Overview of Functions of an Operating System

Norman Matloff
University of California, Davis

December 1, 2004

Contents

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.

For example, when your program, say **a.out**,[1] 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** during compilation of **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**.[2]

- 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.

  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

[1] Or it could be a program which you didn't write yourself, say **gcc**.

[2] An exception is the system call exit(), which is called by your program when it is finished with execution. If you do not include this call in your C source file, the compiler will put one in for you.

the person was logged in remotely, say via **telnet**—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. The only ways your program can stop running is if it voluntarily does so or is forced to stop by action occuring 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:

- application program loading for execution
- provide services, e.g. I/O, in the form of functions which the application programs can call (e.g. **read()**)
- enable **timesharing**, in which many application programs seem to be running simultaneously but are actually "taking turns," *by coordinating with hardware operations*
- enable **virtual memory** operations for application programs, which both allows flexible use of memory and enforces security, again by *by coordinating with hardware operations*
- maintenance the file system (e.g. recording the locations of all the files on the disk)
- manage I/O, including networking (ISRs)

Note carefully the interaction of the OS with the hardware. The OS manages the I/O devices, thus shielding the authors of application programs from having to deal directly with them. For example, suppose you wish to have your program read from a disk file. It would be a real nuisance if you had to deal with the minute details of the physical locations on disk of the various pieces of the file. Instead, you simply call fopen() and fread(), and the OS will worry about that for you. The OS knows these locations, by the way, because when the file was created, the creation was done through the OS too; at that time, the OS chose those locations, and recorded them.

In addition, if the hardware allows for it, and if the OS is designed to make use of it,[3] then the OS not only *relieves* the application programmer from having to perform the physical accesses of the disk, but also

[3] For example, Pentium CPUs do have such capabilities, but the Windows 98 OS does not make use of them. Linux and Windows NT (and post-NT Windows) do make use of them.

forbids him/her from doing so. This is for the purpose of security; we would not want the applications programmer to either inadvertently or maliciously trash someone else's disk file, for instance.

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 System Bootup

As will be explained later, when we wish to run an application program, the OS loads the program into memory. But how does the OS itself get loaded into memory and begin execution? The process by which this is done is called **bootup**.

The CPU hardware will be designed so that upon powerup the Program Counter (PC) is initialized to some specific value, say 0xfffffff0 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 say 0xfffffff0. 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 the OS) to memory, and then finally executes a JMP instruction (or equivalent) to that section of memory—so that now the OS is running. In a more complicated form, the boot loader only reads part of the OS into memory, and then performs a JMP instruction to the OS; 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,[4] 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.[5] That sector is called the Master Boot Record (MBR).

An operating system will typically define **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.[6] These partitions are defined in the **partition table** in the MBR.[7] 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

[4] These may or may not be used, depending on the OS. Windows uses them, but Linux doesn't.

[5] 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.

[6] A disk may consist of several **platters**, stacked on top of each other. If there are, say, four platters, then there would be a track 15, for instance, on each one. Then those four track 15s would be called **cylinder** 15. The name is of course chosen from the geometric view of the four tracks stacked on top of each other.

[7] Note that there is no physical separation from one partition to the next; the boundary is only defined by the partition table.

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 was written to then load into memory the third-stage code from the Windows partition. (Typically, this will be the only 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. 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 there.[8]

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.

3 Application Program Loading

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 **tcsh** 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()**, and enters a loop in which it reads the command you type using **scanf()**.[9]
- The shell will then make a system call, **execve()**,[10] 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 beginning section of **a.out**, which contains information on the size of the program, a list of the data segments used by the program, and so on.

[8] 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.

[9] 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.

[10] It will also call another system call, **fork()**, but we will not go into that here.

- 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 (called the **text** portion of the program in UNIX terminology), and data—static data items (scalar and array variables, called the **data** segment in UNIX), as well as for stack space and the **heap** (used for calls to **calloc()** and **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!

Note that **a.out** itself may call **execve()** and start other programs running. For example, **gcc** does this.[11] 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.[12] 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()**.

