

# Overview of Functions of an Operating System

Norman Matloff  
University of California, Davis  
©2001-2007, N. Matloff

March 9, 2007

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	It's Just a Program! . . . . .	4
1.2	What Is an OS for, Anyway? . . . . .	5
<b>2</b>	<b>Application Program Loading</b>	<b>6</b>
2.1	Basic Operations . . . . .	6
2.2	Chains of Programs Calling Programs . . . . .	7
2.3	Static Versus Dynamically Linking . . . . .	8
2.4	Making These Concepts Concrete: Commands You Can Try Yourself . . . . .	8
2.4.1	Mini-Example . . . . .	8
2.4.2	The strace Command . . . . .	8
<b>3</b>	<b>OS Bootup</b>	<b>9</b>
<b>4</b>	<b>Timesharing</b>	<b>10</b>
4.1	Many Processes, Taking Turns . . . . .	10
4.2	Example of OS Code: Linux for Intel CPUs . . . . .	11
4.3	Process States . . . . .	13
4.4	What About Background Jobs? . . . . .	14
4.5	Threads: "Lightweight Processes" . . . . .	14

4.5.1	The Mechanics . . . . .	15
4.5.2	Threads Example . . . . .	16
4.6	Making These Concepts Concrete: Commands You Can Try Yourself . . . . .	17
<b>5</b>	<b>Virtual Memory</b>	<b>19</b>
5.1	Make Sure You Understand the Goals . . . . .	19
5.1.1	Overcome Limitations on Memory Size . . . . .	19
5.1.2	Relieve the Compiler and Linker of Having to Deal with Real Addresses . . . . .	19
5.1.3	Enable Security . . . . .	19
5.2	The Virtual Nature of Addresses . . . . .	20
5.3	Overview of How the Goals Are Achieved . . . . .	21
5.3.1	Overcoming Limitations on Memory Size . . . . .	21
5.3.2	Relieving the Compiler and Linker of Having to Deal with Real Addresses . . . . .	22
5.3.3	Enabling Security . . . . .	22
5.4	Who Does What When? . . . . .	23
5.5	Details on Usage of the Page Table . . . . .	23
5.5.1	Virtual-to-Physical Address Translation, Page Table Lookup . . . . .	23
5.5.2	Layout of the Page Table . . . . .	24
5.5.3	Page Faults . . . . .	26
5.5.4	Access Violations . . . . .	26
5.6	VM and Context Switches . . . . .	28
5.7	Improving Performance—TLBs . . . . .	28
5.8	The Role of Caches in VM Systems . . . . .	29
5.8.1	Addressing . . . . .	29
5.8.2	Hardware Vs. Software . . . . .	29
5.9	Making These Concepts Concrete: Commands You Can Try Yourself . . . . .	30
<b>6</b>	<b>A Bit More on System Calls</b>	<b>30</b>
<b>7</b>	<b>OS File Management</b>	<b>32</b>

*CONTENTS*

*CONTENTS*

**8 To Learn More**

**33**

**A Intel Pentium Architecture**

**33**

## 1 Introduction

### 1.1 It's Just a Program!

First and foremost, it is vital to understand that an **operating system** (OS) is just a program—a very large, very complex program, but still just a program. The OS provides support for the loading and execution of other programs (which we will refer to below as “application programs”), and the OS will set things up so that it has some special privileges which user programs don't have, but in the end, the OS is simply a program.

The source code for Linux, for instance, consists of millions of lines, mainly C but some assembly language for certain low-level machine-specific tasks. The binary for the program typically has a name which is some variant of **vmlinuz**. Look for it in your Linux machine, say in the directory **/boot**.

For example, when your program, say **a.out**,<sup>1</sup> is running, the OS is *not* running. Thus the OS has no power to suspend your program while your program is running—since the OS isn't running! This is a key concept, so let's first make sure what the statement even means.

What does it mean for a program to be “running” anyway? Recall that the CPU is constantly performing its fetch/execute/fetch/execute/... cycle. For each fetch, it fetches whatever instruction the Program Counter (PC) is pointing to. If the PC is currently pointing to an instruction in your program, then your program is running! Each time an instruction of your program executes, the circuitry in the CPU will update the PC, having it point to either the next instruction (the usual case) or an instruction located elsewhere in your program (in the case of jumps).

The point is that the only way your program can stop running is if the PC is changed to point to another program, say the OS. How might this happen? There are only two ways this can occur:

- Your program can *voluntarily* relinquish the CPU to the OS. It does this via a **system call**, which is a call to some function in the operating system which provides some useful service. For example, suppose the C source file from which **a.out** was compiled had a call to **scanf()**. The **scanf()** function is a C library function, which was linked into **a.out**. But **scanf()** itself calls **read()**, a function within the OS. So, when **a.out** reaches the **scanf()** call, that will result in a call to the OS, but after the OS does the read from the keyboard, the OS will return to **a.out**.
- The other possibility is that a hardware interrupt occurs. This is a signal—a physical pulse of current along an **interrupt-request** line in the bus—from some input/output (I/O) device such as the keyboard to the CPU. The circuitry in the CPU is designed to then jump to a place in memory which we designated upon bootup of the machine. This will be a place in the OS, so the OS will now run. The OS will attend to the I/O device, e.g. record the keystroke in the case of the keyboard, and then return to the interrupted program via an IRET instruction.

Note in our keystroke example that the keystroke may not have been made by you. While your program is running, some other user of the machine may hit a key. The interrupt will cause your program to be suspended; the OS will run the device driver for whichever device caused the interrupt—the keyboard, if the person was sitting at the console of the machine, or the network interface card, if the

---

<sup>1</sup> Or it could be a program which you didn't write yourself, say **gcc**.

person was logged in remotely, say via **ssh**—which will record the keystroke in the buffer belonging to that other user; and the OS will execute an **iret** to return from the interrupt, causing your program to resume.

A hardware interrupt can also be generated by the CPU itself, e.g. if your program attempts to divide by 0, or tries to access memory which the hardware finds is off limits to your program. (The latter case describes a seg fault; more on this later.)

So, when your program is running, it is king. The OS has no power to stop it, since it is NOT running. The only ways your program can stop running is if it voluntarily does so or is forced to stop by action occurring at an I/O device.

## 1.2 What Is an OS for, Anyway?

A computer might not even have an OS, say in embedded applications. A computer embedded in a washing machine will just run one program (in ROM). So, there are no files to worry about, no issues of timesharing, etc., and thus no need for an OS.

But on a general-purpose machine, we need an OS to do:

- Input/output device management:

If you recall the example in our unit on I/O, we saw that at the machine level it takes about a dozen lines of code just to read one character from the keyboard. We certainly would not want to have to deal with that in our own programs, but fortunately the OS provides this service for us. Let's see what that means in the case of the keyboard.

First, the OS includes the keyboard device driver, so that we do not need to write our own. Second, the OS provides the **read()** function for us to call in order to pick up the characters the keyboard driver has collected for us. Suppose for instance that you have a **scanf()** call or a **cin** statement in your program. When the user types some characters, each keystroke will cause an interrupt.<sup>2</sup> When the interrupt occurs, the keyboard driver will add the character to an array, typically referred to as a **buffer**. Your program's **scanf()** call or **cin** statement will include to a call to **read()**, a function in the OS. That function will then pick up the characters in the keyboard buffer, parse them according to whatever kind of read format you asked for, and return the result to your program.

Again, the central point here is that the OS is doing a big service for you, in that it is alleviating you of the burden of writing all this code.

- File management:

For example, suppose you are writing a program to write a new file. You wouldn't want to have to deal with the details of knowing where the empty space is on the disk, choosing some of it for the file, etc. Again, the OS does this for you.

- Process management:

---

<sup>2</sup>Actually two interrupts, one for key press and one for release, but let's say one here, for simplicity.

## 2 APPLICATION PROGRAM LOADING

When you want to run a program, the OS loads it into memory for you and starts its execution.

The OS enables **timesharing**, in which many application programs seem to be running simultaneously but are actually “taking turns,” *by coordinating with hardware operations*.

- Memory management:

The OS must keep track of which parts of memory are currently in use, so that it knows what areas it can load a program to when the program is requested for execution.

The OS also enables **virtual memory** operations for application programs, which both allows flexible use of memory and enforces security, again by *by coordinating with hardware operations*.

How the OS does these things is explained in the following sections. We will model the discussion after a Unix system, but the description here applies to most modern OSs. It is assumed here that the reader is familiar with basic Unix commands; a Unix tutorial is available at <http://heather.cs.ucdavis.edu/~matloff/unix.html>

## 2 Application Program Loading

### 2.1 Basic Operations

Suppose you have just compiled a program, producing, say for a Linux system, an executable file **a.out**. (And the following would apply equally well to **gcc** or any other program.) To run it, you type

```
% a.out
```

For the time being, assume a very simple machine/OS combination, with no **virtual memory**. Here is what will occur:

- During your typing of the command, the shell is running, say **tsh** or **bash**. Again, the shell is just a program (one which could be assigned as a homework problem in a course). It prints out the ‘%’ prompt using **printf()**,<sup>3</sup> and enters a loop in which it reads the command you type using **scanf()**.<sup>4</sup>
- The shell will then make a system call, **execve()**,<sup>5</sup> asking the OS to run **a.out**. The OS is now running.
- The OS will look in its disk directory, to determine where on disk the file **a.out** is. It will read the **header**, i.e. beginning section of **a.out**, which contains information on the size of the program, a list of the data sections used by the program, and so on.<sup>6</sup>

---

<sup>3</sup>Most shells are written in C.

