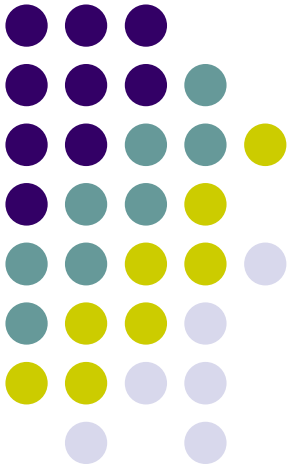
System Programming

11. Booting Process from XV6



2019. Fall

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

datalab

data science laboratory

These slides are based on the book titled as Writing a Simple Operating System – from Scratch By Nick Blundell

References

- Writing a Simple Operating System – from Scratch By Nick Blundell
  - You can find this article by googling

Starting an OS

- Computer turns on, BIOS loads
  - high-level: firmware → bootloader → OS kernel

BIOS

- the pre-OS environment of your computer offers little in the way of rich services:
  - even a simple le system would be a luxury

- BIOS (Basic Input/Output Software)
  - a collection of software routines that are initially loaded from a chip into memory and initialised when the computer is switched on
  - provides auto-detection and basic control of your computer's essential devices, such as the screen, keyboard, and hard disks.

Boot Process (1/2)

- After BIOS completes some low-level tests of the hardware, BIOS must read specific sectors of data
  - (usually 512 bytes in size) from specific physical locations of the disk devices, such as Cylinder 2, Head 3, Sector 5
  - **Boot sector**
    - the easiest place for BIOS to find our OS is in the rst sector of one of the disks (i.e. Cylinder 0, Head 0, Sector 0)

Boot Process (2/2)

- Some of our disk may not contain OS
  - BIOS can determine whether the boot sector of a particular disk is boot code for execution or simply data
  - BIOS checks the last two bytes of an intended boot sector must be set to the magic number 0xaa55.
    - BIOS loops through each storage device

    → reads the boot sector into memory

    → and instructs the CPU to begin executing the first boot sector it finds that ends with the magic number.

A machine code boot sector

- A machine code boot sector
  - the smallest program your computer could run

```
  e9 fd ff 00 00 00 00 00 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa
```

'*' indicates zeros omitted for brevity

Little-endian

  - The initial three bytes 0xe9, 0xfd and 0xff
    - machine code instructions to perform endless jump.
  - The last two bytes 0x55 and 0xaa
    - make up the magic number
  - The file
    - Is padded with zeros, basically to position the magic BIOS number at the end of the 512 byte disk sector.

Testing a boot sector

- Three ways for Testing a Boot sector
  - 1. Use a real machine
    - write this boot block to the first sector of a non-essential storage device then reboot the computer.
  - 2. Use virtual machine software (VMWare or VirtualBox) : (safer and better)
    - set the boot block code as a disk image of a virtual machine
    - then start-up the virtual machine
  - 3. CPU emulation (more convenient option)

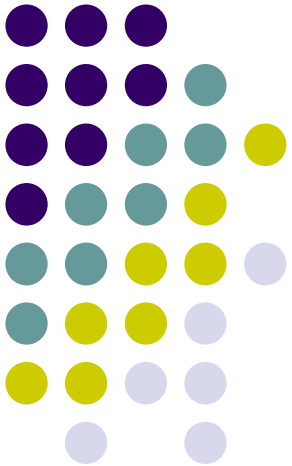
CPU emulation (1/2)

- use a CPU emulator such as Bochs or Qemu
  - Without continuously having to reboot a machine or risk scrubbing your important data off a disk
- involves a program that behaves like a specific CPU architecture
- uses variables to represent CPU registers and high-level control structures to simulate lower level jumps and so on
- is much slower but often better suited for development and debugging such systems.

CPU emulation (2/2)

- QEMU
  - a modern and relatively fast emulator
    - https://www.qemu.org/
  - QEMU can act as a remote debugging target for the GNU debugger (GDB)

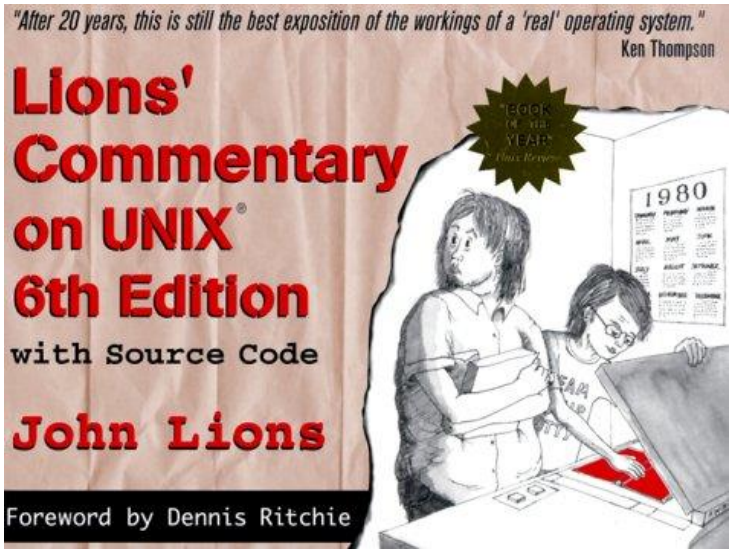
XV6



Unix v6 (1/2)

- Early Unix OS for DEC PDP11
- By Thompson & Ritchie, 1975
- From Bell labs; the 1st version to be widely used outside Bell
- Ancestor of all Unix flavors (Linux, *BSD, Solaris,…)
- (But much smaller)
- Written in C

Unix v6 (2/2)



- 1976 Commentary by Lions: a classic….
- Despite its age still considered an excellent commentary on simple but high quality code
- For many years, was the only Unix kernel documentation publicly available
- Reprinted in 1996; still sold

