DRAFT as of September 20, 2009: Copyright 2009 Cox, Kaashoek, Morris

Chapter 1

Bootstrap

Hardware

A computer's CPU (central processing unit, or processor) runs a conceptually simple loop: it inspects the value of a register called the program counter, reads a machine instruction from that address in memory, advances the program counter past the instuction, and executes the instruction. Repeat. If the execution of the instruction does not modify the program counter, this simple loop will interpret the memory pointed at by the program counter as a simple sequence of machine instructions to run one after the other. Instructions that do change the program counter implement conditional branches, unconditional branches, and function calls.

The execution engine is useless without the ability to store and modify program data. The simplest, fastest storage for data is provided by the processor's register set. A register is a storage cell inside the processor itself, capable of holding a machine word-sized value (typically 16, 32, or 64 bits). Data stored in registers can typically be read or written quickly, in a single CPU cycle. The x86 provides eight general purpose 32-bit registers—`%eax`, `%ebx`, `%ecx`, `%edx`, `%edi`, `%esi`, `%ebp`, and `%esp`—and a program counter `%eip` (the "instruction pointer"). The common `e` prefix stands for extended, as these are 32-bit extensions of the 16-bit registers `%ax`, `%bx`, `%cx`, `%dx`, `%di`, `%si`, `%bp`, `%sp`, and `%ip`. The two register sets are aliased so that, for example, `%ax` is the bottom half of `%eax`: writing to `%ax` changes the value stored in `%eax` and vice versa. The first four registers also have names for the bottom two 8-bit bytes: `%al` and `%ah` denote the low and high 8 bits of `%ax`; `%bl`, `%bh`, `%cl`, `%ch`, `%dl`, and `%dh` continue the pattern. In addition to these registers, the x86 has eight 80-bit floating-point registers as well as a handful of special-purpose registers like the control registers `%cr0`, `%cr2`, `%cr3`, and `%cr4`; the debug registers `%dr0`, `%dr1`, `%dr2`, and `%dr3`; the segment registers `%cs`, `%ds`, `%es`, `%fs`, `%gs`, and `%ss`; and the global and local descriptor table pseudo-registers `%gdtr` and `%ldtr`. The control, segment selector, and descriptor table registers are important to any operating system, as we will see in this chapter. The floating-point and debug registers are less interesting and not used by xv6.

Registers are very fast but very expensive in bulk. Most processors provide at most a few tens of general-purpose registers. The next conceptual level of storage is the main random-access memory (RAM). Main memory is 10-100x slower than a register, but it is much cheaper, so there can be more of it. A typical x86 processor has at most a kilobyte of registers, but a typical PC today has gigabytes of main memory. Because of the enormous differences in both access speed and size between registers and main memory, most processors, including the x86, store copies of recently-accessed sections of main memory in on-chip cache memory. The cache memory serves as a middle ground between registers and memory both in access time and in size. Today's x86 processors typically have two levels of cache, a small first-level cache with access times relatively close to the processor's clock rate and a larger second-level cache with access times in between the first-level cache and main memory. This table shows actual numbers for an Intel Core 2 Duo system:

Intel Core 2 Duo E7200 at 2.53 GHz

TODO: Plug in non-made-up numbers!

storage	access time	size
register	0.6 ns	64 bytes
L1 cache	0.5 ns	64 kilobytes
L2 cache	10 ns	4 megabytes
main memory	100 ns	4 gigabytes

For the most part, x86 processors hide the cache from the operating system, so we can think of the processor as having just two kinds of storage—registers and memory—and not worry about the distinctions between the different levels of the memory hierarchy. The exceptions—the only reasons an x86 operating system needs to worry about the memory cache—are concurrency (Chapter 4) and device drivers (Chapter 6).