3.1 Making These Concepts Concrete: Commands You Can Try Yourself

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
```

You will see that a call **execve()** to launch **a.out**, as well as the various systems call made by the latter.

[11] 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.

[12] 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.

4 Timesharing

4.1 Many Processes, Taking Turns

Timesharing involves having several programs running in what appears to be a simultaneous manner.[13] Since the system has only one CPU (we will exclude the case of multiprocessor systems in this discussion), then 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 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**) 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.

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.[14]

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'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.)

Most modern CPUs run in two or more **privilege levels**. We for example would not want to give ordinary application programs direct access to I/O devices, e.g. disk drives, for security reasons. Thus the CPU is designed so that certain instructions, for example those which perform I/O, can be executed only at higher privilege levels, say Kernel Mode. (The term **kernel** refers to the OS.) Among other things, the interrupt from the timer places the CPU in Kernel Mode, so the interrupt not only causes the OS to run but it also gives the OS the privileges it needs.

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.

[13] These programs could be from different users or the same user; it doesn't matter.

[14] 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.

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.[15]

The OS maintains a **process table**, which is simply an array of **struct**s, 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.

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:

```
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.[16] Upon entry, **u**'s turn has just ended, having been interrupted by the timer. 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**.[17]

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.[18]

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.[19] 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 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.

[15] This was as of Linux kernel version 2.3. Also, I have slightly modified some of this section for the sake of simplicity.

[16] More precisely, it is code called by that ISR.

[17] The OS does keep track of which process it has currently given a turn to, so it 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.

[18] 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 **struct**s are not shown.

[19] 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.

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 get's **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, **v** is now running!

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 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 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 (say it was waiting for just a single character). 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.

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; it's source code might have a section like this:

```
while (1)  {
   scanf("%c",&KeyStroke);
   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 **scanf()** 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.

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?

Suppose you have a program, say **a.out**, which you expect to run a long time. While it is running, you want to be able to do something else, say use the **vim** text editor to edit the file xyz. On UNIX, you could type

```
% a.out &
% vim xyz
```

with the ampersand meaning, "Run this job in the background." What does that really mean?

The answer is that it really means nothing to the OS. The entry for the **a.out** process in the OS' process table will not state whether this program is background or foreground; it is simply a process.

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."

4.5 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:

- state (Run, Sleep, etc.)
- page usage (how many pages, number of page faults, etc.; see material on virtual memory below)
- ancestry (which process is the "parent" of the given process)

The reader is urged to try this out. You will understand the concepts presented here much better after seeing some concrete information which the ps command can give you. The format of this command's options varies somewhat from machine to machine (on Linux, I recommend **ps ax**), so check the **man** page for details, but run it with enough options that you get the fullest output possible.

Another command to try is **w.** One of the pieces of information given by the **w** command is the average number of processes in Run state in the last few minutes. The larger this number is, the slower will be the response time of the machine as perceived by a user, as his program is now taking turns together with more programs run by other people.

On Linux (and some other UNIX) systems, you can also try the **pstree** command, which graphically shows the "family tree" (ancestry relations) of each process. For example, here is the output I got by running it on one of our CSIF PCs:

```
% pstree
init-+-atd
     |-crond
     |-gpm
     |-inetd---in.rlogind---tcsh---pstree
     |-kdm-+-X
     |     `-kdm---wmaker-+-gnome-terminal-+-gnome-pty-helpe
     |                    |                `-tcsh-+-netscape-commun---netscape-+
     |                    |                       `-vi
     |                    |-2*[gnome-terminal-+-gnome-pty-helpe]
     |                    |                   `-tcsh]
     |                    |-gnome-terminal-+-gnome-pty-helpe
     |                    |                `-tcsh---vi
     |                    `-wmclock
     |-kerneld
     |-kflushd
     |-klogd
     |-kswapd
     |-lpd
     |-6*[mingetty]
     |-2*[netscape-commun---netscape-commun]
     |-4*[nfsiod]
     |-portmap
     |-rpc.rusersd
     |-rwhod
     |-sendmail
     |-sshd
     |-syslogd
     |-update
     |-xconsole
     |-xntpd
     `-ypbind---ypbind
```