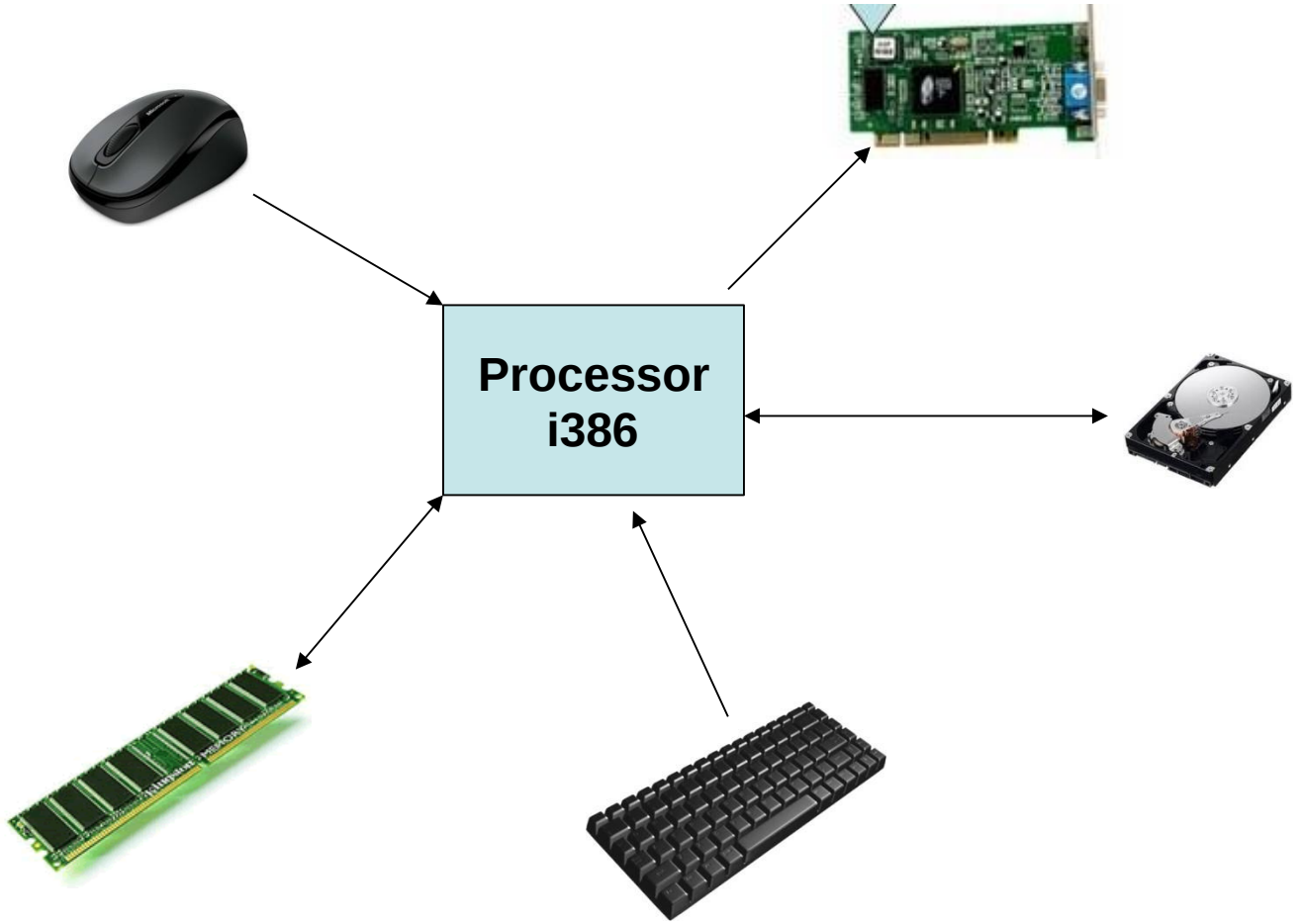
Xv6 (1/2)

- xv6 is an MIT reimplementation of Unix v6
  - https://pdos.csail.mit.edu/6.828/2016/xv6.html
- Runs on x86 (well if you insist; we'll use QEMU)
- Even smaller than v6
- Preserves basic structure (processes, files, pipes, etc.)
- Runs on multicores
- Just this year (2011) got paging

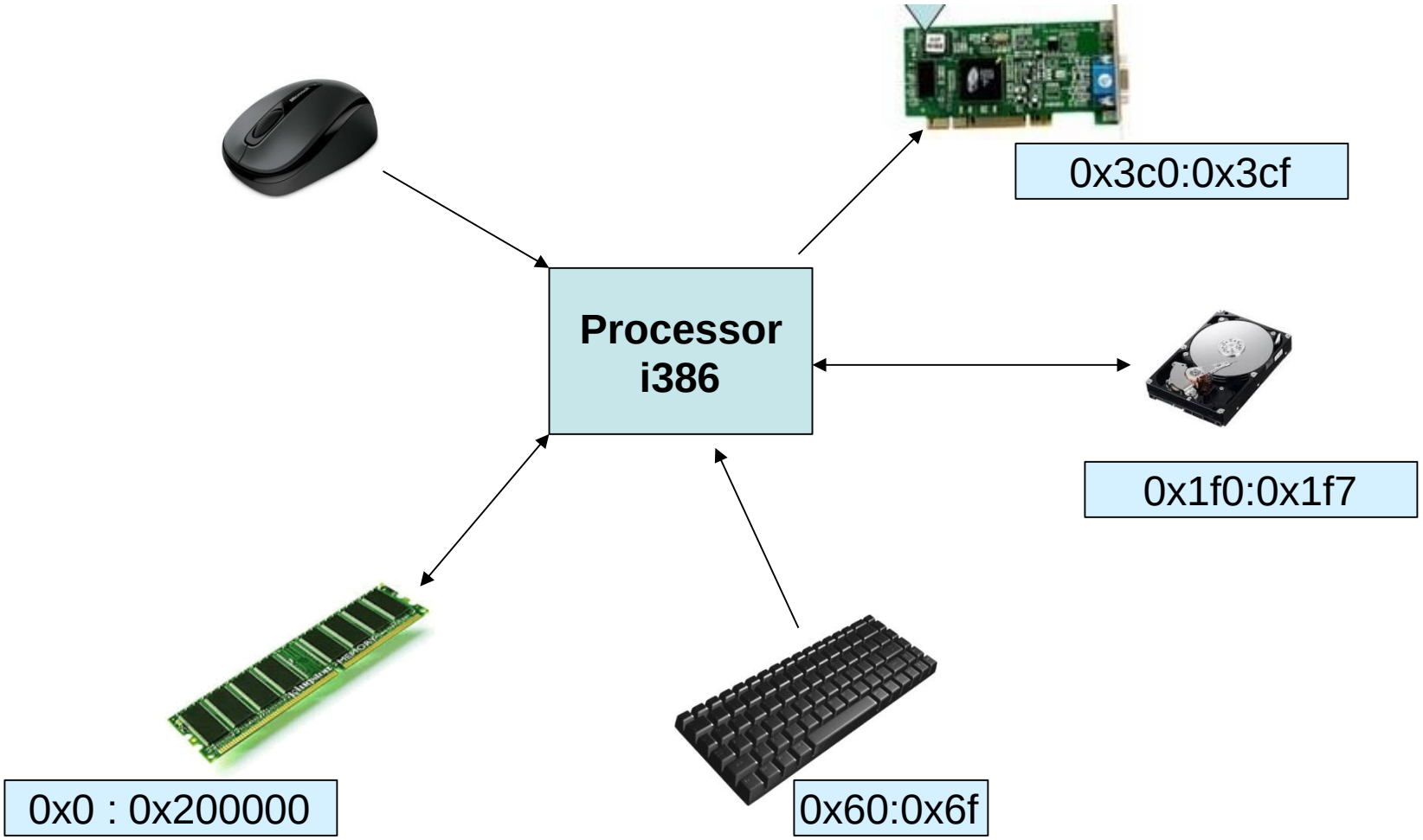
Xv6 (2/2)