<sup>4</sup>Note that there will be an interrupt each time you type a character. The OS will store these characters until you hit Enter, in which case the OS will finally “wake” the shell program, whose **scanf()** will now finally execute. See also Sleep and Run states later in this unit.

<sup>5</sup>It will also call another system call, **fork()**, but we will not go into that here.

<sup>6</sup>You can read that information yourself, using the **readelf** command in Linux.

- The OS will check its memory-allocation table (this is just an array in the OS) to find an unused region or regions of memory large enough for **a.out**. (Note that the OS has loaded all currently-active programs in memory up to now, just like it is loading **a.out** now, so it knows exactly which parts of memory are in use and which are free.) We need space both for **a.out**'s instructions, in the **.text** section, its static data (in the **.data** section for uninitialized variables, and in the **.bss** section for initialized variables), as well as for stack space and the **heap**.

The heap is used for calls to **malloc()** or for invocations of the C++ operator **new**. With the GCC compiler, these are one and the same, since the internal implementation of **new** calls **malloc()**. There will be a data structure there which keeps track of which parts of the heap are in use and which are free. Of course, they are initially all free. Each time we need new memory, we use parts of the heap, and they will be marked as in use. Calling **free()** or invoking **delete** will result in the space being marked as free again. All the marking is done by **malloc()**.

- The OS will then load **a.out** (including text and data) into those regions of memory, and will update its memory-allocation table accordingly.
- The OS will check a certain section of the **a.out** file, in which the linker previously recorded **a.out**'s **entry point**, i.e. the instruction in **a.out** at which execution is to begin. (This for example is `_start` in our previous assembly language examples.)
- The OS is now ready to initiate execution of **a.out**. It will set the stack pointer to point to the place it had chosen earlier for **a.out**'s stack. (The OS will save its own register values, including stack pointer value, beforehand.) Then it will place any command-line arguments on **a.out**'s stack, and then actually start **a.out**, say by executing a **JMP** instruction (or equivalent) to jump to **a.out**'s entry point.
- The **a.out** program is now running!

## 2.2 Chains of Programs Calling Programs

Note that **a.out** itself may call **execve()** and start other programs running. We mentioned above that the shell does this, and so for example does **gcc**.<sup>7</sup> It first runs the **cpp** C preprocessor (which translates **#include**, **#define** etc. from your C source file). Then it runs **cc1**, which is the “real” compiler (**gcc** itself is just a manager that runs the various components, as you can see now); this produces an assembly language file.<sup>8</sup> Then **gcc** runs **as**, the assembler, to produce a **.o** machine code file. The latter must be linked with some code, e.g. **/usr/lib/crt1.o** and other files which set up the **main()** structure, e.g. access to the **argv** command-line arguments, and also linked to the C library, **/lib/libc.so.6**; so, **gcc** runs the linker, **ld**. In all these cases, **gcc** starts these programs by calling **execve()**.

<sup>7</sup>Remember, a compiler is just a program too. It is a very large and complex program, but in principle no different from the programs you write.

<sup>8</sup>You can view this file by running **gcc** with its **-S** option. This is often handy if you are going to write an assembly-language subroutine to be called by your C code, so that you can see how the compiler will deal with the parameters in the call to the subroutine.

## 2.3 Static Versus Dynamically Linking

Say your C code includes a call to `printf()`. Even though someone else wrote `printf()`, when you link it in to your program (which GCC does for you when it calls LD), it becomes part of your program. The question is when this occurs.

With **static** linking, LD will put `printf()`'s machine code into your **a.out** executable. However, since most programs make use of the C library, they would collectively take up a lot of disk space if static linking were used.

So, the default is that GCC and LD would set up **dynamic** linking for `printf()` in your program. In your **a.out** file, there will be a message saying that your code calls `printf()`, and that when your program is loaded into memory and is run, `printf()` should be linked in to it at that time.

## 2.4 Making These Concepts Concrete: Commands You Can Try Yourself

### 2.4.1 Mini-Example

Below is a simple illustration of `execve()` in action. The program **run.c** makes this system call to get the program **greeting.c** running. Assume the executable files are named **greeting** and **run**.

```

1 // greeting.c
2
3 main()
4 { printf("hello\n"); }

1 // run.c
2
3 main()
4 { char **a[2], // arguments to "greeting" (none here; see man page)
5     **p[2]; // environment for "greeting" (none here)
6
7     a[0] = ""; a[1] = 0;
8     p[0] = ""; p[1] = 0;
9     execve("./greeting",a,p); }

1 % run
2 hello
```

Note that the shell will do something like this too, when you request that your program **a.out** be run.

### 2.4.2 The `strace` Command

The Unix **strace** command will report which system calls your program makes. Place it before your program name on the command line, e.g.

```
% strace a.out
```

### 3 OS BOOTUP

The output will be rather overwhelming, but if you sift through it<sup>9</sup> You will see that a call `execve()` is made by the shell to launch `a.out`, as well as the various systems call made by `a.out`.

## 3 OS Bootup

As was explained earlier, when we wish to run an application program, the OS loads the program into memory. But wait! The OS is a program too, so how does *it* get loaded into memory and begin execution? The answer is that another program, the **boot loader**, will load the OS into memory. But then how does the boot loader get loaded in memory?! It sounds like we have an endless, “chicken and egg” problem. The resolution is that the boot loader is permanently in memory, in ROM, so it doesn’t have to be loaded.

The process by which the OS is loaded into memory is called **bootup**. We will explain this using the Intel Pentium architecture for illustration.<sup>10</sup>

The circuitry in the CPU hardware will be designed so that upon powerup the Program Counter (PC) is initialized to some specific value, `0xfffff0` in the case of Intel CPUs. And those who **fabricate** the computer (i.e. who put together the CPU, memory, bus, etc. to form a complete system) will include a small ROM at that same address, again `0xfffff0` in the Intel case. The contents of the ROM include the boot loader program. So, immediately after powerup, the boot loader program is running!

The goal of the boot loader is to load in the OS from disk to memory. In the simple form, the boot loader reads a specified area of the disk, copies the contents there—which will be part of the OS—to some section of memory, and then finally executes a `JMP` instruction (or equivalent) to that section of memory—so that now the OS is running. The OS then reads the rest of itself into memory.

For the sake of concreteness, let’s look more closely at the Intel case. The program in ROM here is the BIOS, the Basic I/O System. It contains parts of the device drivers for that machine,<sup>11</sup> and also contains the first-stage boot loader program.

The boot loader program in the BIOS has been written to read some information, to be described now, from the first physical sector of the disk.<sup>12</sup> That sector is called the Master Boot Record (MBR).

There will be one or more **partitions** for the disk. If, say, the disk has 1,000 cylinders, then for example we may have the first 200 as one partition and the remaining 800 as the second partition.<sup>13</sup> These simply divide up parts of the disk for different purposes. A typical example would be that there are two partitions, one for Windows and the other for Linux. Your machine was probably shipped with the entire disk consisting of just one partition, but you can split it into several partitions to accommodate different OSs.

These partitions are defined in the **partition table** in the MBR. Note that there is no physical separation from one partition to the next; the boundary between one partition and the next is only defined by the partition

---

<sup>9</sup>I recommend running the Unix **script** command before you run **strace**. It will keep a record of everything that appears on your screen. After you are done running **strace**, type `exit` to leave the shell created by **script**. Then the record of your session is in a file named **typescript**. You can then browse through it using your favorite text editor.

<sup>10</sup>An overview of that architecture is presented in the Appendix.

<sup>11</sup>These may or may not be used, depending on the OS. Windows uses them, but Linux doesn’t.

<sup>12</sup>The boot device could be something else instead of a hard drive, such as a floppy or a CD-ROM. This is set in the BIOS, with a priority ordering of which device to try to boot from first.

<sup>13</sup>You should review the material on disks in our unit on I/O.

## 4 TIMESHARING

table. Exactly one of the primary partitions will be marked in the table as **active**, meaning bootable.

The MBR also contains the second-stage boot loader program. The boot loader program in the BIOS will read this second-stage code into memory and then jump to it, so that that program is now running. That program reads the partition table, to determine which partition is the active one. It then goes to the first sector in that partition, reads in the third-stage boot loader program into memory from there, and then jumps to that program.

Now if the machine had originally been shipped with Windows installed, the second-stage code in the MBR had been written to then load into memory the third-stage code from the Windows partition. If on the other hand the machine had been shipped with Linux or some other OS installed, there would be a partition for that OS; the code in the MBR would have been written accordingly, and that OS would now be loaded into memory.

