditional operating system running multiple processes. A normal OS virtualizes CPU, memory, and other resources to provide a virtual machine abstraction to each process: each process sees its own memory space and a CPU which can execute user-mode instructions and a special system call instruction. A hypervisor, in contrast, performs a similar task of virtualizing memory and CPU, but provides a virtual machine abstraction which is identical to that of the hardware itself.

Figure 8.4 shows a representation of this trap-and-emulate process. It allows almost all instructions to run directly, at native hardware speed, while the specific instructions which need to be executed in privileged mode (a tiny fraction of all instructions) are emulated. This form of virtualization was originally developed by IBM in the late 60s and early 70s for mainframes, where it continues to be used.

But how does a hypervisor handle exceptions? An operating system relies heavily on exceptions; in fact, just about everything an OS does is part of some exception handler, whether that exception is a system call, a page fault, or a timer or I/O device interrupt. Since the guest OS runs in user mode, exceptions such as system calls or page faults generated by guest applications will be delivered to the hypervisor rather than the guest OS. The solution is for the hypervisor to just emulate the real CPU operation:

[4]on the right processors, as described below

1. Set the emulated supervisor bit to 1
2. (with the emulated CPU) Handle user/supervisor stack switch, pushing registers, and all the other exception-handling mechanisms that take up so many pages in the CPU reference manuals.
3. Return to user mode, load user-visible registers from the emulated CPU, and call the guest OS exception handler.
4. When the guest exception handler returns, set the emulated supervisor bit to 0, restore user registers from the kernel stack, switch to user stack, then jump back to direct execution at the instruction where the exception occurred[5].

How does it know where to find that exception handler? The hardware CPU locates exception handlers via one or more control registers which point to interrupt descriptors which are located in memory. (e.g in Intel-compatible CPUS the `IDTR` register, which points to the *interrupt descriptor table*) These registers may only be accessed in supervisor mode, so the hypervisor is able to virtualize access to these registers and maintain a separate emulated copy for each virtual machine. The real hardware register points to the hypervisor exception handler table, and when a hypervisor exception handler determines that an exception should be forwarded to the guest operating system it looks in the table pointed to by the emulated register to find the guest OS exception handler.

Review questions

8.1.1. Which of the following statements are true?

a) Software emulation uses special-purpose CPU hardware to run virtual machine instructions.
b) Software emulation is slower than native execution.

8.1.2. Trap-and-emulate virtualization:

a) Allows the guest OS to run in user mode, and intercepts exceptions that occur when it executes privileged instructions
b) Allows the guest OS to run in supervisor mode, and intercepts exceptions that occur when it executes privileged instructions
c) Prevents exceptions from occurring while the guest OS is executing

[5]Or the immediately following instruction in the case of *traps* such as system calls.

8.2 Virtualized memory

In a full software emulation, guest virtual addresses were translated into accesses to "fake" physical memory, e.g. the `memory[ ]` array in the example code. However, with trap-and-emulate virtualization, guest applications and most OS code execute directly on the CPU, and virtual addresses are translated to physical addresses in hardware, by the TLB and page tables. This is a problem, because to run multiple virtual machines on a single host, the hypervisor must be able to prevent each from accessing physical memory assigned to the other. Further complicating things, in many cases each guest OS expects physical memory to be in the same place, typically starting at physical address 0. This requires two levels of address translation to get from virtual addresses (used by the guest applications and OS) to real physical memory:

1. Virtual address to "fake" physical address: this translation is maintained by the guest OS
2. "Fake" physical address to real physical address: this translation is maintained by the hypervisor

How does this work on a CPU that only supports one level of virtual-to-physical address translation? By having the hypervisor maintain the real page tables (the ones pointed to by the real CR3) and making sure these tables contain the full virtual → fake physical → real physical translation. This requires two page tables: one pointed to by the emulated CR3 and used by the guest, and one "shadow" table that the real CR3 points to. When a page fault occurs the hypervisor page fault handler uses the following logic:

```
If faulting address is in guest page table:
  1. Look up virtual-to-fake-physical (guest page table) and
       fake physical-to-real-physical (hypervisor) mappings
  2. Install virtual → real physical mapping in
       shadow page table
  3. Return
else (not in guest page table):
  1. invoke guest OS page fault handler
```

Review questions

8.2.1. Address translation for a virtual machine is handled by:

a) Allowing the guest OS to maintain control of the hardware page tables.
b) Having the hypervisor determine the mappings which go in the hardware page tables

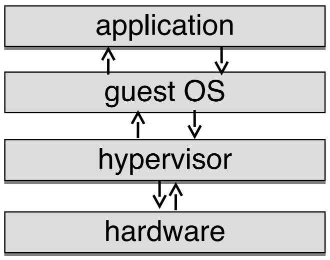


Figure 8.5: "Type 1" Hypervisor — no host OS.

c) Loading the hardware page tables with mappings which combine the guest OS page tables and the hypervisor memory maps.