- xv6 is big enough to illustrate the basic OS design & implementation
- Yet xv6 is far smaller => much easier to understand
- Structure similar to many modern OSes
  - Once you've explored xv6 you'll find your way inside kernels such as Linux

CPUs



Everything has an address

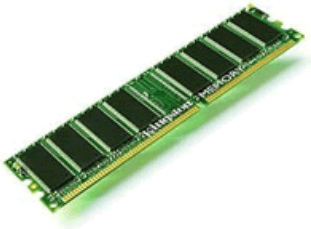


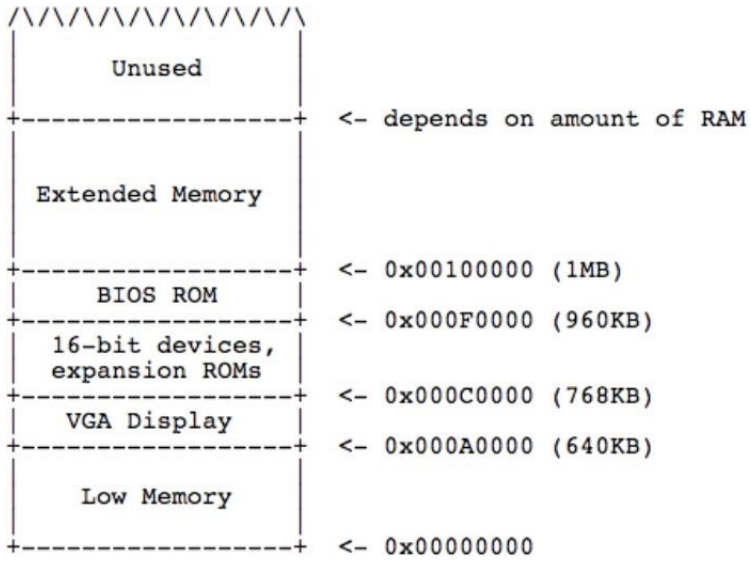
Address Types

- Memory Addresses
- IO Addresses
- Memory Mapped IO Addresses

Address Types : (Memory Addresses)

- Range : 0 to (RAM size or 2^{32}-1)
- Where main memory is mapped
  - Used to store data for code, heap, stack, OS, etc.
- Accessed by load/store instructions





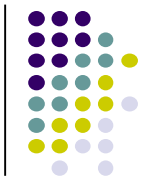
Memory Address Map

Low and Extended Memory (Legacy Issues)

- Why study it?
  - Backward compatibility
- 8088 has 20 address lines; can address 2^{20} bytes (1MB)

- Memory Ranges
  - 0 to 640KB used by IBM PC MSDOS
    - Other DOS versions have a different memory limit
  - 640 KB to 1MB used by video buffers, expansion ROMS, BIOS ROMs
  - 1 MB onwards called extended memory
- Modern processors have more usable memory
  - OSes like Linux and x86 simply ignore the first 1MB and load kernel in extended memory

Address Types : (IO Ports)



- Range : 0 to 2^{16}-1
- Used to access devices
- Uses a different bus compared to RAM memory access
  - Completely isolated from memory
- Accessed by in/out instructions