Many people who use Linux retain both Windows and Linux on their hard drives, and have a **dual-boot** setup. They start with a Windows machine, but then install Linux as well, so both OSs are on the machine. As part of the process of installing Linux, the old MBR is copied elsewhere, and a program named LILO (Linux Loader) is written into the MBR.<sup>14</sup> In other words, LILO will be the second-stage boot loader. LILO will ask the user whether he/she wants to boot Linux or Windows, and then go to the corresponding partition to load and execute the third-stage boot loader which is there.<sup>15</sup>

In any case, after the OS is loaded into memory by the third-stage code, that code will perform a jump to the OS, so the OS is running.

## 4 Timesharing

### 4.1 Many Processes, Taking Turns

Suppose you and someone else are both using the computer **pc12** in our lab, one of you at the console and the other logged in remotely. Suppose further that the other person's program will run for five hours! You don't want to wait five hours for the other person's program to end. So, the OS arranges things so that the two programs will take turns running. It won't be visible to you, but that is what happens.

**Timesharing** involves having several programs running in what appears to be a simultaneous manner.<sup>16</sup> If the system has only one CPU (for simplicity, we will exclude the case of multiprocessor systems in this discussion), this simultaneity is of course only an illusion, since only one program can run at any given time, but it is a worthwhile illusion, as we will see.

First of all, how is this illusion attained? The answer is that we have the programs all take turns running, with each turn—called a **quantum** or **timeslice**—being of very short duration, for example 50 milliseconds. Say we have four programs, **u**, **v**, **x** and **y**, running currently. What will happen is that first **u** runs for 50

<sup>14</sup>A newer, now more popular alternative to LILO is GRUB. But the principle is the same for both, so we will just refer to LILO here for simplicity.

<sup>15</sup>The partition does not have to be the active one. The term *active* applies only from the point of view of the Windows second-stage boot loader which had originally been installed in the MBR.

<sup>16</sup>These programs could be from different users or the same user; it doesn't matter.

milliseconds, then **u** is suspended and **v** runs for 50 milliseconds, then **v** is suspended and **x** runs for 50 milliseconds, and so on. After **y** gets its turn, then **u** gets a second turn, etc. Since the turn-switching, formally known as **context-switching**,<sup>17</sup> is happening so fast (every 50 milliseconds), it appears to us humans that each program is running continuously (though at one-fourth speed), rather than on and off, on and off, etc.<sup>18</sup>

But how can the OS enforce these quanta? For example, how can the OS force the program **u** above to stop after 50 milliseconds? As discussed earlier, the answer is, “It can’t! The OS is dead while **u** is running.” Instead, the turns are implemented via a timing device, which emits a hardware interrupt at the proper time. For example, we could set the timer to emit an interrupt every 50 milliseconds. We would write a timer device driver, and incorporate it into the OS.

We will make such an assumption here. However, what is more common is to have the timer interrupt more frequently than the desired quantum size. On a PC, the 8253 timer interrupts 100 times per second. Every sixth interrupt, the Linux OS will perform a context switch. That results in a quantum size of 60 milliseconds. But this can be changed, simply by changing the count of interrupts needed to trigger a context switch.

The timer device driver saves all **u**’s current register values, including its PC value and the value in its EFLAGS register. Later, when **u**’s next turn comes, those values will be restored, and **u** will resume execution as if nothing ever happened. For now, though, the OS routine will restore **v**’s previously-saved register values, making sure to restore the PC value last of all. That last action forces a jump from the OS to **v**, right at the spot in **v** where **v** was suspended at the end of its last quantum. (Again, the CPU just “minds its own business,” and does not “know” that one program, the OS, has handed over control to another, **v**; the CPU just keeps performing its fetch/execute cycle, fetching whatever the PC points to, oblivious to which process is running.)

At any given time, there are many different **processes** in memory. These are instances of executions of programs. If for instance there are three users running the **gcc** C compiler right now on a given machine, here one program corresponds to three processes.

## 4.2 Example of OS Code: Linux for Intel CPUs

Here is a bit about how the context switch is done in Linux, in the version for Intel machines.<sup>19</sup>

The OS maintains a **process table**, which is simply an array of **structs**, one for each process. The **struct** for a process is called the Task State Segment (TSS) for that process, and stores various pieces of information about that process, such as the register values which the program had at the time its last turn ended. Remember, these need to be stored somewhere so that we can restore them at the program’s next turn; here’s where we store them.

As an example of the operations performed, and to show you concretely that the OS is indeed really a program with real code, here is a typical excerpt of code:

<sup>17</sup>We are switching from the “context” of one program to another.

<sup>18</sup>Think of a light bulb turning on and off extremely rapidly, with half the time on and half off. If it is blinking rapidly enough, you won’t see it go on and off. You’ll simply see it as shining steadily at half brightness.

<sup>19</sup>This was as of Linux kernel version 2.3. I have slightly modified some of this section for the sake of simplicity.

```

1  pushl %esi
2  pushl %edi
3  pushl %ebp
4  movl %esp, 532(%ebx)
5  ...
6  movl 532(%ecx), %esp
7  ...
8  popl %ebp
9  popl %edi
10 popl %esi
11 ...
12 iret

```

This code is the ISR for the timer. Upon entry, **u**'s turn has just ended, having been interrupted by the timer. In code not shown here, the OS has pointed the registers EBX and ECX to the TSSs of the process whose turn just ended, **u**, and the process to which we will give the next turn, say **v**.

By the way, how does the OS know that the process whose turn just ended was **u**'s? The answer is that whenever the OS gives a process a turn, it records which process it was. In Linux this information is stored at address 0xffffe000, and the macro **GET\_CURRENT()** is used to fetch it and place it into a register specified by the programmer; in our situation here, the register would be EBX. So the OS knows where to point EBX. The OS will make a decision as to which process to next give a turn to, and point ECX to it.

Here is what that code does. The source code for Linux includes a variable **tss**, which is the TSS **struct** for the current process. In that struct is a field named **esp**. So, **tss.esp** contains the previously-stored value of ESP, the stack pointer, for this process; this field happens to be located 532 bytes past the beginning of the TSS.<sup>20</sup>

Now, upon entry to the above OS code, ESP is still pointing to **u**'s stack, so the three PUSH instructions save **u**'s values of the ESI, EDI and EBP registers on **u**'s own stack.<sup>21</sup> The other register values of **u** must be saved too, including its value of ESP. The latter is done by the MOV, which copies the current ESP value, i.e. **u**'s ESP value, to **tss.esp** in **u**'s TSS. Other register saving is similar, though not shown here.

Now the OS must prepare to start **v**'s next turn. Thus **v**'s previously-saved register values must be restored to the registers. To understand how that is done, you must keep in mind that that same code we see above had been executed when **v**'s last turn ended. Thus **v**'s value of ESP is in **tss.esp** of its TSS, and the second MOV we see above copies that value to ESP. So, now we are using **v**'s stack.

Next, note similarly that at the end of **v**'s last turn, its values of ESI, EDI and EBP were pushed onto its stack, and of course they are still there. So, we just pop them off, and back into the registers, which is what those three POP instructions do.

Finally, what actually gets **v**'s new turn running? To answer this, note that the mechanism which made **v**'s last turn end was a hardware interrupt from the timer. At that time, the values of the Flags Register, CS and PC were pushed onto the stack. Now, the IRET instruction you see here pops all that stuff back into the corresponding registers. Note only does that restore registers, but since **v**'s old PC value is restored to the PC register, **v** is now running!

<sup>20</sup>In the C source code, this field is referred to as **tss.esp**, but in assembly language we can only use the 532, as field names of **structs** are not shown.

<sup>21</sup>Note the need to write this in assembly language instead of C, since C would not give us direct access to the registers or the stack. Most of Linux is written in C, but machine-dependent operations like the one here must be done in assembly language.

### 4.3 Process States

The OS maintains a **process table** which shows the state of each process in memory, mainly Run state versus Sleep state. A process which is in Run state means that it is ready to run but simply waiting for its next turn. The OS will repeatedly cycle through the process table, starting turns for processes which are in Run state but skipping over those in Sleep state. The processes in Sleep state are waiting for something, typically an I/O operation, and thus currently ineligible for turns. So, each time a turn ends, the OS will browse through its process table, looking for a process in Run state, and then choosing one for its next turn.

Say our application program **u** above contains a call to **scanf()** to read from the keyboard. Recall that **scanf()** calls the OS function **read()**. The latter is in the OS, so now the OS is running, and this function will check to see whether there are any characters ready in the keyboard buffer. Typically there won't be any characters there yet, because the user has not started typing yet. In this case the OS will place this process in Sleep state, and then start a turn for another process.

How does a process get switched to Run state from Sleep state? Say our application program **u** was in Sleep state because it was waiting for user input from the keyboard, waiting for just a single character, in **raw** mode.<sup>22</sup>

As explained earlier, when the user hits a key, that causes a hardware interrupt from the keyboard, which forces a jump to the OS. Suppose at that time program **v** happened to be in the midst of a quantum. The CPU would temporarily suspend **v** and jump to the keyboard driver in the OS. The latter would notice that the program **u** had been in Sleep state, waiting for keyboard input, and would now move **u** to Run state.

