CS 111

Scribe Notes for 4/6/05

by Conrad Chan, Derek Chu, Amer Marji & James Szeto

Overview

The purpose of this lecture is to build up the modern operating system with its different components, starting with a very simple operating system.

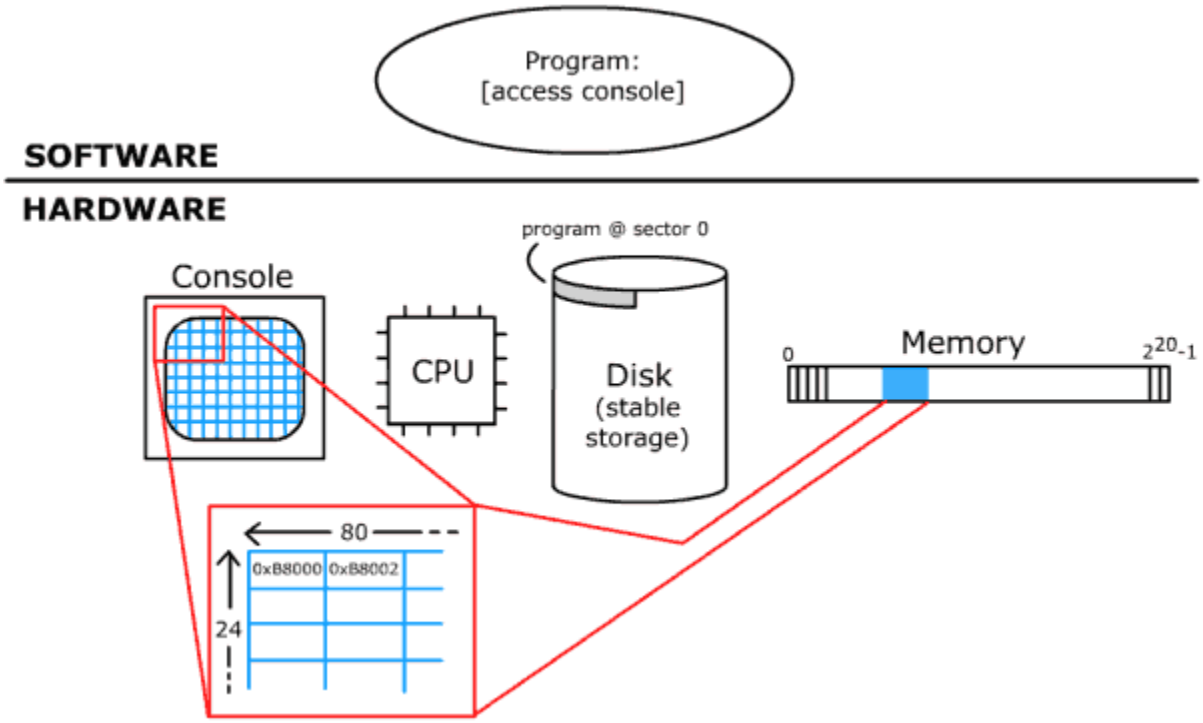

Fig. 2-1: Device access uses memory & programmed I/O.

The "Print 111" Operating System

Let's begin with the "Print 111" operating system. A machine running this simple operating system will:

1) turn on, and
2) print "111" to the console.

This is a fairly simple task, so how should we go about doing this? Well, we know that the console is broken up into many sections and each section's position is mapped in memory. So, with a bit of assembly language code, we can literally tell the console to write the corresponding output to the appropriate position on the console, such as this:

```
movw $49, 0xB8000
movw $49, 0xB8002
movw $49, 0xB8004
I: jmp I    // after print "111" go to infinite loop
```

Now that we know this is what this simple program acting as an operating system will do, how do we go about getting this "Print 111" program loaded on the computer so that when the machine turns on, this code will be run? We could put the program in the **BIOS**, a region of stable memory that is read-only. That could make sense, because when the computer turns on it is hardwired to immediately jump to this memory location and would then run this code. But then if this memory for the BIOS is read-only, that means if we ever want to change the software (as if there were any other need besides printing "111" to a screen) we would have to scrap the entire computer and build a new one with different code written on that read-only BIOS memory. (If we're lucky, the machine would have an updatable BIOS, but still, changing the BIOS is difficult and can really screw up the machine!)