One reason memory access is so much slower than register access is that the memory is a set of chips physically separate from the processor chip. To allow the processor to communicate with the memory, there is a collection of wires, called a bus, running between the two. A simple mental model is that some of the wires, called lines, carry address bits; some carry data bits. To read a value from main memory, the processor sends high or low voltages representing 1 or 0 bits on the address lines and a 1 on the "read" line for a prescribed amount of time and then reads back the value by interpreting the voltages on the data lines. To write a value to main memory, the processor sends appropriate bits on the address and data lines and a 1 on the "write" line for a prescribed amount of time. This model is an accurate description of the earliest x86 chips, but it is a drastic oversimplification of a modern system. Even so, thanks to the processor-centric view the operating system has of the rest of the computer, this simple model suffices to understand a modern operating system. The details of modern I/O buses are the province of computer architecture textbooks.

Processors must communicate not just with memory but with hardware devices too. The x86 processor provides special `in` and `out` instructions that read and write values from device addresses called ports. The hardware implementation of these instructions is essentially the same as reading and writing memory. Early x86 processors had an extra address line: 0 meant read/write from a device port and 1 meant read/write from main memory.

Many computer architectures have no separate device access instructions. Instead the devices have fixed memory addresses and the processor communicates with the device (at the operating system's behest) by reading and writing values at those addresses. In fact, modern x86 architectures use this technique, called memory-mapped

I/O, for most high-speed devices such as network, disk, and graphics controllers. For reasons of backwards compatibility, though, the old `in` and `out` instructions linger, as do legacy hardware devices that use them, such as the IDE disk controller, which we will see shortly.

Bootstrap

When an x86 PC boots, it starts executing a program called the BIOS, which is stored in flash memory on the motherboard. The BIOS's job is to prepare the hardware and then transfer control to the operating system. Specifically, it transfers control to code loaded from the boot sector, the first 512-byte sector of the boot disk. The BIOS loads a copy of that sector into memory at `0x7c00` and then jumps (sets the processor's `%ip`) to that address. When the boot sector begins executing, the processor is simulating an Intel 8088, the core of the original IBM PC released in 1981. The xv6 boot sector's job is to put the processor in a more modern operating mode and then transfer control to the xv6 kernel. In xv6, the boot sector comprises two source files, one written in a combination of 16-bit and 32-bit x86 assembly (`bootasm.S`; (0900)) and one written in C (`bootmain.c`; (1100)). This chapter examines the operation of the xv6 boot sector, from the time the BIOS starts it to the time it transfers control to the kernel proper. The boot sector is a microcosm of the kernel itself: it contains low-level assembly and C code, it manages its own memory, and it even has a device driver, all in under 512 bytes of machine code.

Code: Assembly bootstrap

The first instruction in the boot sector is `cli` (0915), which disables processor interrupts. Interrupts are a way for hardware devices to invoke operating system functions called interrupt handlers. The BIOS is a tiny operating system, and it might have set up its own interrupt handlers as part of the initializing the hardware. But the BIOS isn't running anymore—xv6 is, or will be—so it is no longer appropriate or safe to handle interrupts from hardware devices. When xv6 is ready (in Chapter 3), it will re-enable interrupts.

Remember that the processor is simulating an Intel 8088. The Intel 8088 had eight 16-bit general-purpose registers but 20 wires in its address bus leading to memory, and thus could be connected to 1 megabyte of memory. The segment registers `%cs`, `%ds`, `%es`, and `%ss` provided the additional bits necessary to generate 20-bit memory addresses from 16-bit registers. In fact, they provide more than enough bits: the segment registers are 16 bits wide too. A full memory reference on the 8088 consists of two 16-bit words, a segment and an offset, written *segment*:*offset*. Typically, the segment is taken from a segment register and the offset from a general-purpose register. For example, the `movs` instruction copies data from `%ds`:`%si` to `%es`:`%di`. The 20-bit memory address that went out on the 8088 bus was the segment times 16 plus the offset. We'll call the addresses the processor chip sends to memory "physical addresses," and the addresses that programs directly manipulate "virtual addresses." Thus, on an

8088, a virtual address consists of a 16-bit segment register combined with a 16-bit general-purpose register, for example `0x8765:0x4321`, and translates to a 20-bit physical address sent to the memory chips, in this case `0x87650+0x4321 = 0x8b971`.