8.3 Virtualized I/O Devices

Memory-mapped I/O devices are straightforward to emulate in a trap-and-emulate system. When the guest OS maps the physical memory addresses of emulated device registers, the hypervisor leaves the corresponding pages unmapped in the shadow table, so that all read and write accesses will result in a page fault. The hypervisor page fault handler handles faults on these pages specially, calling code that emulates reading from or writing to the emulated I/O device.

8.4 Hosted and "bare-metal" hypervisors

In Figure 8.5 you can see how this works together. Exceptions from user applications (page faults, system calls, etc.) are handled by the hypervisor, which in most cases passes them to the guest OS. Interrupts from I/O devices are passed to drivers in the hypervisor, which may in turn decide that it's time for a virtual hardware device to send an interrupt to a guest OS.

This image describes server systems (like VMware ESX), where the machine boots the hypervisor instead of a regular OS, and does nothing but run virtual machines. But what about a "hosted" system like VirtualBox or VMware Workstation? In particular, how does it run "on top of" a host OS?

The short answer is, it doesn't. When you install VirtualBox it installs a set of drivers, which (like normal hardware drivers) run as part of the kernel, in supervisor mode. When a virtual machine starts running, these

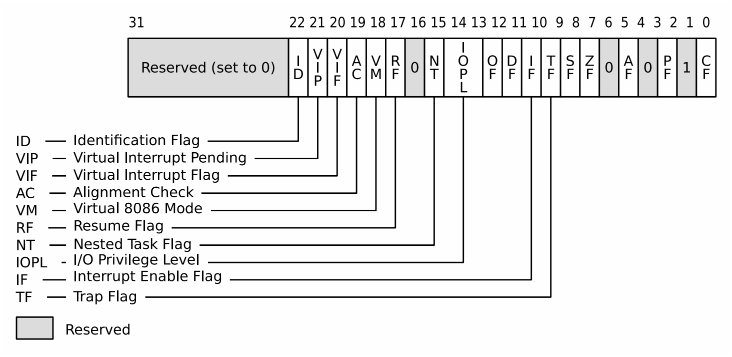


Figure 8.6: Sensitive state in the Intel architecture EFLAGS register

drivers insert themselves under the host operating system, "stealing" exceptions such as page faults and system calls whenever a virtual machine is executing, and forwarding them to the hypervisor. Running on the same system as a host operating system has its advantages, though, as the host OS has drivers for all of its hardware, a file system, display, and other useful interfaces. A hosted hypervisor can take advantage of this, passing I/O requests back to the host OS (via a rather complicated route) to be handled through these standard interfaces, rather than requiring its own drivers for any hardware it uses[6].

8.5 Non-Virtualizable CPUs

There is a minor problem with the classic trap-and-emulate virtualization mechanism as described above: it doesn't work. Or rather, it doesn't work on the machines you want it to work on.

In order to perform classic virtualization to work, every "sensitive" instruction (in other words, one that has to be emulated, like loading CR3 to switch page tables) must trap so that it can be emulated by the hypervisor. Unfortunately, some CPU architectures (in particular, Intel x86 CPUs) have instructions that fail this requirement. For example, on x86 CPUs, a number of instructions which modify supervisor-mode state will silently do nothing when executed in user mode, rather than causing an exception.

As an example, the EFLAGS register as shown in Figure 8.6 contains some commonly-used flags such as carry (CF) and zero (ZF) which it inherited

[6]That's how it works with binary emulation. With hardware virtualization support, the CPU has provisions to allow the "root" environment to continue to run without virtualization, but it's complicated.

from the 16-bit 8086, as well as system flags such as interrupt enable and "IO privilege level", the CPU user/supervisor privilege level. The POPF instruction modifies this register, by loading it with a value popped from the top of the stack. To prevent application code from arbitrarily disabling interrupts or turning on supervisor mode, when POPF is executed in user mode it silently ignores any privileged bits like IOPL and interrupt enable; when kernel code executes POPF in supervisor mode, these bits are loaded with new values. If we try to run the kernel in user mode this instruction will silently do the wrong thing, rather than trapping into the hypervisor.

Instructions like this complicate the task of performing efficient virtualization, but it is still possible, using one of three approaches:

- **Emulation with binary translation:** This is the simplest approach to describe, although rather difficult to implement well. Whenever the guest transitions into supervisor mode (for example, for a system call or an interrupt) the hypervisor emulates all instructions in software, using binary to translation speed up this process, and only resuming direct execution when the guest returns to user mode. This is slower for normal instructions in the kernel, but faster for privileged instructions, as it can translate them once instead of incurring the overhead of trapping and emulating each privileged instruction every time it executes.
- **Hardware virtualization:** Modern x86 CPUs include virtualization extensions, which add a third privilege level more powerful than supervisor mode. The guest runs in normal user and supervisor mode, but certain instructions trap into hypervisor mode for emulation, just as in trap-and-emulate virtualization on a virtualizable CPU. Which instructions? It depends: there are configuration registers providing the hypervisor with a menu of which operations it wants to intercept.
- **Paravirtualization:** rather than providing complete emulation of the hardware platform, the hypervisor provides an OS-like interface so that the operating system can request operations (e.g. address space switch) which would be performed by hardware instructions (e.g. LOAD CR3) on bare hardware. The guest operating system must be modified to use these requests, and so while binary translation and hardware virtualization can run unmodified guest operating systems (e.g. standard Windows installation media) paravirtualization can only support guest operating systems which have been modified for paravirtualization.
  The changes required in a guest OS are actually not that extensive, as most modern operating systems (even Windows) are structured so that they can be (relatively) easily modified to support different

machine types, with hardware-specific portions isolated into a small, replaceable part of the code.