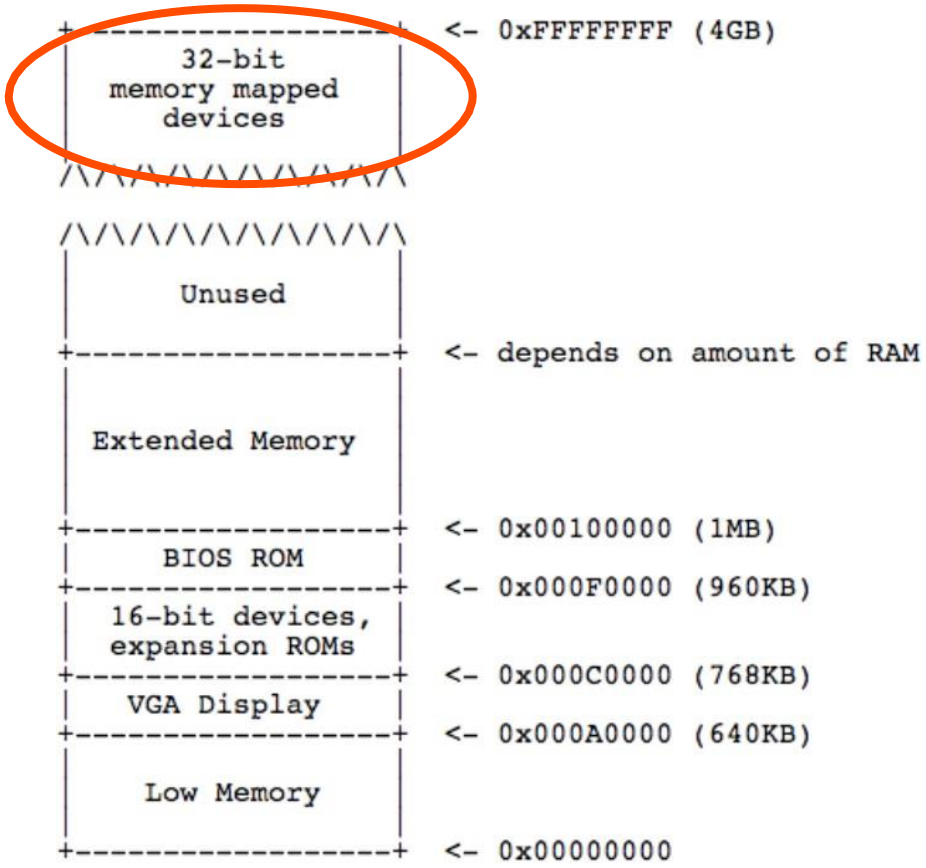
```
inb $0x64, %al
outb %al, $0x64
```

I/O address range	Device
00 – 1F	First DMA controller 8237 A-5
20 – 3F	First Programmable Interrupt Controller, 8259A, Master
40 – 5F	Programmable Interval Timer (System Timer), 8254
60 – 6F	Keyboard, 8042
70 – 7F	Real Time Clock, NMI mask
80 – 9F	DMA Page Register, 74LS612
87	DMA Channel 0
83	DMA Channel 1
81	DMA Channel 2
82	DMA Channel 3
8B	DMA Channel 5
89	DMA Channel 6
8A	DMA Channel 7
8F	Refresh
A0 – BF	Second Programmable Interrupt Controller, 8259A, Slave
C0 – DF	Second DMA controller 8237 A-5
F0	Clear 80287 Busy
F1	Reset 80287
F8 – FF	Math coprocessor, 80287
F0 – F5	PCjr Disk Controller
F8 – FF	Reserved for future microprocessor extensions
100 – 10F	POS Programmable Option Select (PS/2)
110 – 1EF	System I/O channel
140 – 15F	Secondary SCSI host adapter
170 – 177	Secondary Parallel ATA Disk Controller
1F0 – 1F7	Primary Parallel ATA Hard Disk Controller
200 – 20F	Game port
210 – 217	Expansion Unit
220 – 233	Sound Blaster and most other sound cards
278 – 27F	Parallel port 3

http://bochs.sourceforge.net/techspec/PORTS.LST

Memory Mapped I/O

- Why?
  - More space
  - Devices and RAM share the same address space
  - Instructions used to access RAM can also be used to access devices.
    - Eg load/store



Memory Map

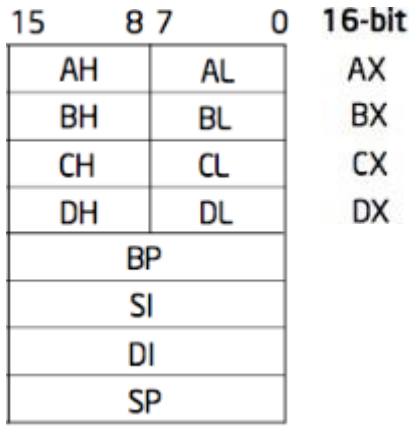
Whodecides the address ranges?

- Standards / Legacy
  - Such as the IBM PC standard
  - Fixed for all PCs.
  - Ensures BIOS and OS to be portable across platforms
- Plug and Play devices
  - Address range set by BIOS or OS
  - A device address range may vary every time the system is restarted

The x86 Evolution(8088)

- 8088
  - 16 bit microprocessor
  - 20 bit external address bus
    - Can address 1MB of memory
  - Registers are 16 bit
    - General Purpose Registers
      - AX, BX, CD, DX,
    - Pointer Registers
      - BP, SI, DI, SP
    - Instruction Pointer : IP
    - Segment Registers
      - CS, SS, DS, ES
  - Accessing memory
    - (segment_base << 4) + offset eg: (CS << 4) + IP

General Purpose Registers



GPRs can be accessed as
8 bit or 16 bit registers
Eg.

```
mov $0x1, %ah ; 8 bit move
mov $0x1, %ax ; 16 bit move
```

The x86 Evolution(80386)

- 80386 (1995)
  - 32 bit microprocessor
  - 32 bit external address bus
    - Can address 4GB of memory
  - Registers are 32 bit
    - General Purpose Registers
      - EAX, EBX, ECD, EDX,
    - Pointer Registers
      - EBP, ESI, EDI, ESP
    - Instruction Pointer : IP
    - Segment Registers
      - CS, SS, DS, ES
  - Lot more features
    - Protected operating mode
    - Virtual addresses

General Purpose Registers

General-Purpose Registers

31 16	15 8	7 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

GPRs can be accessed as
8, 16, 32 bit registers
e.g.

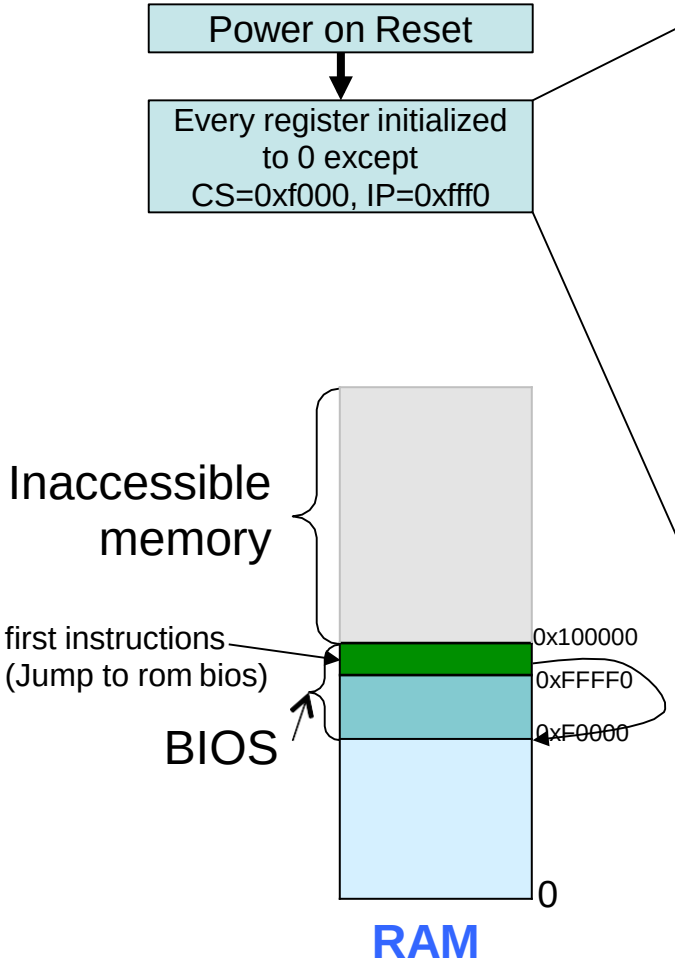
```
mov $0x1, %ah ; 8 bit move
mov $0x1, %ax ; 16 bit move
mov $0x1, %eax ; 32 bit move
```

The x86 Evolution(k8)

- AMD k8 (2003)
  - RAX instead of EAX
  - X86-64, x64, amd64, intel64: all same thing
- Backward compatibility
  - All systems backward compatible with 8088