Note, though, that that does not mean that the OS now starts **u**'s next turn; **u** simply becomes eligible to run. Recall that each time one process' turn ends, the OS will select another process to run, from the set of all processes currently in Run state, and **u** will now be in that set. Eventually it will get a turn.

To make all this concrete, say **u** is running the **vi** text editor, **v** is running a long computational program, and user **w** is also running some long computational program. Remember, **vi** is a program; its source code might have a section like this:

```
while (1) {
    KeyStroke = getch();
    if (KeyStroke == 'x') DeleteChar();
    else if (KeyStroke = 'j') CursorDown();
    else if ...
}
```

Here you see how **vi** would read in a command from the user, such as **x** (delete a character), **j** (move the cursor down one line) and so on. Of course, **DeleteChar()** etc. are functions, whose code is not shown here.

When during a turn for **u**, **vi** hits the **getch()** line, the latter calls **read()**, which is in the OS, so now the OS is running. Assuming **u** has not hit a key yet (almost certainly the case, since the CPU is a lot faster than **u**'s hand, not to mention the fact that **u** might have gone for a lunch break), the OS will mark **u**'s process as being in Sleep state in the OS's Process Table.

<sup>22</sup>Recall from our earlier unit that in this mode, a character is made available to a program as input the minute the user strikes the key. In ordinary, **cooked**, mode, the program doesn't see the input until the user hits the Enter key.

Then the OS will look for a process in Run state to give the next turn to. This might be, say, **v**. It will run until the timer interrupt comes. That pulse of current will cause the CPU to jump to the OS, so the OS is running again. This time, say, **w** will be run.

Say during **w**'s turn, **u** finally hits a key. The resulting interrupt from the keyboard again forces the CPU to jump to the OS. The OS read the character that **u** typed, notices in its Process Table that **u**'s process was in Sleep state pending keyboard input, and thus changes **u**'s entry in the Process Table to Run state. The OS then does IRET, which makes **w**'s process resume execution, but when the next timer interrupt comes, the OS will likely give **u** a turn again.

#### 4.4 What About Background Jobs?

For example, you could run the Firefox Web browser by typing

```
% firefox
```

at the command line. But that would tie up the window you typed it into; you wouldn't be able to run any other commands until you quit Firefox, which might be a very long time. To solve this problem, you can run it *in the background*, as follows.

You could type

```
% firefox &
```

with the ampersand meaning, "Run this job in the background." The shell would start Firefox, and then print out a prompt, inviting you to run other commands. What is really happening here?

The first point to be made here is that it really means nothing to the OS. The entry for the Firefox process in the OS' process table will not state whether this program is background or foreground; the OS doesn't know.

The only meaning of that ampersand was for the shell. We used it to tell the shell, "Go ahead and launch **a.out** for me, but don't wait for it to finish before giving me your '%' prompt again. Give me the prompt right away, because I want to run some other programs too."<sup>23</sup>

#### 4.5 Threads: "Lightweight Processes"

A very important type of application programming is that of **threads**. Thread libraries are offered by any modern OS. On Unix-family systems, for instance, a popular thread package is **pthread**s, which we will assume for concreteness here.

Threading is pervasive in computing today. Most Web servers are threaded, for instance. Also many GUI ("graphical user interface") programs are threaded too.

<sup>23</sup>To do this, the shell must first create a new process, by calling the **fork()** system call, and then calling **execve()** from the child process. That way, the parent process—the shell—can keep running and give you the prompt.

Threading is also the standard way of programming on multiprocessor systems. Since such systems are now common even at the household level—the term **dual core** means two CPUs—you can see how important threaded programming has become.

#### 4.5.1 The Mechanics

Threads, of the “system” type discussed here, are quasi-processes.<sup>24</sup> If your program creates three threads on a Unix-family system such as Linux, for example, you may (depending on the OS) see three entries for them in the output of the **ps** command which displays processes. (For Linux, type **ps axms**. These will correspond to three entries in the process table. Thus the threads take turns running, in the usual timesharing manner. The turn taking includes the other processes too, of course.

From the programmer’s point of view, the main difference between threads and processes is that in threaded programs all global variables of the program are shared.<sup>25</sup> This is the way the different threads communicate with each other, by reading and writing these shared variables.

This may come as a shock to readers who were taught that shared variables are “evil.” In my opinion, shared variables are fine for general programming (within reason), but whether one agrees with that notion or not, **the fact is that in threaded programming one must use global variables.**

Each thread does have its own local variables, though. So, the various threads of a program will share the same **.data** and **comm** sections, but each will have its own separate stack.

One creates a thread by calling **pthread\_create()**. One of its arguments must be a pointer to a function in the application program. For instance, in a GUI program, we might have one thread for the mouse, one thread for the keyboard, etc. The function which handles mouse events (movement and clicks) would be specified as an argument to **pthread\_create()**, and basically the thread would consist of executing that function. There would also be a function for the keyboard, etc.

Key to threaded programming is **locks**. Suppose for instance that an airline reservation system is threaded, and that there is one seat left for a particular flight. We want to make sure to avoid a situation in which two clerks both sell that last seat. (Ignore the issue of overbooking.) So, the two threads running for the two clerks must cooperate, and not try to access that flight’s record at the same time. In other words, we want access to that record to be **atomic**. Lock variables, of type **pthread\_mutex\_t**, are locked and unlocked via calls to **pthread\_mutex\_lock()** and **pthread\_mutex\_unlock()**, take care of this for us.

Threads are often called **lightweight processes**. The term *lightweight* here refers to the fact that creating a thread involves much less overhead than creating a process. In **pthreads**, for instance, calling **pthread\_create()** has a lot less work to do than would be involved in calling **execve()** to make a new process. Among other things, the latter requires finding memory for the new process, whereas a new thread would simply use the same memory.

---

<sup>24</sup>What we are describing here are **system-level** threads, i.e. those provided by the OS, which is the case for **pthreads**. There are also **user-level** threads. These, exemplified by the GNU **pth** library, are handled internally to the application program. Basically, the library runs its own private processes, with its own private process table!

<sup>25</sup>It is actually possible to have different processes share memory too, using the system call **shmget()**. However, this is unwieldy and involves a lot of system overhead, and is not commonly done.

### 4.5.2 Threads Example

Examples of threaded programming tend to be rather elaborate, so I’ve written the following simple (if a bit contrived) example:

```

1 // HelloWorld.c, a baby-level first example of threaded programming.
2
3 // Doesn't actually print out "hello world." :-) But it's of that level
4 // of simplicity, so I gave it that name anyway.
5
6 // The program reads from each of several keyboard input sources. To
7 // avoid having to introduce network programming, we will have these
8 // sources be from several windows on the same console on the same
9 // machine.
10
11 // The issue here is that we don't know which keyboard input will have
12 // some activity next. If we simply call scanf() in nonthreaded fashion
13 // (and don't resort to something called nonblocking I/O), our program
14 // would wait forever for input from one nonactive window when there is
15 // input waiting at another window. The solution is to use threads,
16 // with one thread for each window.
17
18 // Each source of keyboard input is considered a different file. (In
19 // Unix, I/O devices are considered "files.") We determine the file
20 // name by running the "who am i" command in the given window. If for
21 // instance the answer is pts/6, the keyboard input there is /dev/pts/6.
22
23 // Each time there is keyboard input from somewhere, the program will
24 // report what number was typed.
25
26 // For simplicity, the program has no mechanism for stopping.
27
28 // Linux compilation: gcc -g -o hw HelloWorld.c -lpthread
29
30 // usage: hw first_keybd_input_filename second_keybd_input_filename ...
31 // e.g. hw /dev/pts/4 /dev/pts/6
32
33 #define MAX_THREADS 20
34
35 #include <stdio.h>
36 #include <pthread.h> // required for threads usage
37
38 int nthreads; // number of threads to be run
39
40 // lock, needed so that the printf() in one thread does not rudely
41 // interrupt the printf() in another when a context switch occurs
42 pthread_mutex_t numlock = PTHREAD_MUTEX_INITIALIZER;
43 // ID structs for the threads
44 pthread_t id[MAX_THREADS];
45
46 // each thread runs this routine
47 void *worker(char *fname) // fname is file name
48 { FILE *kb; // note separate stacks, so separate kb's
49   int num; // separate num's too
50   kb = fopen(fname,"r"); // open this keyboard input, read only
51   while(1) {
52     fscanf(kb,"%d",&num); // wait for user to type in this window
53     pthread_mutex_lock(&numlock);
54     printf("read %d from %s\n",num,fname);
55     pthread_mutex_unlock(&numlock);
56   }
57 }
```

```

58
59 main(int argc, char **argv)
60 { int i;
61   nthreads = argc - 1;
62   // get threads started
63   for (i = 0; i < nthreads; i++) {
64     // this next call says to create a thread, record its ID in the array
65     // id, and get the thread started executing the function worker(),
66     // passing the argument argv[i+1] to that function
67     pthread_create(&id[i], NULL, worker, argv[i+1]);
68   }
69   while(1) ; // dummy loop, to keep main() from exiting, killing
70             // the other threads
71 }

```

The program reads keyboard input from two different windows on the same console on the same machine. This is the contrived part. A typical example would have input coming from keyboards on other machines, with the characters being sent across a network. But I wanted to avoid bringing in network programming.