PC BIOSes guarantee to copy the boot sector to physical address `0x7c00` and start it executing, but there is no guarantee that they will choose to set `%cs:%ip` to `0x0000:0x7c00`. In fact, some BIOSes use `0x0000:0x7c00` when the boot sector is from a hard disk and use `0x07c0:0x0000` when the boot sector is from a bootable CD or DVD. There are no guarantees at all about the initial contents of the segment registers used for data accesses (`%ds`, `%es`, `%ss`), so first order of business after disabling interrupts is to set `%ax` to zero and then copy that zero into `%ds`, `%es`, and `%ss` (0918-0921).

The address calculation can produce a 21-bit address, but the Intel 8088 could only address 20 bits of memory, so it discarded the top bit: `0xffff0+0xffff = 0x10ffef`, but virtual address `0xffff:0xffff` on the 8088 referred to physical address `0x0ffef`. The Intel 80286 had 24-bit physical addresses and thus could address 16 megabytes of memory, so its real mode did not discard the top bit: virtual address `0xffff:0xffff` on the 80286 referred to physical address `0x10ffef`. The IBM PC AT, IBM's 1984 update to the IBM PC, used an 80286 instead of an 8088, but by then there were programs that depended on the 8088's address truncation and did not run correctly on the 80286.

IBM worked around this incompatibility in hardware: the PC AT design connected the 20th address line of memory (A20) to the logical AND of the 20th address line coming out of the 80286 processor and the second bit of the keyboard controller's output port. When the PC AT booted, the keyboard output port's second bit was zero, making the memory controller always see zero on the A20 line, which in turn made the 80286's memory accesses behave like an 8088, so that 8088 programs would run correctly. Of course, IBM wanted to allow new programs to take advantage of the expanded memory. PC AT-specific software instructed the keyboard controller to change the output port bit to a 1, allowing the 80286's A20 values to pass unfiltered to the memory controller. To this day, modern PCs continue this backwards compatibility dance, and low-level software probably continues to depend on 8088 behavior at boot. The boot sector must enable the A20 line using I/O to the keyboard controller on ports 0x64 and 0x60 (0923-0941).

The 8088 had 16-bit general-purpose registers, so that a program that wanted to use more than 65,536 bytes of memory required awkward manipulation of segment registers. The 8088's 20-bit physical addresses also limited the total amount of RAM to a size that seems small today. Modern software expects to be able to use tens to thousands of megabytes of memory, and expects to be able to do it without fiddling with segment registers. The minimum modern expectation is that a processor should have 32-bit registers that can be used directly as addresses. The Intel x86 architecture arrived at those capabilities in two stages of evolution. The 80286 introduced "protected mode" which allowed the segmentation scheme to generate physical addresses with as many bits as required. The 80386 introduced "32-bit mode" which replaced the 80286's 16-bit registers with 32-bit registers. The xv6 boot sequence enables both modes as follows.

In protected mode, a segment register is not a simple base memory address anymore. Instead, it is an index into a segment descriptor table. Each table entry specifies a base physical address, a maximum virtual address called the limit, and permission bits for the segment. These additions are the protection in protected mode: they can be used to make sure that one program cannot access memory belonging to another program (including the operating system itself). Chapter 2 will put the protection features to good use; the boot sector simply wants access to more than 20 bits of memory. It executes an `lgdt` instruction (0954) to set the processor's global descriptor table (GDT) register with the value `gdtdesc` (0995-0997), which in turns points at the table `gdt` (0990-0993).