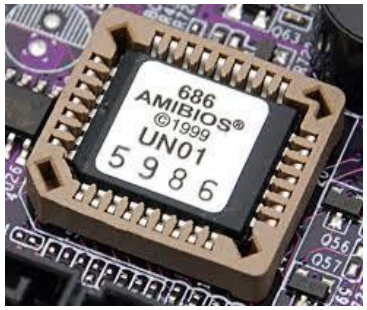
Powering Up

Poweringup : Reset

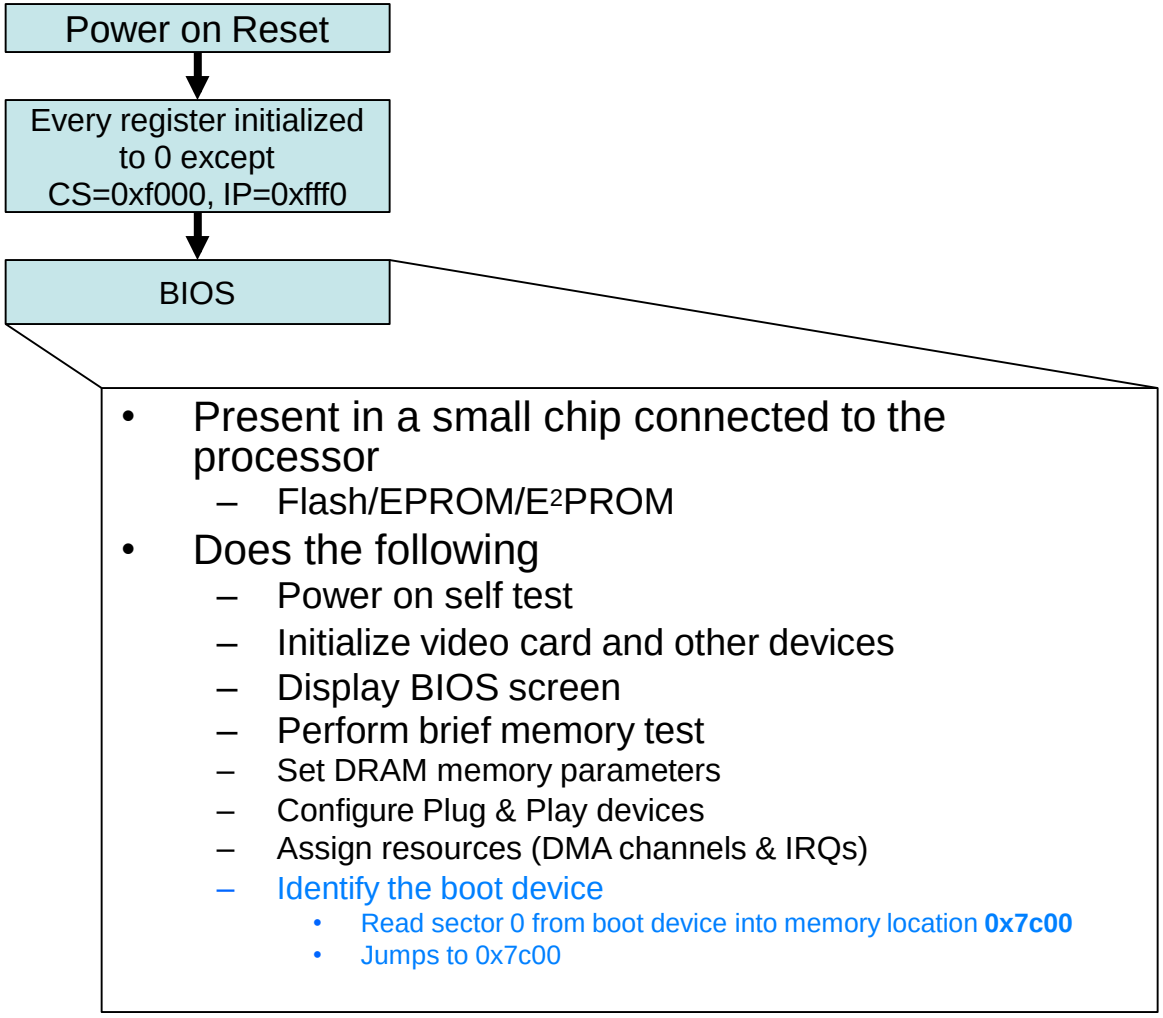


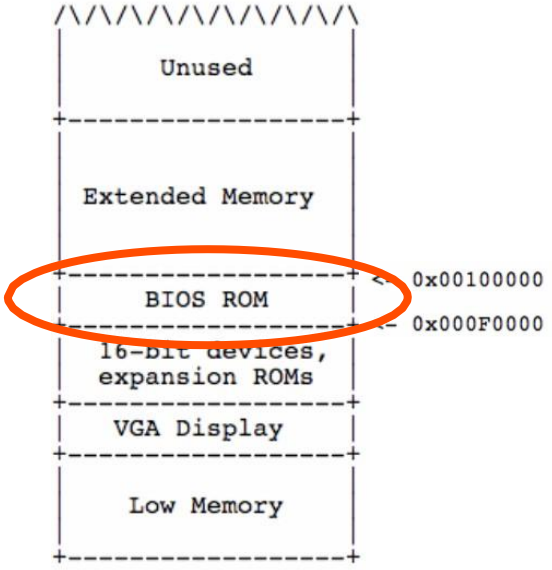
Physical address = (CS << 4) + IP
= 0xffff0

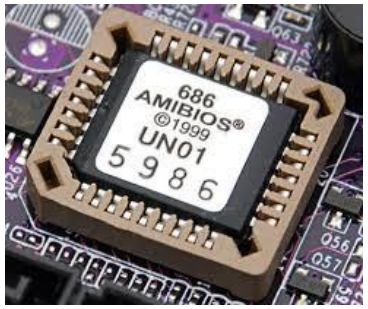
- first instruction fetched from location 0xffff0.
- Processor in real mode (backward compatible to 8088)
  - Limited to 1MB addresses
  - No protection; no privilege levels
  - Direct access to all memory
  - No multi-tasking
- First instruction is right on top of accessible memory
  - Should jump to another location