A paravirtualized hypervisor looks sort of like a regular OS: it runs in supervisor mode, with guests running in user mode making requests via system calls using TRAP instructions. Unlike a normal OS, however, these system calls perform hardware-level operations like loading a page table, allocating physical memory, or installing a page fault handler.

Although paravirtualization requires some modifications to the guest OS, in some cases it provides higher performance. As an example, the hypervisor interface can be as efficient as a system call, while hardware virtualization extensions require many cycles to trap, decode, and emulate each instruction.

If you're curious, the Linux code to switch address spaces can be found in the `activate_mm` macro in `arch/x86/include/asm/context.h`. On regular hardware it calls `switch_mm` which executes a `LOAD CR3` instruction; in paravirtualized mode it calls `paravirt_activate_mm` (in `arch/x86/include/asm/paravirt.h`) which invokes a "hypercall" to request the hypervisor to perform the operation.

For years Amazon EC2 used a modified version of the Xen paravirtualized hypervisor, although as hardware virtualization support continues to improve, this remains the case only for a small number of their instance types.

What type of virtualization is fastest? This is actually a hard question—putting something (like virtualization support) into hardware doesn't automatically make it faster. State-of-the-art hardware and software-based (binary translation) hypervisors can have equivalent performance[7], so the choice between them often comes down to features.

Review questions

8.5.1. Which of the following are correct?

a) Paravirtualization requires specialized hardware support
b) Paravirtualization provides a system call-like interface that the guest OS uses to e.g. switch page tables
c) Paravirtualization requires modification to the guest operating system

[7] citation here - Ageson

8.6 Paravirtualized I/O Devices

It is common for hypervisors to have optional drivers (VMware Tools, VirtualBox Guest Additions, etc.) which can be loaded in the guest to improve performance. These typically include paravirtualized drivers for the disk controller and network interface: rather than catching writes to emulated registers, the paravirtualized driver uses a system call-like interface to make I/O requests to the hypervisor. Note that this works because almost all operating systems provide a simple means to load arbitrary kernel-mode drivers for third-party hardware; a paravirtualized device is just another piece of (virtualized) hardware that you need to install a driver for. In contrast, operating systems writers don't typically anticipate the need to support plugging in a different type of CPU.

8.7 Linux Containers and Docker

Running different applications within separate virtual machines provides a number of benefits when compared to running them all on the same unvirtualized operating system:

- Security—if one application is compromised, or is untrusted, the only way for it to attack the other applications (other than via the external network) is by subverting the hypervisor. This is difficult, as they are small and have tended to be quite reliable in practice. (i.e. with few bugs that can be exploited)
- Performance isolation—server-class virtualization systems can enforce resource limits (memory, CPU time, disk and network I/O) which ensure that heavy loads on one application do not negatively impact another.
- Management isolation—each virtual machine has its own file system, administrative (root) account, installed libraries, etc. and can be configured without regard to the dependencies of other applications running in other virtual machines.
- Packaging convenience—a virtual machine image is a convenient and useful format for storing a virtual machine and all of its configuration, as well as for distributing it to others. (like the CS-5600 virtual machine image you received at the beginning of this class)

Note, however, that none of these benefits actually *requires* hardware virtualization[8]. If all of the applications are going to be running on the same operating system, then *Operating System Virtualization* can be used: rather than pretending that a single hardware machine is actually multiple

[8]That is, unless you need to run multiple different operating systems.

virtual machines, we pretend that a single instance of the operating system is actually multiple instances. This approach was first used in FreeBSD *jails* and Solaris *containers*, but the mostly widely known example today is Linux Containers (LXC) and Docker.

LXC allows the creation of isolated process groups: each process in such a group (and any children of those processes) thinks that the group has the entire operating system to itself. This is done via two mechanisms:

Namespaces - in recent Linux versions, any access to the file system, process ID, networking, user or group IDs, or several more obscure system parameters (e.g. hostname) is relative to a *namespace*. In a normal system with no containers there will be a single namespace, visible to all processes. (or at least those that have sufficient permission, in the case of e.g. file system access) However you can also create new namespaces, with a restricted view of the file system (e.g. only able to see a small subtree), with their own process ID space and user names and IDs, and separate network interfaces and addresses. Within a namespace you can have a root user which can perform privileged operations within the namespace, but which has no power or visibility outside of it.

Control groups (cgroups) - these are used to control operating system allocation of resources such as memory, CPU time, or disk and network bandwidth. A cgroup can be associated with a process group, and the process group as a whole will be subject to any limits (e.g. on memory, CPU time, etc.) placed on that cgroup.