As you can see, `main()` starts up the threads. Once they are started, they take turns running, like processes. Note that `main()` is considered a thread too, the parent of those created by it.

I ran the program on the windows `/dev/pts/4` and `/dev/pts/6` on my Linux machine at home.<sup>26</sup> In each window I typed

```
% sleep 10000
```

to put the shell to sleep, ensuring that the input I type in those windows would go to my program rather than to the shells.

In a third window, I ran the program. In window 4 I then typed 5 and 12, and then 13 in window 6, then 168 in window 4 again. The output in the third window was

```

% a.out /dev/pts/4 /dev/pts/6
read 5 from /dev/pts/6
read 12 from /dev/pts/6
read 13 from /dev/pts/4
read 168 from /dev/pts/6

```

**REMINDER TO THE READER:** Don't worry so much about how to do `pthread`s programming; focus on the concepts relevant to our course. Think especially of the role of interrupts here. Threads take turns running, with a turn ending either from a timer interrupt or from a thread voluntarily relinquishing its turn via a system call, in this case `read()` being called by `fscanf()`. In the latter situation, an interrupt from the keyboard will change a thread from Sleep to Run state.

## 4.6 Making These Concepts Concrete: Commands You Can Try Yourself

First, try the `ps` command. On Unix systems, much information on current processes is given by the `ps` command, including:

<sup>26</sup>As the comments in the code indicate, I determined their device names by running the `who am i` command in each window.



## 5 VIRTUAL MEMORY

**daemons.** A **daemon** in Unix parlance means a kind of server program. In Unix, the custom is to name such programs with a ‘d’ at the end, standing for “daemon.”

For example, you can see above that **init** started **lpd** (“line printer daemon”), which is the print server.<sup>27</sup> When users issue **lpr** and other print commands, the OS refers them to **lpd**, which arranges for the actual printing to be done.

Often daemons spawn further processes. For example, look at the line

```
|--inetd---in.rlogind---tcsh---pstree
```

The **inetd** is an Internet request server. The system administrator can optionally run this daemon in lieu of some other network daemons which are assumed to run only rarely. Here the user (me) had done **rlogin**, a login program like **ssh**<sup>28</sup>, to remotely login to this machine. The system administrators had guessed that **rlogin** would be used only rarely, so they did not have **init** start the corresponding daemon, **in.rlogind**, upon bootup. Instead, any network daemon not started at bootup will be launched by **inetd** when the need arises, which as you can see occurred here for **in.rlogind**. The latter then started a shell for the user who logged in (me), who then ran **pstree**. Note again that the shell is the entity which got **pstree** started.

## 5 Virtual Memory

### 5.1 Make Sure You Understand the Goals

Now let us add in the effect of virtual memory (VM). VM has the following basic goals, which are so important to understand that I am describing in separate short sub-subsections:

#### 5.1.1 Overcome Limitations on Memory Size

We want to be able to run a program, or collectively several programs, whose memory needs are larger than the amount of physical memory available.

#### 5.1.2 Relieve the Compiler and Linker of Having to Deal with Real Addresses

We want to facilitate **relocation** of programs, meaning that the compiler and linker do not have to worry about where in memory a program will be loaded when it is run.

#### 5.1.3 Enable Security

We want to ensure that one program will not accidentally (or intentionally) harm another program’s operation by writing to the latter’s area of memory, read or write to another program’s I/O streams, etc.

---

<sup>27</sup>Another common print server daemon is **cupsd**.

<sup>28</sup>But no longer allowed on CSIF.

## 5.2 The Virtual Nature of Addresses

The word *virtual* means “apparent.” It will appear that a program resides entirely in main memory, when in fact only part of it is there. It will appear that your program variables are stored at the places the linker assigned to them,<sup>29</sup> when in fact they actually will be stored somewhere else.

(For the time being, it will be easier to understand VM by assuming there is no cache. We will return to this in Section 5.8.)

To make this more concrete, suppose our C source file includes the following statements:

```
int x;
...
x = 8;
printf("%d", &x);
```

Recall that addresses of variables are set in the following manner. Say **x** is global. Then the compiler will generate a **.comm** line in assembly language it produces.<sup>30</sup> The latter will be translated by the assembler, and then the linker will deal with the machine code. Among other things, the linker will assign an address for **x**. It will be this address which is printed out above.

If **x** had been local, it would be stored on the stack, but that is still memory of course, and thus it would have an address too. Again, that would be printed out.

In any case, either of these addresses would be fake. Let’s see what this really means. Suppose that **x** is global, and that the compiler and linker assign it address 200, i.e. **&x** is 200. Then the above code

```
x = 8;
```

would correspond to an instruction in **a.out** something like

```
movl $8, 200
```

At the time **a.out** is loaded by the OS into memory, the OS will divide both the **.text** (instructions) and data portions of **a.out** (**.data**, **.bss**, stack, heap, etc.) into chunks, and find unused places in memory at which to place these chunks. The chunks are called **pages** of the program, and the same-sized places in memory in which the OS puts them are called pages of memory.<sup>31</sup> The OS sets up a **page table** for this process, which is an array which is maintained by the OS, in which the OS records the correspondences, i.e. lists which page of the process is stored in which page of memory. Remember, the OS is a program, and the page table is an array in that program, and thus it too is in memory.

What appears to be in word 200 in memory from the program code above may actually be in, say, word 1204. At the time the CPU executes that instruction, the CPU will determine where “Word 200” really is by

<sup>29</sup>You can determine these by running **readelf** on your executable file, with the **-s** option.

<sup>30</sup>If **x** had been declared in initialized form, e.g. **int x = 28;** then it would have been in a **.data** section.

<sup>31</sup>As with the stack, etc., keep in mind that these pages are only conceptual; there is no “fence” or any other physical demarcation between one page in memory and the next. A page is simply a unit of measurement, like acres and liters.

doing a lookup in the page table. In our example here, the table will show that the item we want is actually in word 1204, and the CPU will then write to that location. The hardware would put 1204 on the address bus, 8 on the data bus, and would assert the Memory Write line in the control bus. ✓

In this example, we say the **virtual address** is 200, and the **physical address** is 1204.

Note that while virtual addressing is enabled (it is turned off in Kernel Mode), all addresses seen by the hardware will be virtual. For example, the contents of ESP will be a virtual address, with the real top-of-stack address being something else, not what ESP says. A similar statement holds for the PC, etc.

### 5.3 Overview of How the Goals Are Achieved

Let's look at our stated goals in Section 5.1 above, and take a first glance at how they are achieved (details will be covered later):

#### 5.3.1 Overcoming Limitations on Memory Size

To conserve memory space, the OS will initially load only part of **a.out** into memory, with the remainder being left back on disk. The pages of the program which are not loaded will be marked in the page table as currently being **nonresident**, meaning not in memory, and their locations on disk will be shown in the table. During execution of the program, if the program needs one of the nonresident pages, the CPU will notice that the requested page is nonresident. This is called a **page fault**, and it causes an internal interrupt. The interrupt number is 0xe, i.e. entry 0xe in the IDT. That causes a jump to the OS, which will bring in that page from disk, and then jump back to the program, which will resume at the instruction which accessed the missing page.

Note that pages will often go back and forth between disk and memory in this manner. Each time the program needs a missing page, that page is brought in from disk and a page which had been resident is written back to disk in order to make room for the new one. Note also that a given virtual page might be in different physical pages at different times.

A big issue is the algorithm the OS uses to decide which page to move back to disk (i.e. which page to replace) whenever it brings a page from disk after a page fault. Due to the huge difference in CPU and disk speeds, a page fault is a catastrophic even in terms of program speed. We hope to have as few page faults as possible when our program runs.

So, we want to check the pages to evict very carefully. If we often evict a page which is needed again very soon, our program's performance will really suffer. This is called **thrashing**. Details of page replacement policies are beyond the scope of this document here, but one point to notice is that the policy will be chosen so as to work well on "most" programs. For any given policy, it will do well on some programs (i.e. produce few page faults) while doing poorly on some other programs (produce many page faults).

So, what they try to do is come up with a policy which works reasonably well on a reasonably broad variety of programs. Most policies are some variation on the Least Recently Used policy common for associative caches.

### 5.3.2 Relieving the Compiler and Linker of Having to Deal with Real Addresses

This is clear from the example above, where the location “200” which the compiler and linker set up for `x` was in effect changed by the OS to 1204 at the time the program was loaded. The OS recorded this in the page table, and then during execution of the program, the VM hardware in the CPU does lookups in the page table to get the correct addresses. The point is that the compiler and linker can assign `x` to location 200 without having to worry whether location is actually available at the time the program will be run, because that variable actually **won’t** be at 200.