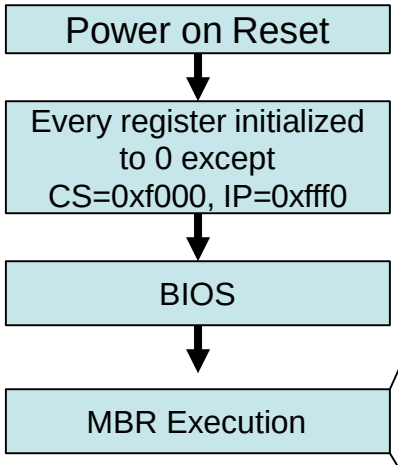
Poweringup : BIOS





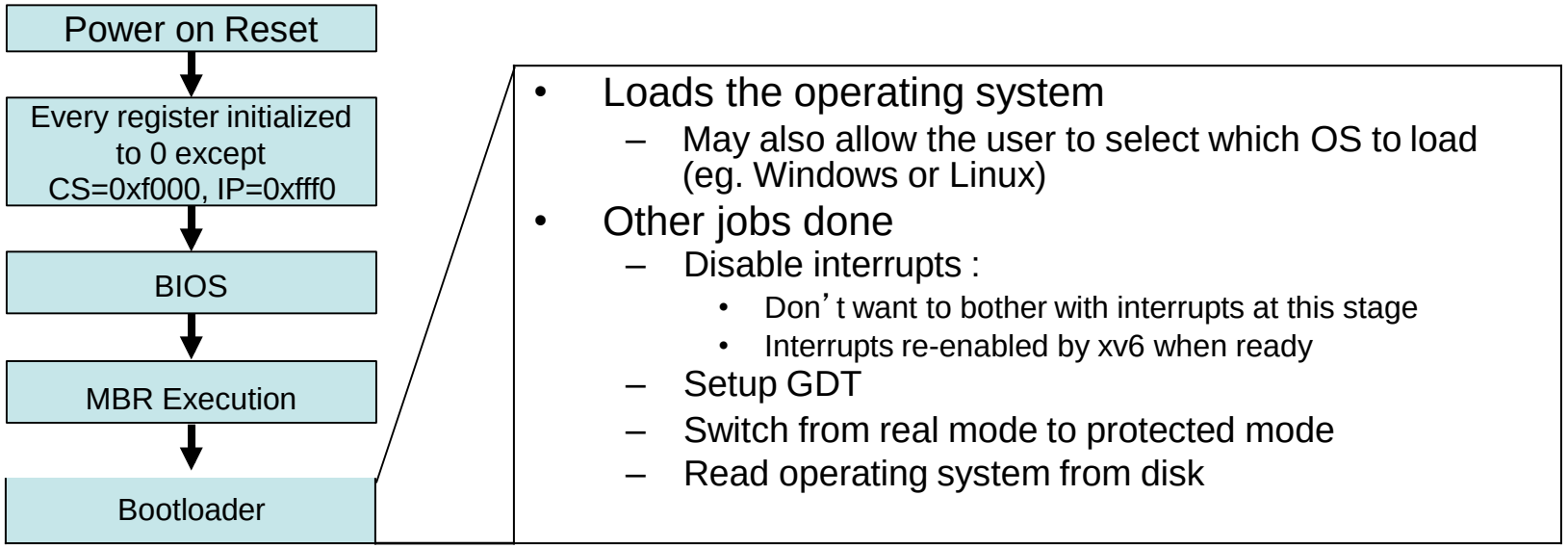


Poweringup : MBR



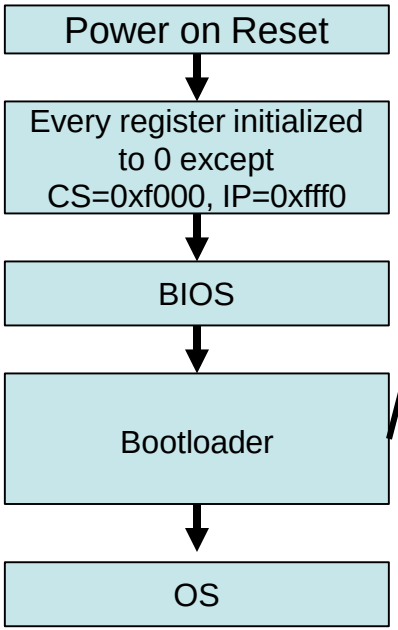
- Sector 0 in the disk called Master Boot Record (MBR)
- Contains code that boots the OS or another boot loader
- Copied from disk to RAM (@0x7c00) by BIOS and then begins to execute
- Size 512 bytes
  - 446 bytes bootable code
  - 64 bytes disk partition information (16 bytes per partition)
  - 2 bytes signature
- Typically, MBR code looks through partition table and loads the bootloader (such as Linux or Windows)
- or, it may directly load the OS

Powering Up : bootloader



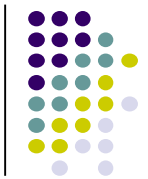
The bootloader may be present in the MBR (sector 0) itself

Powering Up : xv6



- Bootloader
  - Present in sector 0 of disk.
  - 512 bytes
  - 2 parts:
    - bootasm.S (*8900*)
      - Enters in 16 bit real mode, leaves in 32 bit protected mode
      - Disables interrupts
        - We don't want to use BIOS ISRs
      - Enable A20 line
      - Load GDT (only segmentation, no paging)
      - Set stack to 0x7c00
      - Invoke bootmain
      - Never returns
    - bootmain.c (*9017*)
      - Loads the xv6 kernel from sector 1 to RAM starting at 0x100000 (1MB)
      - Invoke the xv6 kernel entry
        - _start present in entry.S (sheet 10)
        - This entry point is known from the ELF header

xv6 : bootasm.S



Gets loaded into 0x7c00 by the BIOS?

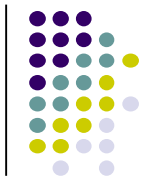
```
8909 .code16                    # Assemble for 16-bit mode
8910 .globl start
8911 start:
8912   cli                      # BIOS enabled interrupts; disable
8913
8914   # Zero data segment registers DS, ES, and SS.
8915   xorw    %ax,%ax           # Set %ax to zero
8916   movw    %ax,%ds           # -> Data Segment
8917   movw    %ax,%es           # -> Extra Segment
8918   movw    %ax,%ss           # -> Stack Segment
8919
8920   # Physical address line A20 is tied to zero so that the first PCs
8921   # with 2 MB would run software that assumed 1 MB.  Undo that.
8922 seta20.1:
8923   inb     $0x64,%al               # Wait for not busy
8924   testb   $0x2,%al
8925   jnz     seta20.1
8926
8927   movb    $0xd1,%al               # 0xd1 -> port 0x64
8928   outb    %al,$0x64
8929
8930 seta20.2:
8931   inb     $0x64,%al               # Wait for not busy
8932   testb   $0x2,%al
8933   jnz     seta20.2
8934
8935   movb    $0xdf,%al               # 0xdf -> port 0x60
8936   outb    %al,$0x60
```

Note 16 bit code (compatible with 8088)

Loading :
Handled by the BIOS

Linking :
What linker options need to be set?