The combination of these two features allows the creation of separate *containers*, each with its own file system, network interfaces, etc., and where processes within a container are isolated from those in other containers or in the "base" or root operating system within which these containers were created. Processes in a container interact with the OS kernel in exactly the same way as in a non-containerized system; the only difference is in what they *see*, which is controlled via namespaces, and how their CPU time and I/O are *scheduled*, which is controlled via cgroups. Containers are thus more efficient, as there is no virtualization overhead, and can be created almost as quickly as normal processes.

Since there is still a single operating system kernel, all containers in a system share the same operating system version. Note, however, that they may have entirely different file systems; thus it is quite possible to have both a Red Hat and an Ubuntu distribution running in separate containers on the same machine, although each will be using the kernel of the underlying system.

Docker is based on LXC; however perhaps its main innovation is the way in which it manages container file systems. A Docker container uses a *union file system* to join together multiple file systems—the first one of which is writeable, and with one or more read-only ones “behind” it. To understand the operation of a union file system, consider how the Unix shell finds an executable: it searches each directory in the PATH variable in order, and takes the first version of the file that it finds. Thus if the value of PATH is `/usr/bin:/sbin:/bin`, and you type `ls`, it will search `/usr/bin/ls` (not found), `/sbin/ls` (not found), and then `/bin/ls` (successful). A union file system operates in a similar fashion: on read access to a file (or directory) it will search through each underlying file system in order until it is found. When writing to a file, however, it will write to the first writable file system in the list, providing a form of copy-on-write.

This allows various environments to be *stacked*: e.g. a base file system containing the files from a minimal Linux installation, with additional file systems “on top” of it containing installed versions of various packages one wishes to use, and a writable file system on top for per-instance configuration parameters, application data, etc.

Answers to Review Questions

8.0.1 (4), all of the above.
8.1.1 (1) False: that’s why it’s called software emulation. (2) True: in fact it’s much slower.
8.1.2 (1). privileged instructions will trap in user mode, and the hypervisor emulates them.
8.2.1 (3). The guest cannot be allowed to manipulate hardware page tables, and the hypervisor does not know the guest mappings, but the hypervisor can compose the “fake” guest page tables with hypervisor mappings to provide the correct translation.
8.5.1 (2) and (3). The hardware interface is replaced with “hypercalls”, and the guest OS must be modified to use them instead of direct hardware access.

Appendix A

The CSx600 Micro-Computer

A.1 Overview

The CSx600 is a fictional computer used for examples in this class. The architecture of the system is shown in Figure A.1, below.

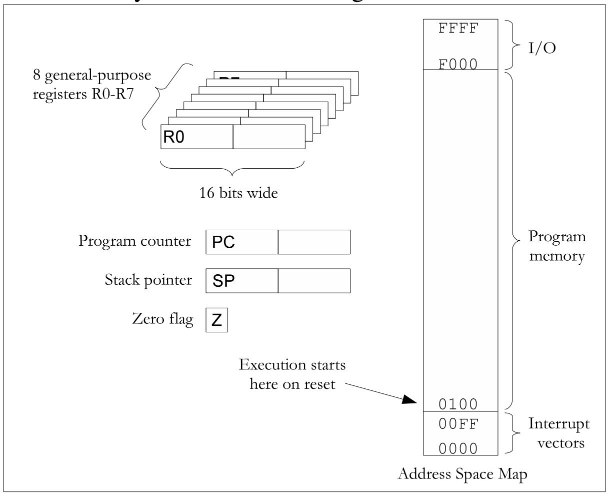


Figure A.1: CSx600 System Architecture

It has 64K bytes of memory, with an address width of 16 bits, and 10 16-bit registers plus two condition flags. Like most modern computers, memory may be accessed as individual bytes or in multi-byte *words*, as shown in Figure A.2; bytes within a word are stored in little-endian fashion as in Intel processors.

Instructions are either a single 16-bit word (2 bytes) for simple instructions, or 4 bytes for instructions which require an additional 16-bit value. They

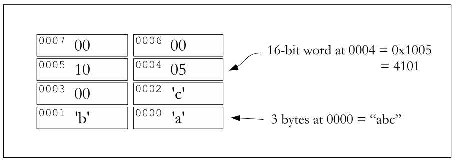


Figure A.2: Byte and 16-bit word addressing of memory. Note that ordering of bytes in words is *little-endian* - the lower address contains the "little" (i.e. least-significant) byte.

are grouped into the following 9 categories:

- Load, store - move data between registers and memory.
- Add, subtract - perform basic arithmetic.
- Push, pop - manipulate the stack.
- Call, return - subroutine invocation.
- Jump - go to another address, either unconditionally or conditionally.
- Interrupt - a subroutine-like mechanism used to implement system calls.

A.2 Calling conventions

The CSx600 CPU uses standard calling conventions, with R7 dedicated as the *base pointer*:

Arguments are promoted to 16-bit values, and pushed onto the stack starting with the last argument; then the CALL instruction is executed.