This simple GDT has three entries: the processor requires entry 0 to be a null entry; entry 1 is a 32-bit code segment with offset 0 and limit 0xffffffff, allowing access to all of physical memory; and entry 2 is a data segment with the same offset and limit. "32-bit code segment" enables the 80386's 32-bit mode, so that the processor will default to 32-bit registers, addresses, and arithmetic when executing in the segment. In protected mode, the bottom two bits of a segment register give the processor's privilege level (0 is kernel, 3 is user mode, 1 and 2 are intermediate). The next bit selects between the global descriptor table (0) and a second table called the local descriptor table (1). The rest of the bits are an index into the given table. Thus 0x8 and 0x10 refer to GDT entries 1 and 2 with kernel privilege level. Those entries are the code and data segments the boot sector will use in protected mode. The code refers to these numbers using the aliases `SEG_KCODE` and `SEG_KDATA` (0907-0908).

Once it has loaded the GDT register, the boot sector enables protected mode, by setting the 1 bit (`CR0_PE`) in register `%cr0` (0955-0957). Enabling protected mode does not change how the processor translates virtual to physical addresses or whether it is in 32-bit mode; it is only when one loads a new value into a segment register that the processor reads the GDT and changes its internal segmentation settings. Thus the processor continues to execute in 16-bit mode with the same segment translations as before. The switch to 32-bit mode happens when the code executes a far jump (`ljmp`) instruction (0961). The jump continues execution at the next line (0964) but in doing so sets `%cs` to `SEG_KCODE`, which causes the processor to load the descriptor entry from the `gdt` table. The entry describes a 32-bit code segment, so the processor switches into 32-bit mode. The boot sector code has nursed the processor through an evolution from 8088 through 80286 to 80386.

The boot sector's first action in 32-bit mode is to initialize the data segment registers with `SEG_KDATA` (0966-0969). The segments are set up so that the processor uses 32-bit virtual addresses directly as 32-bit physical addresses, without translation, so the software can now conveniently use all of the machine's memory. The only step left before executing C code is to set up a stack in an unused region of memory. The memory from `0xa0000` to `0x100000` is typically littered with device memory regions, and the xv6 kernel expects to be placed at `0x100000`. The boot sector itself is at `0x7c00` through `0x7d00`. Essentially any other section of memory would be a fine location for the stack. The boot sector chooses `0x7c00` (known in this file as `$start`) as the top of the stack; the stack will grow down from there, toward `0x0000`, away from the boot sector code.

Finally the boot sector calls the C function `bootmain` (0976). `Bootmain`'s job is to load and run the kernel. It only returns if something has gone wrong. In that case, the code sends a few output words on port 0x8a00 (0978-0984). On real hardware, there is no device connected to that port, so this code does nothing. If the boot sector is running inside the PC simulator Bochs, port 0x8a00 is connected to Bochs itself; the code sequence triggers a Bochs debugger breakpoint. Bochs or not, the code then executes an infinite loop (0985-0986). A real boot sector might attempt to print an error message first.

Code: C bootstrap

The C part of the boot sector, `bootmain.c` (1100), loads a kernel from an IDE disk into memory and then starts executing it. The kernel is an ELF format binary, defined in `elf.h`. An ELF binary is an ELF file header, `struct elfhdr` (0855), followed by a sequence of program section headers, `struct proghdr` (0874). Each `proghdr` describes a section of the kernel that must be loaded into memory. These headers typically take up the first hundred or so bytes of the binary. To get access to the headers, `bootmain` loads the first 4096 bytes of the file, a gross overestimation of the amount needed (1113). It places the in-memory copy at address `0x10000`, another out-of-the-way memory address.

`bootmain` casts freely between pointers and integers (1123, 1126, and so on). Programming languages distinguish the two to catch errors, but the underlying processor sees no difference. An operating system must work at the processor's level; occasionally it will need to treat a pointer as an integer or vice versa. C allows these conversions, in contrast to languages like Pascal and Java, precisely because one of the first uses of C was to write an operating system: Unix.

Back in the boot sector, what should be an ELF binary header has been loaded into memory at address `0x10000` (1113). The next step is to check that the first four bytes of the header, the so-called magic number, are the bytes `0x7F`, `'E'`, `'L'`, `'F'`, or `ELF_MAGIC` (0852). All ELF binary headers are required to begin with this magic number as identication. If the ELF header has the right magic number, the boot sector assumes that the binary is well-formed. There are many other sanity checks that a proper ELF loader would do, as we will see in Chapter 9, but the boot sector doesn't have the code space. Checking the magic number guards against simply forgetting to write a kernel to the disk, not against malicious binaries.