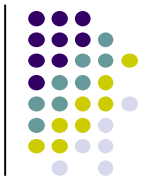
xv6 : bootasm.S



```
8909 .code16                       # Assemble for 16-bit mode
8910 .globl start
8911 start:
8912   cli                         # BIOS enabled interrupts; disable
8913
8914   # Zero data segment registers DS, ES, and SS.
8915   xorw    %ax,%ax             # Set %ax to zero
8916   movw    %ax,%ds             # -> Data Segment
8917   movw    %ax,%es             # -> Extra Segment
8918   movw    %ax,%ss             # -> Stack Segment
8919
8920   # Physical address line A20 is tied to zero so that the first PCs
8921   # with 2 MB would run software that assumed 1 MB.  Undo that.
8922 seta20.1:
8923   inb     $0x64,%al               # Wait for not busy
8924   testb   $0x2,%al
8925   jnz     seta20.1
8926
8927   movb    $0xd1,%al               # 0xd1 -> port 0x64
8928   outb    %al,$0x64
8929
8930 seta20.2:
8931   inb     $0x64,%al               # Wait for not busy
8932   testb   $0x2,%al
8933   jnz     seta20.2
8934
8935   movb    $0xdf,%al               # 0xdf -> port 0x60
8936   outb    %al,$0x60
```

Disable interrupts. Ini tialize registers to 0

xv6 : bootasm.S

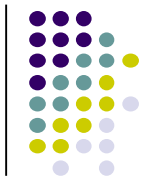


```
8909 .code16                       # Assemble for 16-bit mode
8910 .globl start
8911 start:
8912   cli                         # BIOS enabled interrupts; disable
8913
8914   # Zero data segment registers DS, ES, and SS.
8915   xorw    %ax,%ax             # Set %ax to zero
8916   movw    %ax,%ds             # -> Data Segment
8917   movw    %ax,%es             # -> Extra Segment
8918   movw    %ax,%ss             # -> Stack Segment
8919
8920   # Physical address line A20 is tied to zero so that the first PCs
8921   # with 2 MB would run software that assumed 1 MB.  Undo that.
8922 seta20.1:
8923   inb     $0x64,%al               # Wait for not busy
8924   testb   $0x2,%al
8925   jnz     seta20.1
8926
8927   movb    $0xd1,%al               # 0xd1 -> port 0x64
8928   outb    %al,$0x64
8929
8930 seta20.2:
8931   inb     $0x64,%al               # Wait for not busy
8932   testb   $0x2,%al
8933   jnz     seta20.2
8934
8935   movb    $0xdf,%al               # 0xdf -> port 0x60
8936   outb    %al,$0x60
```

Enable A20 line.

Why do we have it?

xv6 : bootasm.S



Switch from real to protected mode

```
8938    # Switch from real to protected mode.  Use a bootstrap GDT that makes
8939    # virtual addresses map directly to physical addresses so that the
8940    # effective memory map doesn't change during the transition.
8941    lgdt    gdtdesc
8942    movl    %cr0, %eax
8943    orl     $CR0_PE, %eax
8944    movl    %eax, %cr0
```

```
8980 # Bootstrap GDT
8981 .p2align 2                                # force 4 byte alignment
8982 gdt:
8983   SEG_NULLASM                             # null seg
8984   SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
8985   SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg
8986
8987 gdtdesc:
8988   .word   (gdtdesc - gdt - 1)             # sizeof(gdt) - 1
8989   .long   gdt                             # address gdt
8990
```

GDT related information

xv6 : bootasm.S

Switch from real to protected mode

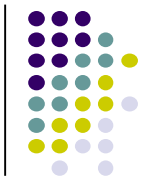
16 bit code

```
8938    # Switch from real to protected mode.  Use a bootstrap GDT that makes
8939    # virtual addresses map directly to physical addresses so that the
8940    # effective memory map doesn't change during the transition.
8941    lgdt    gdtdesc
8942    movl    %cr0, %eax
8943    orl     $CR0_PE, %eax
8944    movl    %eax, %cr0
```

```
8950    # Complete transition to 32-bit protected mode by using long jmp
8951    # to reload %cs and %eip.  The segment descriptors are set up with no
8952    # translation, so that the mapping is still the identity mapping.
8953    ljmp    $(SEG_KCODE<<3), $start32
8954
8955 .code32  # Tell assembler to generate 32-bit code now.
8956 start32:
8957    # Set up the protected-mode data segment registers
```

32 bit code

xv6 : bootasm.S

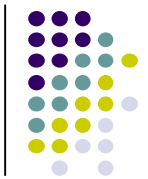


```
8909 .code16                          # Assemble for 16-bit mode
8910 .globl start
8911 start:
8912   cli                            # BIOS enabled interrupts; disable
8913
8914   # Zero data segment registers DS, ES, and SS.
8915   xorw    %ax,%ax                # Set %ax to zero
8916   movw    %ax,%ds                # -> Data Segment
8917   movw    %ax,%es                # -> Extra Segment
8918   movw    %ax,%ss                # -> Stack Segment
8919
8920   # Physical address line A20 is tied to zero so that the first PCs
8921   # with 2 MB would run software that assumed 1 MB.  Undo that.
8922 seta20.1:
8923   inb     $0x64,%al              # Wait for not busy
8924   testb   $0x2,%al
8925   jnz     seta20.1
8926
8927   movb    $0xd1,%al              # 0xd1 -> port 0x64
8928   outb    %al,$0x64
8929
8930 seta20.2:
8931   inb     $0x64,%al              # Wait for not busy
8932   testb   $0x2,%al
8933   jnz     seta20.2
8934
8935   movb    $0xdf,%al              # 0xdf -> port 0x60
8936   outb    %al,$0x60
```

Enable A20 line.

Why do we have it?

xv6 : bootasm.S



```
8956 start32:
8957   # Set up the protected-mode data segment registers
8958   movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
8959   movw    %ax, %ds                # -> DS: Data Segment
8960   movw    %ax, %es                # -> ES: Extra Segment
8961   movw    %ax, %ss                # -> SS: Stack Segment
8962   movw    $0, %ax                 # Zero segments not ready for use
8963   movw    %ax, %fs                # -> FS
8964   movw    %ax, %gs                # -> GS
8965
8966   # Set up the stack pointer and call into C.
8967   movl    $start, %esp
8968   call    bootmain
```

Set up stack and call a C function.