### 5.3.3 Enabling Security

The page table will consist of one entry per page. That entry will, as noted earlier, include information as to where in memory that page of the program currently resides, or if currently nonresident, where on disk the page is stored. But in addition, the entry will also list the permissions the program has to access this particular page—read, write, execute—in a manner analogous to file-access permissions. If an **access violation** occurs, the VM hardware in the CPU will cause an internal interrupt (again it’s interrupt number 0xe, as for page faults), causing the OS to run. The OS will then kill the process, i.e. remove it from the process table.

Normally data-related sections such as `.data` and the stack will be the only ones with write permission. However, you can also arrange for the `.text` section to be writable, via the `-N` option to `ld`.

You might wonder why execute permission is included. One situation in which this would be a useful check is that in which we have a pointer to a function. If for example we forgot to initialize the pointer, a violation will be detected.

Suppose for instance your program tries to read from virtual page number 40000 but that virtual address is out of the ranges of addresses which the program is allocated. The hardware will check the entry for 40000 in the page table, and find that not only is that page not resident, but also it isn’t on disk either. Instead, the entry will say that there is no virtual page 40000 among the pages allocated to this program.

We don’t want an ordinary user programs to be able to, say maliciously, access the I/O streams of other programs. To accomplish this, we want to forbid a user program from accessing the I/O buffer space of another program, which is easy to accomplish since that space is in memory; we simply have the OS set up the page table for that program accordingly. But we also need to forbid a user program from directly performing I/O, e.g. the `INB` and `OUTB` instructions on Intel machines.

This latter goal is achieved by exploiting the fact that most modern CPUs run in two or more **privilege levels**. The CPU is designed so that certain instructions, for example those which perform I/O such as `INB` and `OUTB`, can be executed only at higher privilege levels, say Kernel Mode. (The term **kernel** refers to the OS.)

But wait! Since the OS is a program too, how does *it* get into Kernel Mode? Obviously there can’t be some instruction to do this; if there were, then ordinary user programs could use it to put themselves into Kernel Mode. So, how is it done?

When the machine boots up, it starts out in Kernel Mode. Thus the OS starts out in Kernel Mode. Among

other things, the interrupts can be configured to change the privileges level of the CPU. Recall that the OS starts a turn for a user program (including that program's first turn) by executing an IRET instruction, a software interrupt. So, the OS can arrange both for user programs to run in non-Kernel Mode and for Kernel Mode to be restored when an interrupt comes during a user program is running. So for example, an interrupt from the timer will not only end a user program's turn, but also will place the CPU in Kernel Mode, thus having the OS run in that mode.

## 5.4 Who Does What When?

Note carefully the roles of the players here: It is the software, the OS, that creates and maintains the page table, but it is the hardware that actually uses the page table to generate physical addresses, check page residency and check security. In the event of a page fault or security violation, the hardware will cause a jump to the OS, which actually responds to those events.

The OS writes to the page table (including creating it in the first place), and the hardware reads it.

The hardware will have a special Page Table Register (PTR) to point to the page table of the current process. When the OS starts a turn for a process, it will restore the previously-saved value of the PTR, and thus this process' page table will now be in effect.

On the Pentium, the name of the PTR is CR3. (Actually, the Pentium uses a two-level hierarchy for its page tables, but we will not pursue that point here.)

## 5.5 Details on Usage of the Page Table

### 5.5.1 Virtual-to-Physical Address Translation, Page Table Lookup

Whenever the running program generates an address—either the address of an instruction, as will be the case for an instruction fetch, or the address of data, as will be the case during the execution of instructions which have memory operands—this address is only virtual. It must be translated to the physical address at which the requested item actually resides. The circuitry in the CPU is designed to do this translation by performing a lookup in the page table.

For convenience, say the page size is 4096 bytes, which is the case for Pentium CPUs. Both the virtual and physical address spaces are broken into pages. For example, consider virtual address 8195. Since

$$8195 = 2 \times 4096 + 3 \tag{1}$$

that address would be in virtual page 2 (the first page is page 0). We can then speak of how far into a page a given address is. Here, because of the remainder 3 in Equation (1), we see that virtual address 8195 is byte 3 in virtual page 2. We refer to the 3 as the **offset** within the page, i.e. its distance from the beginning of the page.

You can see that for any virtual address, the virtual page number is equal to the address divided by the page size, 4096, and its offset within that page is the address mod 4096. Using our knowledge of the properties

of powers of 2 and the fact that  $4096 = 2^{12}$ , this means that for a 32-bit address (which we'll assume throughout), the upper 20 bits contain the page number and the lower 12 bits contain the offset.

The page/offset description of the position of a byte is quite analogous to a feet/inches description of distance. We could measure everything in inches, but we choose to use a larger unit, the foot, to give a rough idea of the distance, and then use inches to describe the remainder. The concept of offset will be very important in what follows.

Now, to see how the page table is used to convert virtual addresses to physical ones, consider for example the Intel instruction

```
movl $6, 8195
```


This would copy the constant 6 to location 8195. Remember, this is page 2, offset 3. However, it is a virtual address. The hardware would see the 8195,<sup>32</sup> The hardware knows that any address given to it is a virtual one, which must be converted to a physical one. So, the hardware would look at the entry for virtual page 2, and find what physical page that is; suppose it is 5. Then that page starts at physical memory location  $5 \times 4096 = 20480$ .

What about the offset within that physical page 5? The custom is that an item will have the same offset, no matter whether we are talking about its virtual address or its physical one. So, the offset of our destination in physical page 5 would be 3. In other words, the physical address is

$$5 \times 4096 + 3 = 20483 \quad (2)$$

The CPU would now be ready to execute the instruction, which, to refresh your memory, was

```
movl $6, 8195
```

The CPU knows that the real location of the destination is 20483, not 8195. It would put 20483 on the address bus, put 6 on the data bus, and assert the Memory Write line in the control bus, and the instruction would be done. 

### 5.5.2 Layout of the Page Table

Suppose the entries in our page table are 32 bits wide, i.e. one word per entry.<sup>33</sup> Let's label the bits of an entry 31 to 0, where Bit 31 is in the most-significant (i.e. leftmost) position and Bit 0 is in the least significant (i.e. rightmost) place. Suppose the format of an entry is as follows:

- Bits 31-12: physical page number if resident, disk location if not

<sup>32</sup>Remember, it will be embedded within the instruction itself, as this is direct addressing mode.

<sup>33</sup>If we were to look at the source code for the OS, we would probably see that the page table is stored as a very long array of type **unsigned int**, with each array element being one page table entry.

- Bit 11: 1 if page is resident, 0 if not
- Bit 10: 1 if have read permission, 0 if not
- Bit 9: 1 if have write permission, 0 if not
- Bit 8: 1 if have execute permission, 0 if not
- Bit 7: 1 if page is “dirty,” 0 if not (see below)
- Bits 6-0: other information, not discussed here

Now, here is what will happen when the CPU executes the instruction

```
movl $6, 8195
```

above:

- The CPU does the computation  $\times 4096 + 3 = 20483$  and finds that virtual address 8195 is virtual page number 2, offset 3.
- Since we are dealing with virtual page 2, the CPU will go to get the entry for that virtual page in the page table, as follows. Suppose the contents of the PTR is 5000. Then since each entry is 4 bytes long, the table entry of interest here, i.e. the entry for virtual page 2, is at location

$$5000 + 4 \times 2 = 5008 \quad (3)$$

The CPU will read the desired entry from that location, getting, say, 0x000005e0.

- The CPU looks at Bits 11-8 of that entry, getting 0xe, finding that the page is resident (Bit 11 is 1) and that the program has read and write permission (Bits 10 and 9 are 1) but no execute permission (Bit 8 is 0). The permission requested was write, so this is OK.
- The CPU looks at Bits 31-12, getting 5, so the hardware would know that virtual page 2 is actually physical page 5. The virtual offset, which we found earlier to be 3, is always retained, so the CPU now knows that the physical address of the virtual location 8195 is

$$5 \times 4096 + 3 = 20483 \quad (4)$$

- The CPU puts the latter on the address bus, puts 6 on the data bus, and asserts the Write line in the bus. This writes 6 to memory location 20483, and we are done.

By the way, all this was for Step C of the above MOV instruction. The same actions would take place in Step A. The value in the PC would be broken down into a virtual page number and an offset; the virtual page number would be used as an index into the page table; Bits 10 and 8 in the page table element would be checked to see whether we have permission to read and execute that instruction; assuming the permissions

are all right, the physical page number would be obtained from Bits 31-12 of the page table element; the physical page number would be combined with the offset to form the physical address; and the physical address would be placed in the MAR and the instruction fetched.

Recall from above that the upper 20 bits of an address form the page number, and the lower 12 bits form the offset. A similar statement holds for physical addresses and physical page numbers. So, all the hardware need do is: use the upper 20 bits of the virtual address as an index in the page table (i.e. multiply this by 4 and add to c(PTR)); take bits 31-12 of from the table entry reached in this manner, to get the physical page number; and finally, concatenate this physical page number with the lower 12 bits of the original virtual address. Then the hardware has the physical address, which it places on the address bus.