What else can we do then? Well, instead, let's put the program on the disk at a known location. The BIOS code will simply load that known part of the disk into memory, then jump to it. In fact the BIOS reads the first sector of the disk (offset 0), and writes it to address 0x7C00. But this raises the issue of knowing how to access the disk. To solve this problem, the ISA uses a specified block of memory, addresses 0x1F0-0x1F7, that can use the special instructions inb/outb, as demonstrated in the following code:

```
while ((inb(0x1F7) & 0xC0) != 0x40)
    /* spin */;
outb(0x1F2,1);     // number of sectors to read
outb(0x1F3,0);     // bits 0-7 (low bits) of 28-bit offset
outb(0x1F4,0);     // bits 8-15 of 28-bit offset
outb(0x1F5,0);     // bits 16-23 of 28-bit offset
outb(0x1F6,0xE0);  // bits 24-27 of 28-bit offset
                   // bit 28 (= 0) means Disk 0
                   // other bits (29-31) must be set to one
outb(0x1F7,0x20);  // READ SECTORS command
while ((inb(0x1F7) & 0xC0) != 0x40)
    /* spin */;
insl(0x1F0,0x7C00,128); // get results as 128 "long words"
                    // 1 long word == 4 bytes; 128 * 4 == 512 bytes,
                   // the length of a sector
```

The clause in the while statement basically checks to see if the disk is ready. Once it is ready, we use the outb instructions to write commands to the disk and then the last line, the insl instruction, reads the result back. Using these instructions, the operating system can manage the hardware.

The "Password Checking" Operating System

We now try to implement a slightly more complex operating system, the "Password Checking" OS. We want this "operating system" to accomplish two tasks:

1) print the password, and

2) check the password.

Let's choose an arbitrary password, let's say we use our most valuable string "111" again. This following code represents program P, the printing of the password:

```
int main(int c, char* v[]) {    // program P
    printf("111\n");
    return 0;
}
```

And the following code represents program C, the checking of the password:

```
int main() {    // program C
    char buf[1000];
    fgets(buf,1000,stdin);
    if(strcmp(buf,"111")==0)
        printf("Y");
    else
        printf("N");
}
```

Processes

We see here that this operating system is performing two tasks: printing the password and checking the password. Now we ask the question, is there a way to optimize the operating system by interweaving the two programs? To answer that question, we introduce the concept of **processes** to the operating system. So how do we interweave the two? Let's take a close look at what's really going on. We see that when P prints "111", what it really does is print one character at a time:

Print("1"), Print("1"), Print("1"), Print("\n")

At the same time, what C is really doing is walking along an array that reads in the input and checks each character, one by one. So we see that we can switch back and forth between the two tasks as each character is being printed or read. We accomplish this implementation of processes by having two execution pointers, a program counter, registers, and stack space (memory to hold local variables). In order to switch between processes the operating system must:
1) save the current state of the program being executed, and
2) load the next state of the other program to be executed.
(in x86: pushflags, popflags, iret).

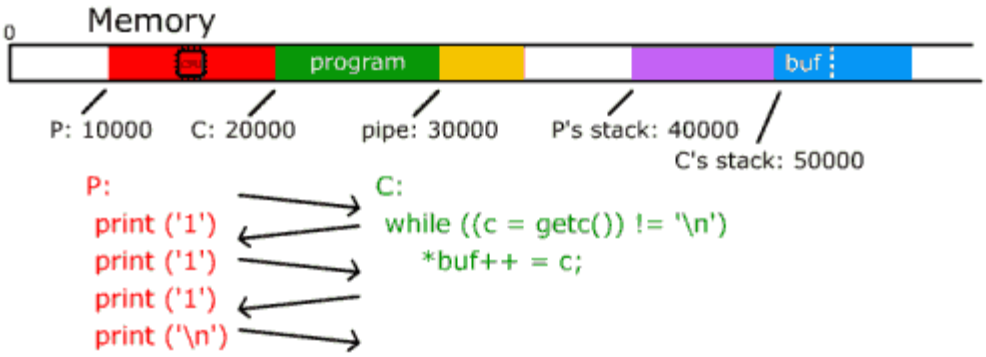

Fig. 2-2: Interweaving programs P and C.

Protection

Now we have two tasks concurrently running and performing the task of inputting a password and checking for correctness. So here we raise the issue of **memory protection**. It would be really easy for task P ("Print 111") to hack into task C ("Check 111") and obtain the password and get permissions to the system if there were no protection. This can be done in multiple ways. For example, if the password was saved in some memory location or on the disk, task P could read this password and then supply it as input for C. Another way is for P to change some instructions in C by going straight to the saved code and change some conditions in the "if" statement.

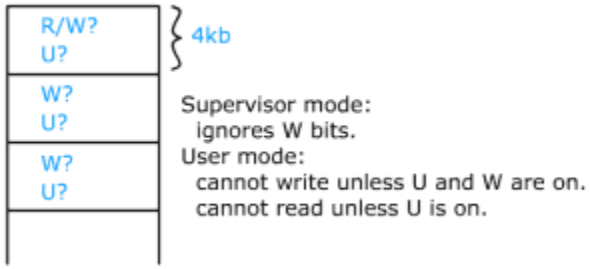

Fig. 2-3: An x86 Page Table.

