1 Computer Systems Overview(i.e. stuff you should already know)

Chap. 1

Operating Systems run on top of the bare metal of the computer. In fact, a good working definition of an OS is the collection of software components that interface directly with the hardware. So before we get into the nitty-gritty of Operating Systems, let's review the stuff you should already know.

1.1 The computer as a flow chart

ch. 1.1 and 1.2

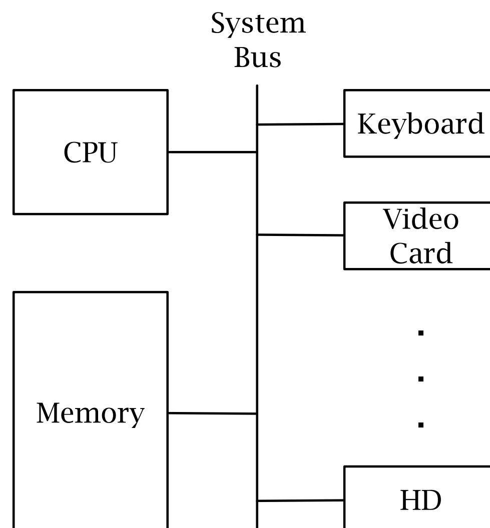

Figure 1: The Computer as a Flowchart

A computer can be simplified down to a little flow chart, where you have the central processor (CPU), a big pile of memory (RAM, usually) and a collection of devices (these are usually known as Input/Output (I/O) devices). These three components are all connected by a big batch of wires known as the system bus. Of course, I'm glossing over a tremendous amount of detail here, but most computers can be reduced to this simple picture. The CPU is where all the computation happens, it's where your programs actually run, and it's also where the operating system is doing its thing.

Main memory is a large, usually volatile, collection of bits where all the data and code is placed. The CPU itself has a limited collection of extremely fast storage locations (registers), and so anything larger than a few 10's of bytes has to reside in main memory. This size comes at a cost,

namely speed. Main memory is at least an order of magnitude slower than registers. Also, because main memory is so large, data has to be referenced by its address in memory.

I/O devices are everything else. Your keyboard, your hard drive, your web-cam, etc. Although these devices can basically be anything, they all more-or-less integrate with the generic bus protocol (whatever that happens to be). In this way, all interactions with I/O devices become simply passing bits along the bus between the CPU and the device. So even though when the keyboard sends the bit pattern '01100001' to the CPU, it means that the user just pushed the 'a' key; and when the video card sends the bit pattern '01000010' to the CPU it means that the screen just refreshed, all the bus sees is a stream of bits that it has to manage. Because the bus is so simple, and merely makes sure that bits get where they're going, the CPU has to make all the complicated decisions about how to interpret and react to device messages. This is, in fact, a large part of the responsibilities of an Operating System (but more on that later).

1.2 Fetch, Decode, Execute Model

ch. 1.3

For our purposes, it is sufficient to reduce all the superscalar, out-of-order microarchitectural wizardry of modern computers to the ancient fetch-decode-execute model. In this model, the processor first grabs an instruction from memory (at the address specified in the program counter (PC)) , looks at it to see what to do (decode), and then perform the operation specified by the instruction (execute). When the CPU's dome with that instruction, it goes out and fetches another one. This model is sufficient to explain how a user program executes, however to explain an OS, it needs to be extended with I/O operations. So, as the CPU is merily fetching and executing some user code, an I/O device decides it needs some attention (the user moved the mouse, say). The CPU now has to deal with this device. The CPU has to stop executing the user code, and switch over to run code (OS code) that will handle the device request (this, by the way is actually what a device driver is). This code, however is also run under the fetch-decode-execute model! So, from the point of view of the processor, an OS is just another program! A special program, to be sure, but another program nonetheless.

1.2.1 Example 1

In this example, a single user process is executing and makes a call to to OS to do some I/O (output a string to the console, in this case).

```
---User Mode----
inst1
inst2
inst3
inst4
call OS_Write //this is actually usually a trap or interrupt
---Supervisor Mode----
-save process context (registers, process struct, etc.)
```

```
-enter OS context
os_inst1
os_inst2
os_inst3
...
os_instN
-copy return result into user's memory
-restore user's context
---User Mode---
-enter User context
inst5
inst6
...
instN
call OS_Exit
```

Even though the I/O request originates in user space, the Operating System has to perform a context switch to ensure the integrity of the user's (and OS's) code and data. This is an example of what is often called a system call or software interrupt. It is a request by user software for OS intervention. Usually this is accomplished by a special user-mode instruction (INT on intel, trap on sparc, SWI on ARM, etc.). This instruction effectively 'wakes up' the OS and gets it to perform a specific function. It functions very similarly to a function call, just with lots more overhead. Note also, that in modern OSes, most I/O system calls will cause the user process to be swapped out with a runnable process, and won't be rescheduled until the I/O is complete.