### 5.5.3 Page Faults

Suppose in our example above Bit 11 of the page table entry had been 0, indicating that the requested page was not in memory. As mentioned earlier, this event is known as a **page fault**. If that occurs, the CPU will perform an internal interrupt, and will also record the PC value of the instruction which caused the page fault, so that that instruction can be restarted after the page fault is processed. In Pentium CPUs, the CR2 register is used to store this PC value. This will force a jump to the OS.

The OS will first decide which currently-resident page to replace, then will write that page back to disk, if the Dirty Bit is set (see below). The OS will then bring in the requested page from disk. The OS would then update two entries in the page table: (a) it would change the entry for the page which was replaced, changing Bit 11 to 0 to indicate the page is not resident, and changing Bits 31-12 and possible Bit 7; and (b) the OS would update the page table entry of the new item's page, to indicate that the new item is resident now in memory (setting Bit 11 to 1), show where it resides (by filling in Bits 31-12), and setting Bit 7 to 0.

The role of the Dirty Bit is as follows: When a page is brought into memory from disk, this bit will be set to 0. Subsequently, if the page is written to, the bit will be changed to 1. So, when it comes time to evict the page from memory, the Dirty Bit will tell us whether there is any discrepancy between the contents of this page in memory and its copy on disk. If there is a difference, the OS must write the new contents back to disk. That means all 4096 bytes of the page. We must write back the whole page, because we don't know what bytes in the page have changed. The Dirty Bit only tells us that there has been some change(s), not where the change(s) are. So, if the Dirty Bit is 0, we avoid a time-consuming disk write. ✓

Since accessing the disk is far, far slower than accessing memory, a program will run quite slowly if it has too many page faults. If for example your PC at home does not have enough memory, you will find that you often have to wait while a large application program is loading, during which time you can hear the disk drive doing a lot of work, as the OS ejects many currently-resident pages to bring in the new application.

### 5.5.4 Access Violations

If on the other hand an access violation occurs, the OS will announce an error—in Unix, referred to as a **segmentation fault**—and kill the process, i.e. remove it from the process table.

For example, considering the following code:

```


1  int q[200];
2
3  main()
4
5  {  int i;
6
7      for (i = 0; i < 2000; i++) {
8          q[i] = i;
9      }
10
11 }
```

Notice that the programmer has apparently made an error in the loop, setting up 2000 iterations instead of 200. The C compiler will not catch this at compile time, nor will the machine code generated by the compiler check that the array index is out of bounds at execution time.

If this program is run on a non-VM platform,<sup>34</sup> then it will merrily execute without any apparent error. It will simply write to the 1800 words which follow the end of the array **q**. This may or may not be harmful, depending on what those words had been used for.

But on a VM platform, in our case Unix, an error will indeed be reported, with a “Segmentation fault” message.<sup>35</sup> However, as we look into how this comes about, the timing of the error may surprise you. The error is not likely to occur when **i** = 200; it is likely to be much later than that.

To illustrate this, I ran this program under **gdb** so that I could take a look at the address of **q[199]**.<sup>36</sup> After running this program, I found that the seg fault occurred not at **i** = 200, but actually at **i** = 728. Let’s see why.

From queries to **gdb** I found that the array **q** ended at 0x080497bf, i.e. the last byte of **q[199]** was at that address. On Intel machines, the page size is 4096 bytes, so a virtual address breaks down into a 20-bit page number and a 12-bit offset, just as in Section 5.5.1 above. In our case here, **q** ends in virtual page number 0x8049 = 32841, offset 0x7bf = 1983. So, after **q[199]**, there are still 4096-1984 = 2112 bytes left in the page. That amount of space holds 2112/4 = 528 **int** variables, i.e. elements “200” through “727” of **q**. Those elements of **q** don’t exist, of course, but as discussed in an earlier unit the compiler will not complain. Neither will the hardware, as we will be writing to a page for which we do have write permission. But when **i** becomes 728, that will take us to a new page, one for which we don’t have write (or any other) permission; the hardware will detect this and trigger the seg fault. 

We could get a seg fault not only by accessing off-limits data items, but also by trying to execute code at an off-limits location. For example, suppose in the above example **q** had been local instead of global. Then it would be on the stack. As we go past the end of **q**, we would go deeper and deeper into the stack. This may not directly cause a seg fault, if the stack already starts out fairly large and is stored in physically contiguous pages in memory. But we would overwrite all the preceding stack frames, including return addresses. When we tried to “return” to those addresses,<sup>37</sup> we would likely attempt to execute in a page for which we do not have execute permission, thus causing a seg fault.

<sup>34</sup> Recall that “VM platform” requires both that our CPU has VM capability, and that our OS uses this capability.

<sup>35</sup> On Microsoft Windows systems, it’s called a “general protection error.”

<sup>36</sup> Or I could have added a **printf()** statement to get such information. Note by the way that either running under **gdb** or adding **printf()** statement will change the load locations of the program, and thus affect the results.

<sup>37</sup> Recall that **main()** is indeed called by other code, as explained in our unit on subroutines.


As another example of violating execute permission, consider the following code, with a pointer to a function.<sup>38</sup>

```

1 int f(int x)
2 { return x*x; }
3
4 // review of pointers to functions in C/C++: below p is a pointer to a
5 // function; the first int means that whatever function p points to, it
6 // returns an int value; the second int means that whatever function p
7 // points to, it has one argument, an int
8 int (*p)(int);
9
10 main()
11 { int u;
12
13   p = f; // point p to f
14   u = (*p)(5); // call f with argument 5
15   printf("%d\n",u); // prints 25
16 }
```

If we were to forget to include the line

```
u = (*p)(5);
```

then the variable **p** would not point to a function, and we would attempt to execute “code” in a location off limits to us. A seg fault would result. 

## 5.6 VM and Context Switches

A context switch will be accompanied by a switch to a new page table.

Say for example Process A has been running, but its turn ends and Process B is given a turn. Of course we must now use B’s page table. So, before starting B’s turn, the OS must change the PTR to point to B’s page table.

Thus, there is actually no such thing as “the” page table. There are many page tables. The term “the” page table merely means the page table which PTR currently points to.

## 5.7 Improving Performance—TLBs

Virtual memory comes at a big cost, in the form of overhead incurred by accessing the page tables. For this reason, the hardware will also typically include a **translation lookaside buffer** (TLB). This is a special cache to keep a copy of part of the page table in the CPU, to reduce the number of times one must access memory, where the page table resides.

One might ask, why not store the entire page table in the TLB? First of all, the page table theoretically consists of  $2^{32}/2^{12}$  entries. This is somewhat large. Though there are ways around this, but we would need

<sup>38</sup>If your classes on C/C++ did not cover this important topic of pointers to functions, the comments in the code below should be enough to introduce it to you. Pointers to functions are used in many applications, such as threaded programs, as mentioned earlier.

to change this large amount of data at each context switch, which would really slow down context switching. Note by the way that we would need to have special instructions which the OS could use to write to the TLB. That's OK, though, and there are other special instructions for the OS already.

## 5.8 The Role of Caches in VM Systems

Up to now, we have been assuming that the machine doesn't have a cache.<sup>39</sup> But in fact most machines which use VM also have caches, and in such cases, what roles do the caches play?

The central point is that speed is still an issue. The CPU will look for an item in its cache first, since the cache is internal to (or at least near) the CPU. If there is a cache miss, the CPU will go to memory for the item. If the item is resident in memory, the entire block containing the item will be copied to the cache. If the item is not resident, then we have a page fault, and the page must be copied to memory from disk, a very slow process, before the processing of the cache miss can occur.

### 5.8.1 Addressing

An issue that arises is whether the cache is the CPU will **virtually addressed** or **physically addressed**. Suppose for instance the instruction being executed reads from virtual address 200. If the cache is virtually addressed, then the CPU would do its cache lookup using 200 as its index. On the other hand, if the cache is physically addressed, then the CPU would first convert the 200 to its physical address (by checking the TLB, and then the page table if need be), and then do the cache lookup based on that address.

A possible advantage of virtual addressing of caches would be that if we have a cache hit, we eliminate the time delay needed for the virtual-to-physical address translation. Moreover, since we usually will have a hit, we can afford to put the TLB in not-so-fast memory external to the CPU, which is done in some MIPS models.

On the other hand, with physical cache addressing, two different processes could both have separate, unrelated copies of different blocks in the cache at the same time, but with the same virtual addresses. They could coexist in the cache since their physical addresses would be different. That would mean we would not have to flush the cache at each new timeslice, possibly increasing performance.

### 5.8.2 Hardware Vs. Software

Note that cache design is entirely a hardware issue. The cache lookup and the block replacement upon a miss is hard-wired into the circuitry. By contrast, in the VM case it is a mixture of hardware and software. The hardware does the page table lookup, and checks page residency and access permissions. But it is the software—the OS—which creates and maintains the page table. Moreover, when a page fault occurs, it is the OS that does the page replacement and updates the page table.