An ELF header points at a small number of program headers (`proghdrs`) describing the sections that make up the running kernel image. Each `proghdr` gives a virtual address (`va`), the location where the section's content lies on the disk relative to the start of the ELF header (`offset`), the number of bytes to load from the file (`filesz`), and the number of bytes to allocate in memory (`memsz`). If `memsz` is larger than `filesz`, the bytes not loaded from the file are to be zeroed. This is more efficient, both in space and I/O, than storing the zeroed bytes directly in the binary. As an example, the xv6 kernel has two loadable program sections, code and data:

```
# objdump -p kernel

kernel:     file format elf32-i386

Program Header:
LOAD off    0x00001000 vaddr 0x00100000 paddr 0x00100000 align 2**12
   filesz 0x000063ca memsz 0x000063ca flags r-x
LOAD off    0x000073e0 vaddr 0x001073e0 paddr 0x001073e0 align 2**12
   filesz 0x0000079e memsz 0x000067e4 flags rw-
 STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
   filesz 0x00000000 memsz 0x00000000 flags rwx
```

Notice that the second section, the data section, has a `memsz` larger than its `filesz`: the first `0x79e` bytes are loaded from the kernel binary and the remaining `0x6046` bytes are zeroed.

`Bootmain` uses the addresses in the `proghdr` to direct the loading of the kernel. It reads each section's content starting from the disk location `offset` bytes after the start of the ELF header, and writes to memory starting at address `va`. `Bootmain` calls `readseg` to load data from disk (1137) and calls `stosb` to zero the remainder of the segment (1139). `Stosb` (0442) uses the x86 instruction `rep stosb` to initialize every byte of a block of memory.

`Readseg` (1179) reads at least `count` bytes from the disk `offset` into memory at `va`. The x86 IDE disk interface operates in terms of 512-byte chunks called sectors, so `readseg` may read not only the desired section of memory but also some bytes before and after, depending on alignment. For the second program segment in the example above, the boot sector will call `readseg((uchar*)0x1073e0, 0x73e0, 0x79e)`. Due to sector granularity, this call is equivalent to `readseg((uchar*)0x107200, 0x7200, 0xa00)`: it reads `0x1e0` bytes before the desired memory region and `0x82` bytes afterward. In practice, this sloppy behavior turns out not to be a problem (see exercise XXX). `Readseg` begins by computing the ending virtual address, the first memory address above `va` that doesn't need to be loaded from disk (1183), and rounding `va` down to a sector-aligned disk offset . Then it converts the offset from a byte offset to a sector offset; it adds 1 because the kernel starts at disk sector 1 (disk sector 0 is the boot sector). Finally, it calls `readsect` to read each sector into memory.

`Readsect` (1160) reads a single disk sector. It is our first example of a device driver, albeit a tiny one. `Readsect` begins by calling `waitdisk` to wait until the disk signals that it is ready to accept a command. The disk does so by setting the top two bits of its status byte (connected to input port `0x1f7`) to `01`. `Waitdisk` (1151) reads the status byte until the bits are set that way. Chapter 6 will examine more efficient ways to wait for hardware status changes, but busy waiting like this (also called polling) is fine for the boot sector.

Once the disk is ready, `readsect` issues a read command. It first writes command arguments—the sector count and the sector number (offset)—to the disk registers on output ports `0x1f2-0x1f6` (1164-1168). The bits `0xe0` in the write to port `0x1f6` signal to the disk that `0x1f3-0x1f6` contain a sector number (a so-called linear block address), in contrast to a more complicated cylinder/head/sector address used in early PC disks. After writing the arguments, `readsect` writes to the command register to trigger the read (1154). The command `0x20` is "read sectors." Now the disk will read the data stored in the specified sectors and make it available in 32-bit pieces on input port `0x1f0`. `Waitdisk` (1151) waits until the disk signals that the data is ready, and then the call to `insl` reads the 128 (`SECTSIZE/4`) 32-bit pieces into memory starting at `dst` (1173).