```
%
```

On UNIX systems, the first thing the OS does after bootup is to start a process named **init**, which will be the parent (or grandparent, great-grandparent, etc.) of all processes. That **init** process then starts several OS **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.[20] 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**[21], 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**, 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:

- 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.

- Relieve the compiler and linker of having to know what memory is free when a program is run:

  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.

- 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

[20] Another common print server daemon is **cupsd**.
[21] But no longer allowed on CSIF.

- Enable sharing:

  We want to be able to have only one copy in memory of the "text" portion of a large program (e.g. a compiler) even though several users are each running instances of the program

5.2 Example of 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 (e.g. from the compiler's point of view) that the program is loaded starting at location 0 in memory, but that will not be the case in actuality.

(For the time being, it will be easier to understand VM by assuming there is no cache. We will return to this in Section 5.7.)

To make this more concrete, suppose our C source file from which we compiled **a.out** included a statement

```
int x;
```

and suppose the compiler and linker had assigned the address 200 to x. In other words, a statement in our C source file like

```
printf("%d",&x);
```

would print out the value 200.

Then **a.out** might have instructions like

```
movl 200,%eax
```

on an Intel machine. This instruction copies the contents of word 200 of memory to the CPU register EAX.

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** 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.[22] The OS sets up a **page table**, which is an array which is maintained by the OS, in which the OS records the correspondences, i.e. lists which page of the program is stored in which page of memory.[23]

So, 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

[22] 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.

[23] Remember, the OS is a program. The page table is an array in that program.

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 read from that location.

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

5.3 Overview of How the Goals Are Achieved

Let's look at our stated goals in Section 5.1 above, and see how they are achieved:

- Overcome 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, 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 page is nonresident (this is called a "page fault"), and cause an internal interrupt.[24] 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.

  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. This is beyond the scope of this document here, but one point to notice is that the algorithm will be chosen so as to work well on "most" programs. For some programs, that algorithm will result in a lot of page faults, due to it frequently being the case that right after a page is replaced, that same page is needed again.

- Relieve the compiler and linker of the burden of knowing what memory is free at the time the program executes:

  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.

- Enable 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, of 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 the program tries to access a page for which it does not have the proper permission, i.e. an **access violation** occurs, the VM hardware in the CPU will cause an internal interrupt (again it's interrupt

[24] The interrupt number is 0xe, i.e. entry 0xe in the IDT.

number 0xe, as for page faults), causing the OS to run.[25] The OS will then kill the process, i.e. remove it from the process table.

- Enable sharing:

  Suppose for example two users of a given machine wish to run **gcc** right now. This is a huge program, so conserving memory is quite important. An obvious strategy on the part of the OS would be to load only one copy of the text (i.e. instructions) part of **gcc**; of course, there must be separate copies of the data sections, as the data (.c source code files) for the two users are different.

  VM allows us to accomplish this. Each user's page table will list the same entries for the text, and thus they will share the text part of the program.

5.4 Creation and Maintenance of the Page Table

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 addresses, check page residency and check security.

When the OS creates a new process, it must find chunks (pages) of memory into which it will load part or all of the given program. It will create a page table for this process, and record in the page table the locations of these chunks (as well as record the locations on disk of the chunks which it did not load into memory).

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.[26]

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.

The address space is broken into pages. For convenience, say the page size is 4096 bytes. 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. Since $4096 = 2^{12}$, that means that in a 32-bit virtual address, the upper 20 bits of an address form the page number, and the lower 12 bits form the offset.[27]

[25] 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, which would be desirable.

[26] 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.

[27] You may wish to think of an offset within page is being similar to a phone number within area code.

Consider for example the Intel instruction

```
movl $3, 0x735bca62
```

This would copy the constant 3 to location 0x735bca62 (1935395426 base-10). That means virtual page number 0x735bc (472508 base-10), offset 0xa62 (2658 base-10) within that page. In other words, the first byte of the word we will write to is byte 2658 within page 472508 in the virtual address space.

Suppose the entries in our page table are 32 bits wide, i.e. one word per entry.[28] 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
- 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
- Bits 7-0: other information, not discussed here

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