So here comes the role of memory protection, which is to protect a process's code, data, stack, and registers from unauthorized intrusion. In modern x86 operating systems, memory protection is maintained through **protection flags** on the processor's **page table**, and through a global **privilege level** that says how privileged the currently running process is. For example, a particular chunk of memory might be labeled "readable, but only to supervisor processes (= most privileged)", or "read/write for user processes (= least privileged)". See Fig. 2-3.

There are 4 levels of privilege that go between 0 and 3, 0 being the most privileged level (supervisor) and 3 being the least (user). These privilege levels are represented in a ring structure.

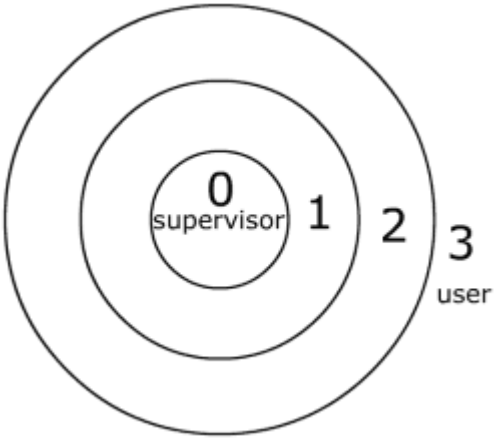

Fig. 2-4: Ring structure of authority levels (supervisor mode --> user mode).

The Kernel & Memory Protection

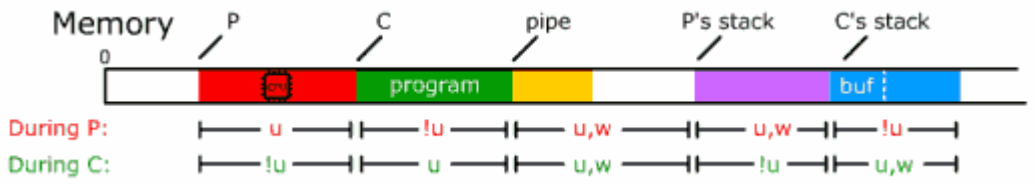

Fig. 2-5: Protection of P and C.

The final portion of the lecture deals with the kernel's role in memory protection. With so much risk involved in programs and memory protection, the **kernel** is in fact the only program that is allowed to run in supervisor mode. This policy maintains safe access to files and programs. The kernel is the only program that starts and maintains user processes.

This seems like a fairly simple idea but how exactly is this carried out? User processes, which are running in user mode, obviously need to call in to the kernel from time to time, to get access to files and such. But user processes can't be allowed to jump into the kernel code at an arbitrary place, and they can't be allowed to raise their own privilege! (It would be a massive security hole if unprivileged code could raise its privilege whenever it wanted.) In fact, the architecture ensures that processes never lower their own privilege level (they can only raise it). So how do we do this? The answer is with traps. **Traps** are user mode function calls that transfer control to the kernel. It is very similar to interrupts that we have studied in CS33. When a process needs to call into the system, it executes a special kind of interrupt to ask for attention from the kernel. In fact, in the x86 machine, the command from user to kernel is "int" while the command from kernel to user is "iret".

Now we turn our attention back to a question we asked earlier: can we still access the disk? The answer is obviously yes, but we do so via the kernel. We first consider raw disk access. Although raw disk access has very advantages speed characteristics, the problems outweigh the quickness of file access.

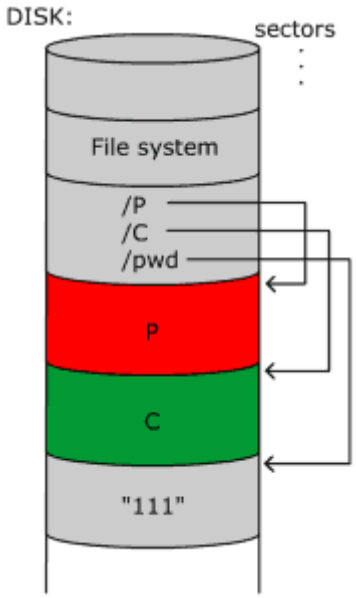

Fig. 2-6: Illustration of disk access.

Raw disk access has problems with **concurrency, protection, safety, and programming**. The users to kernel API for disks and file systems have commands in which the kernel manages the file systems interface. These commands are read(), write(), open() and unlink(). The first three are pretty intuitive. Unlink() is another name for deletion.

This is all the time we had for lecture. The main goals for Wednesday's lectures the following:

- Understanding that device access uses memory and programmed I/O
- The concept of processes
- The importance of memory protection, privileged instructions, and the kernel and traps.
- Abstraction

We will continue exploring these topics and others in future lectures