`Inb`, `outb`, and `insl` are not ordinary C functions. They are inlined functions whose bodies are assembly language fragments (0403, 0421, 0412). When gcc sees the call to `inb` (1154), the inlined assembly causes it to emit a single `inb` instruction. This style allows the use of low-level instructions like `inb` and `outb` while still writing the control logic in C instead of assembly.

The implementation of `insl` (0412) is worth looking at more closely. `Rep insl` is actually a tight loop masquerading as a single instruction. The `rep` prefix executes the following instruction `%ecx` times, decrementing `%ecx` after each iteration. The `insl` instruction reads a 32-bit value from port `%dx` into memory at address `%edi` and then increments `%edi` by 4. Thus `rep insl` copies 4×`%ecx` bytes, in 32-bit chunks, from port `%dx` into memory starting at address `%edi`. The register annotations tell GCC to prepare for the assembly sequence by storing `dst` in `%edi`, `cnt` in `%ecx`, and `port` in `%dx`. Thus the `insl` function copies 4×`cnt` bytes from the 32-bit port `port` into memory starting at `dst`. The `cld` instruction clears the processor's direction flag, so that the `insl` instruction increments `%edi`; when the flag is set, `insl` decrements `%edi` instead. The x86 calling convention does not define the state of the direction flag on entry to a function, so each use of an instruction like `insl` must initialize it to the desired value.

The boot loader is almost done. `Bootmain` loops calling `readseg`, which loops calling `readsect` (1135-1140). At the end of the loop, `bootmain` has loaded the kernel into memory. Now it is time to run the kernel. The ELF header specifies the kernel entry point, the `%eip` where the kernel expects to be started (just as the boot loader expected to be started at `0x7c00`). `Bootmain` casts the entry point integer to a function pointer and calls that function, essentially jumping to the kernel's entry point (1144-1145). The kernel should not return, but if it does, `bootmain` will return, and then `bootasm.S` will attempt a Bochs breakpoint and then loop forever.

Where is the kernel in memory? `Bootmain` does not directly decide; it just follows the directions in the ELF headers. The "linker" creates the ELF headers, and the xv6 Makefile that calls the linker tells it that the kernel should start at `0x100000`.

Assuming all has gone well, the kernel entry pointer will be the kernel's `main` function (see `main.c`). The next chapter continues there.

Real world

The boot sector described in this chapter compiles to around 470 bytes of machine code, depending on the optimizations used when compiling the C code. In order to fit in that small amount of space, the xv6 boot sector makes a major simplifying assumption, that the kernel has been written to the boot disk contiguously starting at sector 1. More commonly, kernels are stored in ordinary file systems, where they may not be contiguous, or are loaded over a network. These complications require the boot loader to be able to drive a variety of disk and network controllers and understand various file systems and network protocols. In other words, the boot loader itself must be a small operating system. Since such complicated boot loaders certainly won't fit in 512 bytes, most PC operating systems use a two-step boot process. First, a simple boot sector like the one in this chapter loads a full-featured boot-loader from a known disk location, often relying on the less space-constrained BIOS for disk access rather than trying to drive the disk itself. Then the full loader, relieved of the 512-byte limit, can implement the complexity needed to locate, load, and execute the desired kernel.

TODO: Also, x86 does not imply BIOS: Macs use EFI. I wonder if the Mac has an A20 line.

Exercises

1. Look at the kernel load addresses; why doesn't the sloppy readsect cause problems?
2. something about BIOS lasting longer + security problems
3. Suppose you wanted bootmain() to load the kernel at 0x200000 instead of 0x100000, and you did so by modifying bootmain() to add 0x100000 to the va of each ELF section. Something would go wrong. What?