```
movl $3, 0x735bca62
```

above:

- The CPU, seeing that this is virtual page number 0x735bc, will go to get that entry in the page table, as follows. Suppose the contents of the PTR is 0x256a1000. Then the table entry of interest here is at location 0x735bc * 4 + 0x256a1000 = 0x2586e6f0. The CPU will read from that location, getting, say, 0xc2248eac.
- The CPU looks at Bits 11-8 of that entry, getting 0xe, finding that the page is resident and that the program has read and write permission but not execute permission. The permission requested was write, so this is OK.

Consider for example my office phone number (530) 752-1953. All those parentheses, hyphens and spaces are there just for clarity and are actually not part of the number. So, my number is actually 5307521953. (We could even say my number is 0015307521953, with the 001 being the country code for USA, but let's ignore that to keep it simple.) Then 7521953 is my phone number within my area code, 530, just like a memory location has a page offset within a page. So my full phone number, 5307521953, is analogous to the memory location's full address, which is is page number concatenated with its page offset.

[28] 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.

- The CPU looks at Bits 31-12, getting 0xc2248. The virtual offset, which we found earlier to be 0xa62, is always retained, so the CPU now knows that the physical address of the virtual location 0x735bca62 is 0xc2248a62. The CPU puts the latter in the Memory Address Register (MAR), puts 3 in the Memory Data Register (MDR), and asserts the Write line in the bus. This writes 3 to memory location 0xc2248a62, 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 concatenated with the offset to form the physical address; and the physical address would be placed in the MAR and the instruction fetched.

5.5.2 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. This event is known as a **page fault**. If that occurs, the CPU will perform an internal interrupt,[29] which will force a jump to the OS. The OS will first decide which currently-resident page to replace,[30] then write that page back to disk.[31] The OS would 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 indicating the page is not resident, and changing Bits 31-12; 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, and show where it resides.

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.

5.5.3 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:

```
int q[200];
```

[29] The CPU 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.

[30] To do this, it will use a Least Recently Used policy similar to those common in associative memory caches.

[31] While we will not assume so here, most OSs will do this write-back only if it is necessary. One of the bits in our field of Bits 7-0 above would be used as the **Dirty Bit** for this purpose. We will not pursue this aspect here.

```
main()

{  int i;

   for (i = 0; i < 2000; i++)~ {
      q[i] = i;
   }

}
```

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,[32] 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. 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]**.[33] 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, consider the following code:

```
1  int f(int x)
2  {
3     return x*x;
```

[32] Recall that "VM platform" requires both that our CPU has VM capability, and that our OS uses this capability.

[33] 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.

```
4  }
5
6  int (*p)(int);
7
8  main()
9  {
10    p = f;
11    u = (*p)(5);
12    printf("%d\n",u);
13 }
```

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 Improving Performance

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.

5.7 What Role Do Caches Play in VM Systems?

Up to now, we have been assuming that the machine doesn't have a cache.[34] 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, before the processing of the cache miss can occur.

Another issue 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,[35] and then do the cache lookup based on that address.

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.

[34] Except for a TLB, which is of course very specialized.

[35] By checking the TLB, and then the page table if need be.

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,[36] running on the same machine, using the same 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 **ps** 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.

5.8 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
```

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.[37] 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. The call to **write()** (which your program could also make directly) is a system call.

Recall, though, that we want to arrange things so that only the OS will be able to perform the actual I/O operations, with instructions like Intel's **in** and **out**. So the hardware is designed so that these instructions can be executed only in Kernel Mode.

[36] Or, Linux versus Windows, etc.

[37] You should not jump to the conclusion that all, or almost all, of the services deal with I/O. For example, 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.

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. In other words, the call to **write()** in your C program (or in printf()) will be translated to

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

The **int** instruction works like a hardware interrupt, in te 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.[38]

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

As an example, the following Intel Linux assembly language program writes the string "ABC
n" to the screen, and then exits:

```
.data
hi:  .string "ABC\n"

.text
_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
```

[38] 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.

```
# 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).

Here are some examples of the numbers (to be placed in EAX before **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.[39]

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.

[39] However, the file contents are still there. This is how "undelete" programs work, by attempting to recover the sectors liberated when the user (accidentally) deleted the file.