So, you could have two different versions of Unix, say,<sup>40</sup> running on the same machine, using the same

---

<sup>39</sup>Except for a TLB, which is of course very specialized.

<sup>40</sup>Or, Linux versus Windows, etc.

compiler, etc. and yet they may have quite different page fault rates. The page replacement policy for one OS may work out better for this particular program than does the one of the other OS.

Note that the OS can tell you how many page faults your program has had (see the **time** command below); each page fault causes the OS to run, so the OS can keep track of how many page faults your program had incurred. By contrast, the OS can NOT keep track of how many cache misses your program has had, since the OS is not involved in handling cache misses; it is done entirely by the hardware.

## 5.9 Making These Concepts Concrete: Commands You Can Try Yourself

The Unix **time** command will report how much time your program takes to run, how many page faults it generated, etc. Place it just before your program's name on the command line. (This program could be either one you wrote, or something like, say, **gcc**.) For example, if you have a program **x** with argument **12**, type

```
% time x 12
```

instead of

```
% x 12
```

In fact, trying running this several times in quick succession. You may find a reduction in page faults, since the required pages which caused page faults in the first run might be resident in later runs.

Also, the **top** program is very good, displaying lots of good information on the memory usage of each process.

## 6 A Bit More on System Calls

Recall that the OS makes available to application programs services such as I/O, process management, etc. It does this by making available functions that application programs can call. Since these are functions in the OS, calls to them are known as **system calls**.

When you call **printf()**, for instance, it is just in the C library, not the OS, but it in turn calls **write()** which *is* in the OS.<sup>41</sup> The call to **write()** is a system call.

Or you can make system calls in your own code. For example, try compiling and running this code:

```
main()
{ write(1, "abc\n", 4); }
```

<sup>41</sup>There is a slight complication here. Calling **write()** from C will take us to a function in the C library of that name. It in turn will call the OS function. We will ignore this distinction here.

## 6 A BIT MORE ON SYSTEM CALLS

The function `write()` takes three arguments: the file handle (here 1, for the Unix “file” `stdout`, i.e. the screen); a pointer to the array of characters to be written [note that NULL characters mean nothing to `write()`]; and the number of characters to be written.

Similarly, executing a `cout` statement in C++ ultimately results in a call to `write()` too. In fact, the G++ compiler in GCC translates `cout` statements to calls to `printf()`, which as we saw calls `write()`. Check this by writing a small C++ program with a `cout` in it, and then running the compiled code under `strace`.

A non-I/O example familiar to you is the `execve()` service is used by one program to start the execution of another. Another non-I/O example is `getpid()`, which will return the process number of the program which calls it.

Calling `write()` means the OS is now running, and `write()` will ultimately result in the OS running code which uses the OUT machine instruction. Recall, though, that we want to arrange things so that only can execute instructions like Intel’s IN and OUT. So the hardware is designed so that these instructions can be executed only in Kernel Mode.

For this reason, one usually cannot implement a system call using an ordinary subroutine CALL instruction, because we need to have a mechanism that will change the machine to Kernel Mode. (Clearly, we cannot just have an instruction to do this, since ordinary user programs could execute this instruction and thus get into Kernel Mode themselves, wreaking all kinds of havoc!) Another problem is that the linker will not know where in the OS the desired subroutine resides.

Instead, system calls are implemented via an instruction type which is called a **software interrupt**. On Intel machines, this takes the form of the INT instruction, which has one operand.

We will assume Linux in the remainder of this subsection, in which case the operand is 0x80.<sup>42</sup> In other words, the call to `write()` in your C program (or in `printf()` or `cout`, which call `write()`) will execute

```
... # code to put parameters values into designated registers
int $0x80
```

The INT instruction works like a hardware interrupt, in the sense that it will force a jump to the OS, and change the privilege level to Kernel Mode, enabling the OS to execute the privileged instructions it needs. You should keep in mind, though, that here the “interrupt” is caused deliberately by the program which gets “interrupted,” via an INT instruction. This is much different from the case of a hardware interrupt, which is an action totally unrelated to the program which is interrupted.

The operand, 0x80 above, is the analog of the device number in the case of hardware interrupts. The CPU will jump to the location indicated by the vector at  $c(IDT)+8*0x80$ .<sup>43</sup>

When the OS is done, it will execute an IRET instruction to return to the application program which made the system call. The `iret` also makes a change back to User Mode.

As indicated above, a system call generally has parameters, just as ordinary subroutine calls do. One parameter is common to all the services—the service number, which is passed to the OS via the EAX register.

---

<sup>42</sup>Windows uses 0x21.

<sup>43</sup>By the way, users could cause mischief by changing this area of memory, so the OS would set up their page tables to place it off limits.

## 7 OS FILE MANAGEMENT

Other registers may be used too, depending on the service.

As an example, the following Intel Linux assembly language program writes the string “ABC” and an end-of-line to the screen, and then exits:

```
.text
hi: .string "ABC\n"
_start:

    # write "ABC\n" to the screen
    movl $4, %eax      # the write() system call, number 4 obtained
                       # from /usr/include/asm/unistd.h
    movl $1, %ebx      # 1 = file handle for stdout
    movl $hi, %ecx     # write from where
    movl $4, %edx     # write how many bytes
    int $0x80         # system call

    # call exit()
    movl $1, %eax     # exit() is system call number 1
    int $0x80         # system call
```

For this particular OS service, **write()**, the parameters are passed in the registers EBX, ECX and EDX (and, as mentioned before, with EAX specifying which service we want). Whoever wrote **write()** has forced us to use those registers for those purposes.

Note that we do indeed need to tell **write()** how many bytes to write. It will NOT stop if it encounters a null character.

Here are some examples of the numbers (to be placed in EAX before calling **int \$0x80**) of other system services:

read	3
file open	5
execve	11
chdir	12
kill	37

## 7 OS File Management

The OS will maintain a table showing the starting sectors of all files on the disk. (The table itself is on the disk.) The reason that the table need store only the starting sector of a given file is that the various sectors of the file can be linked together in “linked-list” fashion. In other words, at the end of the first sector of a file, the OS will store information stating the track and sector numbers of the next sector of the file.

The OS must also maintain a table showing unused sectors. When a user creates a new file, the OS checks this table to find space to put the file. Of course, the OS must then update its file table accordingly.

If a user deletes a file, the OS will update both tables, removing the file’s entry in the file table, and adding the former file’s space to the unused sector table.<sup>44</sup>

---

<sup>44</sup>However, the file contents are still there. This is how “undelete” programs work, by attempting to recover the sectors liberated

## A INTEL PENTIUM ARCHITECTURE

As files get created and deleted throughout the usage of a machine, the set of unused sectors typically becomes like a patchwork quilt, with the unused sectors dispersed at random places all around the disk. This means that when a new file is created, then *its* sectors will be dispersed all around the disk. This has a negative impact on performance, especially seek time. Thus OSs will often have some kind of **defragmenting** program, which will rearrange the positioning of files on the disk, so that the sectors of each individual file are physically close to each other on the disk.

## 8 To Learn More

There are several books on the Linux OS kernel. Before you go into such detail, though, you may wish to have a look at the Web pages <http://www.tldp.org/LDP/tlk/tlk.html> and <http://learnlinux.tsf.org.za/courses/build/internals/>, which appear to be excellent overviews.

## A Intel Pentium Architecture

We will assume Pentia with 32-bit word and address size. On Linux the CPU runs in **flat mode**, meaning that the entire address space of  $2^{32}$  bytes is available to the programmer.

The main registers of interest to us here are named EAX, EBX, ECX, EDX, ESI, EDI, EBP and ESP.<sup>45</sup> ESP is used as the stack pointer. The program counter is EIP. There is also a condition codes register EFLAGS, which is used to record whether the results of operations are zero, negative and so on.

In this document we use AT&T syntax for assembly code. Here is some sample code, which adds 4 elements of an array pointed to by ECX, with the sum being stored in EBX:

```
1      movl $4, %eax # copy the number 4 to EAX
2      movl $0, %ebx # copy the number 0 to EBX
3      movl %x, %ecx # copy the address of the memory location
4                      # whose label is x to ECX
5 top:  addl (%ecx), %ebx # add the memory word pointed to by ECS to EBX
6      addl $4, %ecx # add the number 4 to ECS
7      decl %eax # decrement EAX by 1
8      jnz top # if EAX not 0, then jump to top
```

Pentia use vector tables for interrupt handlers. Typical Pentium-based computers include one or more Intel 8259A chips for controlling and prioritizing bus access by input/output devices.

As with most modern processor chips, Pentia are highly pipelined and superscalar, the latter term meaning multiple ALU components, thus allowing more than one instruction to execute per clock cycle.

For more information see the various PDF files in <http://heather.cs.ucdavis.edu/~matloff/50/PLN>.

---

when the user (accidentally) deleted the file.

<sup>45</sup>We will use the all-caps notation, EAX, EBX, etc. to discuss registers in the text, even though in program code they appear as `%eax`, `%ebx`, ...