Scribe: Anthony Pimentel

Lecture2 April 4, 2013

Computing is difficult because of the scale of **complexity**
Complexity is our biggest problem in operating systems

Definitions:

Moore's Law – the complexity of mainstream chips (the number of transistors on a chip), at minimum cost, doubles every two years (exponential growth).
Natural question: When does this exponential growth crash? We can only fit so many transistors on a chip. *No definite answer
Kryder's Law – just like Moore's, but the complexity is defined as the number of bytes you can put on a hard drive and the growth is a little faster.
Note that NEITHER law talks about **speed**, only **complexity**

1st Univac was designed very conservatively
Then they used it to simulate the Univac II.
The point: using the earlier computers to simulate the next versions of computers makes creating better ones easier (exponential growth).
d(technology)/dt = K*technology

Why would you want to simplify operating systems?
Simplicity gives you **accuracy** (correctness), but sacrifices some speed.
Groups like security agencies (or other paranoid groups) want this instead of Windows, GNU/Linux, etc. because those systems have so many lines that there might be security bugs.
We need a sample application to figure out some minimum requirements of a system.

Example: count the number of words in my text file.

Assume we have a desktop computer (1 TB hard-drive) with only a monitor that has the (null-terminated) text file already saved in it at a predetermined location.
We also need to know that the machine can run as x86 or x86-64 and we choose x86
Think about how booting is going to work for the future.
UI: turn the computer on, then the answer appears on the screen.
A bus connects the CPU, RAM (which consists of all 0s on startup), the disk controller, and the display.
Bootstrapping problem: we want our program in RAM, but we'd need a program to put it into RAM!
(Technically impractical and inflexible) Solution: use ROM (read-only memory) so that the initial instruction pointer in the CPU points into ROM and put our program in ROM
Try a new plan using the standard ROM.

Standard ROM contains BIOS (Basic Input/ Output System), which doesn't change, hardwired into ROM. The instruction pointer initially points into BIOS and BIOS tests the computer, looks for devices, and tries to find a replacement (looks for a device that is bootable and boots it).
How to determine if a device is bootable:
Convention: if the 1st sector of the device looks like a Master Boot Record (MBR)

Example MBR layout for x86:

512 bytes long.
First 446 bytes contain machine code (any x86 code you want) and some data if you want.
The last two bytes are 0x55 and 0xAA (real values don't really matter. Just know that there are some predetermined values).
Middle 64 bytes are split into 4 16-byte values that describe how to split up the disk. Those 16-byte values tell you the sector number of the partition start (4 bytes), the number of sectors in the partition (4 bytes), the status of the partition (bootable, for example), the type (1-byte quantity)

BIOS assesses the MBR, and if it is bootable, it reads the MBR into 0x7c00 in RAM and points the CPU instruction pointer to that.
New solution: we could put the wordcount program instructions into the MBR. This would work, but unfortunately, our program is bigger than 446 bytes.
New new solution: put a tiny program (called a boot loader) into MBR that copies the wordcount program from disk to RAM, then jumps to the wordcount program.
BIOS loads boot loader, then boot loader loads wc (wordcount). This is called **chained loading**.

On Ubuntu system, BIOS loads MBR boot loader, MBR loads VBR (volume boot record), VBR loads GRUB (Grand Unified Boot Loader), GRUB loads Linus kernel, Linux kernel loads 'init' (Process ID 1), init loads some other programs.

Example boot loader source code:

```
int main(void) {
   for( int i = 1; i <=20; i++ ) //20 is the number of sectors for the program
      read_sector( i, 0x100000 + ( i – 1 ) * 512 ); //0x100000 is the location where we copied wc
   goto *0x100000
}
```

x86 instructions to communicate to the disk controller:
inb (in-byte = read from a register on bus)
outb (out-byte = write to a register on bus)
insl (like in-byte, but grabs a bunch of data at once)
The disk controller has registers that can tell you things about it like its status (0x1f7)

```
void read_sector( int s, int ptr_t a ) {
   wait_for_disk();
   outb( 0x1f2, 1 ); //1f2 keeps track of the number of sectors. 1 is the number of sectors
   outb( 0x1f3, s & 0xff );
   outb( 0x1f4, ( s >> 8 ) & 0xff );
   outb( 0x1f5, ( s >> 16 ) & 0xff );
```

```
    outb( 0x1f6, ( s >> 24 ) & 0xff );
    outb( 0x1f7, 0x20 ); //writing to 0x1f7 gives it a command. 0x20 is the read_sectors command
    wait_for_disk(); //after this, the data is ready
    insl( 0x1f0, a, 128 )
}
void wait_for_disk(void) {
    while( ( inb(0x1f7) & 0xc0 ) != 0x40 ) //just look at the first two bits to see if it is busy
        continue;
}

int main(void) {
    int nwords = 0;
    bool inword = 0;
    int s = 5000000000/512 //s is the sector. 5 billion is the address we put the file. 512 is the
sector size
    for( ; ; s++ ) {
        char buf[512];
        read_sector(s,(int ptr_t)buf);
        for( int j=0; j < 512; j++ ) {
            if( buf[j] ) {
                write_out(nwords);
                return;
            }
            char uppercase = buf[j] & ~('a'-'A')
            bool thisalpha = 'A'<= uppercase && uppercase <= 'Z'
            nwords += ~inword & thisalpha
            inword = thisalpha
        }
    }
}

void write_out( int nwords ) {
    char *p = (char*) 0xb8000 + 24 * 80; // address is where the computer display is. 24 * 80 to
center it
    do {
        *--p = nwords%10 + '0'; //--p means decrement p
        *--p = 7;  //gray on black
    } while( ( nwords /= 10 ) != 0 );
}
```