The *function prologue* pushes the old base pointer onto the stack, copies the stack pointer into the base pointer, and then subtracts *nnn* bytes from the stack pointer where *nnn* is the size in bytes (rounded to a multiple of 2) of the local variables.

The first, second, etc. function arguments may now be addressed as *(bp-4), *(bp-6), etc., while the local variables in turn may be addressed as *(bp+2), *(bp+4), ... Note that these expressions do not change, even though the stack pointer moves up and down while calling subroutines.

The *function epilogue* restores the original stack pointer by (a) copying the base pointer into the stack pointer, and (b) popping the old value of the stack pointer.

The *return value* is placed in R0 before returning.

A.3 Memory-mapped I/O

As shown in Figure A.3, addresses from 0000 to EFFF (hexadecimal) are used for normal memory, but the 4KB range from F000 to FFFF is devoted to I/O. What this means is that when the CPU reads or writes an address in this range, the operation will be directed to one of several input/output devices: the frame buffer (for display), keyboard controller, disk controller, or serial terminal controller. The memory map for this region may be seen in Figure 2. Note that there are large undefined sections in this map; the result of reading or writing these addresses is not defined, but is unlikely to be good.

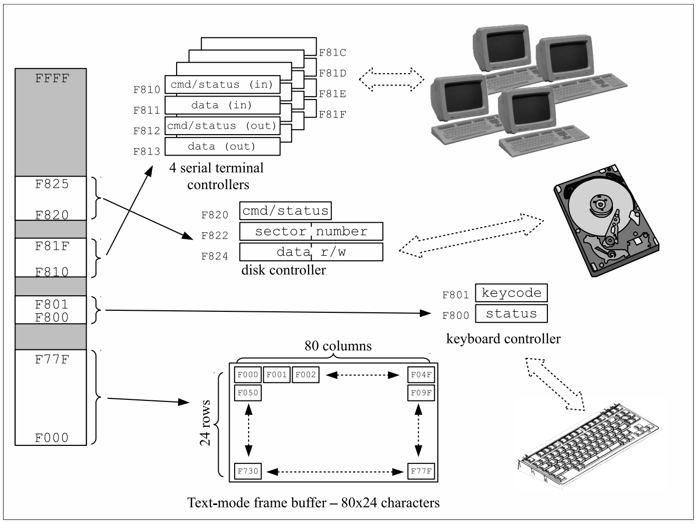
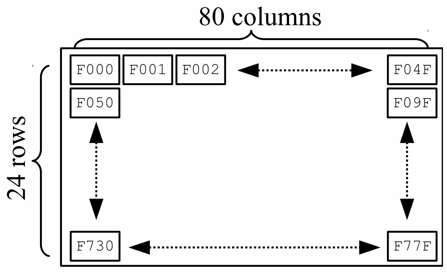
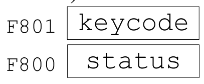


Figure A.3: Memory-mapped I/O devices

Frame buffer (F000 – F77



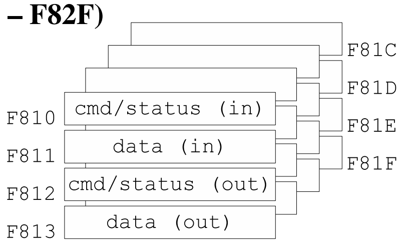
The frame buffer is a contiguous array of 80x24 = 1920 bytes of memory. Each address is mapped to a location on the screen; the byte stored at that address will be displayed in the corresponding screen location. (the VGA screen used by the PC BIOS and e.g. Linux running in console mode works almost identically to this)



Keyboard controller (F800, F801)

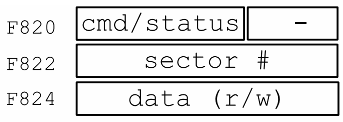
When a key is pressed, the key value is stored in the keycode register (F801) and the status register (F800) is set to 1. After software reads the keycode, it writes a 0 to the status register so that it can detect the next keypress.

Serial port controllers (F820 – F82F)



In order to allow multiple users to access the computer at once, there are four serial ports connected to external terminals. Incoming data from a terminal is received in the same way as for the keyboard controller — the data byte is placed in the data(in) register by the hardware, and status(in) set to 1; the status flag is then set to 0 by software. To send a byte to the terminal, it is written to the data(out) register, and the cmd/status(out) register is set to 1; after the data has been transmitted, the hardware will set the cmd/status(out) register to 0. Note that there are 4 sets of terminal control registers, one for each external terminal.

Disk controller (F820 – FAFF)



The disk controller reads or writes a single 512-byte disk sector at a time. It has a 16-bit register to hold the sector number, and an 8-bit command/status register — a command (read = 0x80, write = 0xC0) is written to the register by software, and a status value (0 = failure, 1 = success) is written to the register by hardware when the command is complete. To read a sector, software sets the sector number register, writes 0x80 to the command register, and waits until the command completes (indicated by the value in the command/status register changing to 01 for success); 512 bytes of data may then be copied from the data register, two bytes at a time. To write a sector, the sector number register is set, the write command (0xC0) is written to the command register, and then 512 bytes of data are written to the data register, after which the controller will write the sector to the disk and set the status register to 1 to signal completion.