1.2.2 Example 2

In this example, an external event (not a system call) triggers the operating system, which is running two processes: process 1 and process 2 (how's that for creative naming :).

```
---User Mode---
proc1_inst1
proc1_inst2
proc1_inst3
---Timer Interrupt Fires--- //Time to swap the processes
---Supervisor Mode---
-save process 1 context
-enter OS context
os_inst1
os_inst2
os_inst3
...
os_instN
```

```
-set Timer
-restore process 2 context
---User Mode---
-enter process 2 context
proc2_inst1
proc2_inst2
proc2_inst3
---Timer Interrupt Fires
---Supervisor Mode---
-save process 2 context
-enter OS context
os_inst1
os_inst2
os_inst3
...
os_instN
-set Timer
-restore process 1 context
---User Mode---
-enter process 1 context
proc1_inst4
proc2_inst5
...
```

In this case, the OS was responding to the timer interrupt which was set by the OS to manage time slices. In this highly contrived case, the slices were about 3 cycles big (usually, they're on the order of milliseconds). By swapping the process' context, the OS preserves the illusion that each process has the machine to itself. For instance, process 1 has no way of determining that it was just swapped out so that process 2 could run. Modern OSes depend upon specific hardware support to efficiently swap processes around. Namely, they need decent interrupt support to alert the OS when an event has occured, and they need the processor to make a distinction between User and Supervisor (or OS) Mode. The way each processor does this is often different. Some processors have seperate registers available only in supervisor mode, some processors have special mechanisms (like register windows) for saving user context. Some have none of these and the OS developer must hand code assembly to save and restore user hardware state.

1.3 I/O Techniques

Chap 1.7

A large part of an OS's resposibilities are to manage I/O devices. If user code wants to communicate with a device, this must be mediated by the OS. Likewise, if a device needs to communicate

with user code (a hard drive finished reading a chunk of data, for instance), the OS must be alerted and take care of it. Over the years, several different techniques have been employed.

1.3.1 Programmed I/O

By far the simplest and oldest technique. In this model, the OS has to periodically check all the I/O devices to see if their status has changed (indicating that some action was performed). In a programmed I/O system, all user code is suspended at regular intervals so the OS can make sure that there aren't any pending requests from I/O devices. Repeatedly looping and checking for a change in state is a technique known as polling. If the state changes infrequently (at least in terms of the time it takes to check the device), then polling is inefficient. However, on hardware without facilities to signal the CPU from I/O devices (as was the case with early computers) there's really no other choice.

1.3.2 Interrupt-driven I/O

In hardware, an interrupt is a signal sent from one device to another. It's basically just letting the processor know that 'Hey, something just happened'. Modern hardware supports interrupts on the silicon, so that when a device signals the processor, an interrupt is raised and supervisor code is executed. Interrupts basically 'wake up' the OS (or, more appropriately, interrupt the code that was executing), and start the processor executing special-purpose code to handle the interrupt. Compared to polling, interrupt-driven I/O is more efficient. Rather than having to put everything on hold to go check each and every device, the OS can go about its business, running user code and then get told when an I/O device needs attention. This is more efficient than programmed I/O, but it is trickier to code around. Now, the OS designer has no idea when an interrupt may arrive. If it happens while running user code, then that would probably look like a normal context switch. But what happens if an interrupt is fired while in the middle of OS code? What if an interrupt fires while the OS is busy processing an older interrupt?

1.3.3 DMA

Many of you may have seen the acronym, DMA. Some of you may even know that it stands for Direct Memory Access. But, what actually is DMA? To understand DMA, first you must realize that a large part of I/O processing is moving bits around. For instance, a user opens a file on disk, the OS goes and finds the bits on the hard drive, and then moves them (usually in ridiculously small chunks, like 64 bits) in small chunks into some area in memory where the user code can get at them. For any decent sized file (kilobytes or larger), the time spent moving the bits will be much greater than the time spent allocating memory and locating the files on disk. DMA is a hardware feature that allows the OS to tell the bus to move bits from an I/O device into a specific place in memory. So rather than the OS having to manually move a few bytes at a time from device x to memory, the OS just tells the bus 'Hey, move 1024 bytes from device x into memory starting at address y', and the bus will do this on its own. This frees up the OS to run more user code. But, like interrupts, this increase in efficiency comes at the cost of increased complexity. What happens

if the DMA transfer fails? What happens if the OS needs to start a second transfer while another transfer is still going on? Of course, the answers to these questions are dependant on both the hardware and what feature the OS is going to support.

2 Memory: The Stack and the Heap

Memory at the hardware level basically just a big bag of bits. However, for programmer convenience it is customary to divide memory into two pieces. The stack, which carries all the function call information and the local variables, and the heap which carries all the dynamic data.

2.1 The Stack

The stack is a LIFO data structure, in memory, that is used to save function invocation information and local variables. In general, programmers (even C programmers) don't mess around directly with the stack. The compiler does a nice job of turning function calls and returns into stack operations. However, in C, it is possible for a called function to return a reference to local data. This is BAD. A general rule is only return pointers to HEAP data (that is, data allocated with `malloc`)

2.2 The Heap

The heap is basically the rest of memory. This is where the programmer allocates more dynamic data. In Java, whenever you called `new` (as in `List ls = new LinkedList();`), you were using the heap. In C, the allocation function is named `malloc` (and the deallocation routine is named `free`). As far as the hardware is concerned, it's all just bytes, so it's up to the OS and the programming language runtime to structure the data more conveniently. The OS structures memory into fixed size chunks called pages (usually with some hardware support). Your average page is 4 or 8 kilobytes in size. Then it's up to the programming language runtime to figure out how to stick programmer allocated data into these pages. In Java, a garbage collector is constantly running in the background to deallocate unused objects and compact live objects into a smaller chunk of memory. In C, the programmer has to deallocate by hand, but there's still a fair amount of jiggery pokery under the hood to lay out allocations efficiently. If you take nothing else away from this review of memory, you should remember that in C stack data (local variables) are allocated and deallocated automatically and heap data has to be allocated and deallocated by hand.