Note the stack pointer points to 0x7c00.
This means the stack grows downwards
from 0x7c00. why?

bootmain

```
9016 void
9017 bootmain(void)
9018 {
9019   struct elfhdr *elf;
9020   struct proghdr *ph, *eph;
9021   void (*entry)(void);
9022   uchar* pa;
9023
9024   elf = (struct elfhdr*)0x10000;  // scratch space
9025
9026   // Read 1st page off disk
9027   readseg((uchar*)elf, 4096, 0);
9028
9029   // Is this an ELF executable?
9030   if(elf->magic != ELF_MAGIC)
9031     return;  // let bootasm.S handle error
9032
9033   // Load each program segment (ignores ph flags).
9034   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
9035   eph = ph + elf->phnum;
9036   for(; ph < eph; ph++){
9037     pa = (uchar*)ph->paddr;
9038     readseg(pa, ph->filesz, ph->off);
9039     if(ph->memsz > ph->filesz)
9040       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
9041   }
9042
9043   // Call the entry point from the ELF header.
9044   // Does not return!
9045   entry = (void(*)(void))(elf->entry);
9046   entry();
9047 }
```

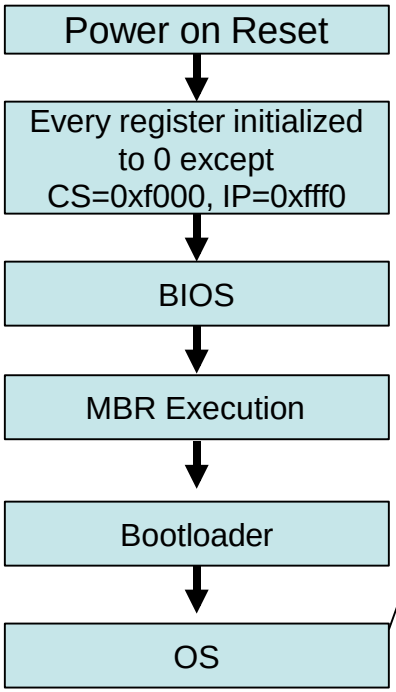
Load in 1MB region

The xv6 kernel is stored as an ELF image.

Read kernel from the disk (sector 1) to RAM.

Read the entry function in the kernel and Invoke it. This starts the OS

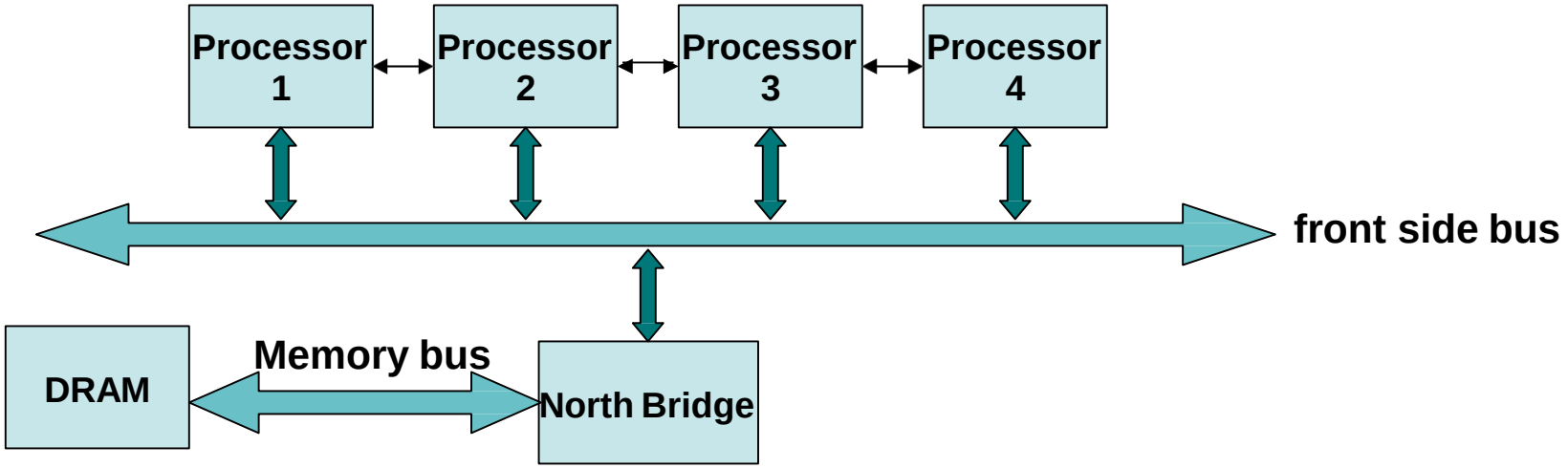
Powering Up : OS



- Set up virtual memory
- Initialize interrupt vectors
- Initilize
  - timers,
  - monitors,
  - hard disks,
  - consoles,
  - filesystems,
- Initialized other processors (if any)
- Startup user process

Multi-processor Bootup

- Multiprocessor Organization



- Memory Symmetry
  - All processors in the system share the same memory space
  - Advantage : Common operating system code
- I/O Symmetry
  - All processors share the same I/O subsystem
  - Every processor can receive interrupt from any I/O device

Multiprocessor Booting

- One processor designated as ‘Boot Processor’ (BSP)
  - Designation done either by Hardware or BIOS
  - All other processors are designated AP (Application Processors)
- BIOS boots the BSP
- BSP learns system configuration
- BSP triggers boot of other AP
  - Done by sending an Startup IPI (inter processor interrupt) signal to the AP

xv6 Multiprocessor Boot

- mpinit (7001) invoked from main (1221)
  - Searches for an MP table in memory
    - (generally put there by the BIOS)
    - Contains information about processors in system along with other details such as IO-APICs, Processor buses, etc.
    - Extracts system information from MP table
  - Fills in the cpu id (7024)
    - CPU is a structure which contains CPU specific data (2304)

Booting APs

- startothers (1274) invoked from main(1237)
  - copy ‘entryother’ to location 0x7000
  - For each CPU found
    - Allocate a stack (1295)
    - Set C entry point to mpenter (1252)
    - Send a Startup IPI (1299)
      - Pass the entryother.S location to the new processor (40:67 ß 0x7000 >> 4)
      - Send inter processor interrupt to the AP processor using its apicid
    - Wait until CPU has started

Q&A