A.4 Detailed Instruction Definitions

Load/Store instructions:

These operate on 16-bit words and 8-bit bytes, and have the following addressing modes (which you have no doubt learned in an architecture course):

- absolute - the address used is given as a parameter to the instruction.
- indirect - the address is contained in a register which is identified as a parameter to the instruction.
- indexed - the address is calculated by adding a constant value (supplied as a parameter to the instruction) to an address contained in a register.
- immediate - no address is used, and the value is supplied as a parameter to the instruction.

Traditional assembler syntax separates the source and destination of an operation with a comma — e.g. mov eax,ebx — and which argument is the source and which the destination varies from machine to machine. To eliminate this ambiguity, we will use an arrow symbol to separate source and destination operands in CSx600 assember syntax. The encodings of the instructions are shown in their descriptions below; since this is a fictitious CPU there are no actual numeric values defined for any of the opcodes.

`LOAD.W` $R_{dst} \leftarrow$ `*addr` — load word from constant address

Opcode = LOAD.W1	R_{dst}
Address	

Retrieves 16 bits starting at *addr* and puts the value into R_{dst}.

`LOAD.B` $R_{dst} \leftarrow$ `*addr` — load byte from constant address

Opcode = LOAD.B1	R_{dst}
Address	

Retrieves 8 bits starting at *addr* and puts the value into R_{dst}. The top 8 bits of R_{dst} are set to 0. (note that the remaining load/store instructions each have byte and word variants; descriptions will be combined for brevity.)

`STORE.W` $R_{src} \rightarrow$ `*addr` — store word to constant address
`STORE.B` $R_{src} \rightarrow$ `*addr` — store byte to constant address

Opcode = STORE.W2 / STORE.B2	R_{src}
Address	

Takes 16-bit word (8-bit byte) from R_{src} and stores it into memory starting at addr.

`LOAD.W` $R_{dst} \leftarrow$ `*(`R_{addr}`)` – load word indirect
`LOAD.B` $R_{dst} \leftarrow$ `*(`R_{addr}`)` – load byte indirect

Opcode = LOAD.W2 / LOAD.B2	R_{dst}	R_{addr}

Fetches a 16-bit word (8-bit byte) from memory, starting at the address found in register R_{addr} and stores it in R_{dst}. If only one byte is loaded, the top 8 bits of R_{dst} are set to zero.

`STORE.W` $R_{src} \rightarrow$ `*(`R_{addr}`)` – store word indirect
`STORE.B` $R_{src} \rightarrow$ `*(`R_{addr}`)` – store byte indirect

Opcode = STORE.W3 / STORE.B3	R_{src}	R_{addr}

Takes a 16-bit word (8-bit byte) from R_{src} and stores it into memory starting at the address found in R_{addr}.

`LOAD.W` $R_{dst} \leftarrow$ `*(`R_{addr} `+offset)` – load word indexed
`LOAD.B` $R_{dst} \leftarrow$ `*(`R_{addr} `+offset)` – load byte indexed

Opcode = LOAD.W4 / LOAD.B4	R_{dst}	R_{addr}
Offset		

Loads a word (byte) into R_{dst} from the address found by adding *offset* to the value in R_{addr}.

`STORE.W` $R_{src} \rightarrow$ `*(`R_{addr} `+offset)` – store word indexed
`STORE.B` $R_{src} \rightarrow$ `*(`R_{addr} `+offset)` – store byte indexed

Opcode = STORE.W5 / STORE.B5	R_{src}	R_{addr}
Offset		

Stores a word (byte) from R_{src} into the address found by adding *offset* to the value in R_{addr}.

`LOAD.I` $R_{dst} \leftarrow$ `value` – load immediate value

Opcode = LOAD.W6	R_{dst}
Value	

Load *Value* into R_{dst}.

A.5 Arithmetic Instructions

These instructions perform arithmetic operations on values in registers. The instructions listed here operate on 16-bit words; for completeness there should probably be byte versions of each, but we will not use them in this class.

`ADD` $R_{src} \rightarrow R_{dst}$ – add register to register

Opcode = ADD	R_{src}	R_{dst}

Adds the 16-bit value in R_{src} to the value in R_{dst} and places the result in R_{dst}. Z flag is set iff the result is zero; S flag is set iff the sign bit (most significant bit) of the result is 1.

`ADD value` $\rightarrow R_{dst}$ – add immediate value to register

Opcode = ADDI	R_{dst}
Value	

Adds value to the value in R_{dst} and places the result in R_{dst}. Sets Z and S flag as above.

`SUB` $R_{src} \rightarrow R_{dst}$ – subtract register from register

Opcode = SUB	R_{src}	R_{dst}

Subtracts the 16-bit value in R_{src} from the value in R_{dst} and places the result in R_{dst}. Sets Z and S flag as above.

`SUB.I value` $\rightarrow R_{dst}$ – subtract immediate value from register

Opcode = SUBI	R_{dst}
Value	

Subtracts *value* from the value in R_{dst} and places the result in R_{dst}. Sets Z and S flag as above.

`CMP` R_{src}`,` R_{dst} – Compare registers

Opcode = SUB	R_{src}	R_{dst}

Subtracts the 16-bit value in R_{src} from the value in R_{dst}; discard result but set Z and S flags.

`CMP.I value,` R_{dst} – compare register with immediate value

Opcode = CMPI	R_{dst}
Value	

Subtract *value* from that in R_{dst}; discard result but set Z and S flags.

`MOV` $R_{src} \rightarrow R_{dst}$ – move (copy) register to register

Opcode = MOV	R_{src}	R_{dst}

Copies the contents of R_{src} to R_{dst}. Sets Z and S flag as above.

Stack and Subroutine instructions

These instructions are used for manipulating the stack and calling / returning from subroutines.

`PUSH` R_{src} – push contents of register to stack

Opcode = PUSH	R_{src}

Subtracts 2 from SP, and then stores the contents of R_{src} to the address in SP.

`POP` R_{dst} – pop top of stack into register

Opcode = POP	R_{dst}

Fetches the contents of the memory location indicated by the address in SP, saves it in R_{dst}, and adds 2 to SP.

`ADD_SP #value` – add immediate to stack pointer
`SUB_SP #value` – subtract immediate from stack pointer

Opcode = ADD_SP / SUB_SP
Value

Adds value to SP, thus discarding value/2 elements from the top of the stack. Alternately, subtracts value from SP, reserving value bytes of storage for local variables.

`CALL #addr` – call subroutine

Opcode = CALL
Addr

Pushes return address (the address of the next instruction after CALL) onto the stack, and jumps to addr. I.e.: SP = SP-2, *SP = PC+4, PC = addr.

`RET` – return from subroutine

Opcode = RET

Pops a return address off the stack and jumps to that address.

Branch instructions

These are unconditional and conditional GOTO instructions, used for e.g. loops and 'if' statements.

`JMP #addr` – jump unconditionally to address

Opcode = JMP
Addr

Loads the program counter (PC) with addr, causing execution to skip to that address.

`JMP_Z #addr` – jump if zero flag set
`JMP_NZ #addr` – jump if zero flag clear

Opcode = JMP_Z / JMP_NZ
Addr

If the Z flag is set (not set), jumps to address addr, causing execution to skip to that address. Otherwise does nothing.

`JMP_NEG #addr` – jump if sign flag set (negative)
`JMP_POS #addr` – jump if sign flag clear

Opcode = JMP_NEG / JMP_POS
Addr

If the S flag is set (not set), jumps to address addr, causing execution to skip to that address. Otherwise does nothing.

`INT #nnn` – software interrupt number nnn

Opcode = INT	nnn

Reads the value of interrupt vector nnn — i.e. the 16-bit value at address 2*nnn — and performs a subroutine call to that address.

Appendix B

The CS5600 File System

This chapter provides the following background information:

- The blkdev interface over which the file system is built
- An overview of the FUSE user-space file system toolkit, used in this assignment.
- Debugging and testing advice.

B.1 Blkdev interface

The block device abstraction we use is implemented in the following structure:

```
struct blkdev {
    struct blkdev_ops *ops;
    void *private;
};
#define BLOCK_SIZE 512 /* 512-byte unit for all blkdev addresses */
struct blkdev_ops {
    int (*num_blocks)(struct blkdev *dev);
    int (*read_blk)(struct blkdev * dev, int first_blk, int num_blks, char *buf);
    int (*write_blk(struct blkdev * dev, int first_blk, int num_blks, char *buf);
    void (*close)(struct blkdev *dev);
};
```

The file system in the assignment has changed since the last time the book was updated; the description of the old file system has been removed. The material related to the blkdev interface and FUSE programming has been retained for reference.

This is a common style of operating system structure, which provides the equivalent of a C++ abstract class by using a structure of function pointers for the virtual method table and a void* pointer for any subclass-specific data. Interfaces like this are used so that independently compiled drivers (e.g. network and graphics drivers) to be loaded into the kernel in an OS such as Windows or Linux and then invoked by direct function calls from within the OS.

The methods provided in the blkdev_ops structure are:

- num_blks: the total size of this block device, in 512-byte blocks
- read: read one or more blocks into a buffer. The caller guarantees that 'buf' points to a buffer large enough to hold the amount of data being requested, and that num_blks>0. Legal return values are SUCCESS and E_BADADDR.
- write: write one or more blocks. The caller guarantees that 'buf' points to a buffer holding the amount of data being written, and that num_blks>0. Legal return values are SUCCESS and E_BADADDR.
- close: the destructor method, this closes the blkdev and frees any memory allocated.

The E_BADADDR error is returned if any address in the requested range is illegal—i.e. less than zero or greater than `blkdev->ops->num_blks(blkdev)`.

We will be working with disk image files, rather than actual devices, for ease of running and debugging your code. You may be familiar with image files in the form of .ISO files, which are byte-for-byte copies of a CD-ROM or DVD, and can be read by the same file system code which interacts with a physical disk; in our case we will be writing to the files as well.

B.2 FUSE API

FUSE (File system in USEr space) is a kernel module and library which allow you to implement Linux file systems within a user-space process. For this homework we will use the C interface to the FUSE toolkit to create a program which can read, write, and mount CS5600fs file systems. When you run your working program, it should mount its file system on a normal Linux directory, allowing you to 'cd' into the directory, edit files in it, and otherwise use it as any other file system.

A program which provides a FUSE file system needs to:

1. define file methods — mknod, mkdir, delete, read, write, getdir, ...

2. register those methods with the FUSE library
3. call the FUSE event loop

You will be given code (`misc.c`) which registers your file methods and calls the FUSE event loop; your job is to write the methods which implement the actual file system.

FUSE Data structures

The following data structures are used in the interfaces to the FUSE methods:

path: this is the name of the file or directory a method is being applied to, relative to the mount point. Thus if I mount a FUSE file system at `/home/pjd/my-fuseFS`, then an operation on the file `/home/pjd/my-fuseFS/subdir/filename.txt` will pass `/subdir/filename.txt` to any FUSE methods invoked.

mode: when file permissions need to be specified, they will be passed as a `mode_t` variable: owner, group, and world read/write/execute permissions encoded numerically as described in 'man 2 chmod'[1].

device: several methods have a `dev_t` argument; this can be ignored.

struct stat: described in 'man 2 lstat', this is used to pass information about file attributes (size, owner, modification time, etc.) to and from FUSE methods.

struct fuse_file_info: this gets passed to most of the FUSE methods, but we don't use it.

Error Codes

FUSE methods return error codes in the standard UNIX kernel fashion—positive and zero return values indicate success, while a negative value indicates an error, with the particular negative value used indicating the error type. The error codes you will need to use are:

- EEXIST: a file or directory of that name already exists
- ENOENT: no such file or directory
- EISDIR, ENOTDIR: the operation is invalid because the target is (or is not) a directory
- ENOTEMPTY: directory is not empty (returned by rmdir)

[1] Special files (e.g. /dev files) are also indicated by additional bits in a mode specifier, but we don't implement them in cs5600fs.

- ENOMEM, ENOSPC: operation failed due to lack of memory or disk space
- EOPNOTSUPP: operation not supported.
- EINVAL: invalid arguments

In each case you will return the negative of the value; e.g.:

```
return -ENOENT; /* file not found */
```

The EOPNOTSUPP error code indicates that the operation implemented by a particular method is not supported. Your code should not be returning this error code—if a particular combination of arguments results in a request which will not be handled (see the simplifications listed below) then you should return EINVAL, for invalid arguments.

FUSE Methods

The methods that you will have to implement are:

- `mkdir(path,mode)`: create a directory with the specified mode. Returns success (0), EEXIST, ENOENT or ENOTDIR if the containing directory can't be found or is a file.
- `rmdir(path)`: remove a directory. Returns success, ENOENT, ENOTEMPTY, ENOTDIR.
- `create(path,mode,finfo)`: create a file with the given mode. Ignore the 'finfo' argument. Return values are success, EEXIST, ENOTDIR, or ENOENT.
- unlink(path): remove a file. Returns success, ENOENT, or EISDIR.
- readdir: read a directory, using a rather complicated interface including a callback function. See the sample code for more details. Returns success, ENOENT, ENOTDIR.
- getattr(path, attrs): returns file attributes. (see 'man lstat' for more details of the format used)
- `read(path,buf,len,offset)`: read 'len' bytes starting at offset 'offset' into the buffer pointed to by 'buf'. Returns the number of bytes read on success - this should always be the same as the number requested unless you hit the end of the file. If 'offset' is beyond the end of the file, return 0—this is how UNIX file systems indicate end-of-file. Errors — ENOENT or EISDIR if the file cannot be found or is a directory.
- `write(path,buf,len,offset)`: write 'len' bytes starting at offset 'offset' from the buffer pointed to by 'buf'. Returns the number of bytes written on success - this should always be the same as the

number requested. If 'offset' is greater than the current length of the file, return EINVAL[2]. Errors: ENOENT or EISDIR.

- `truncate(path,offset)`: delete all bytes of a file after 'offset'. If 'offset' is greater than zero, return EINVAL[3]; otherwise delete all data so the file becomes zero-length.
- `rename(path1,path2)`: rename a file or directory. If 'path2' exists, returns EEXISTS. If the two paths are in different directories, return EINVAL.
- `chmod(path,mode)`: change file permissions.
- `utime(path,timebuf)`: change file access and modification times.
- `statfs(path,statvfs)`: returns statistics on a particular file system instance — block size, total/free/used block count, max name length. Always returns success.

Note that in addition to any error codes indicted above in the method descriptions, the 'write', 'mkdir', and 'create' methods can also return ENOSPC, if they are unable to allocate either a file system block or a directory entry.

[2]UNIX file systems support "holes", where you can write to a location beyond the end of the file and the region in the middle is magically filled with zeros. Linux supports plenty of file systems that don't.

[3]UNIX allows truncating a file to a non-zero length, but this is rarely used so we skip it.