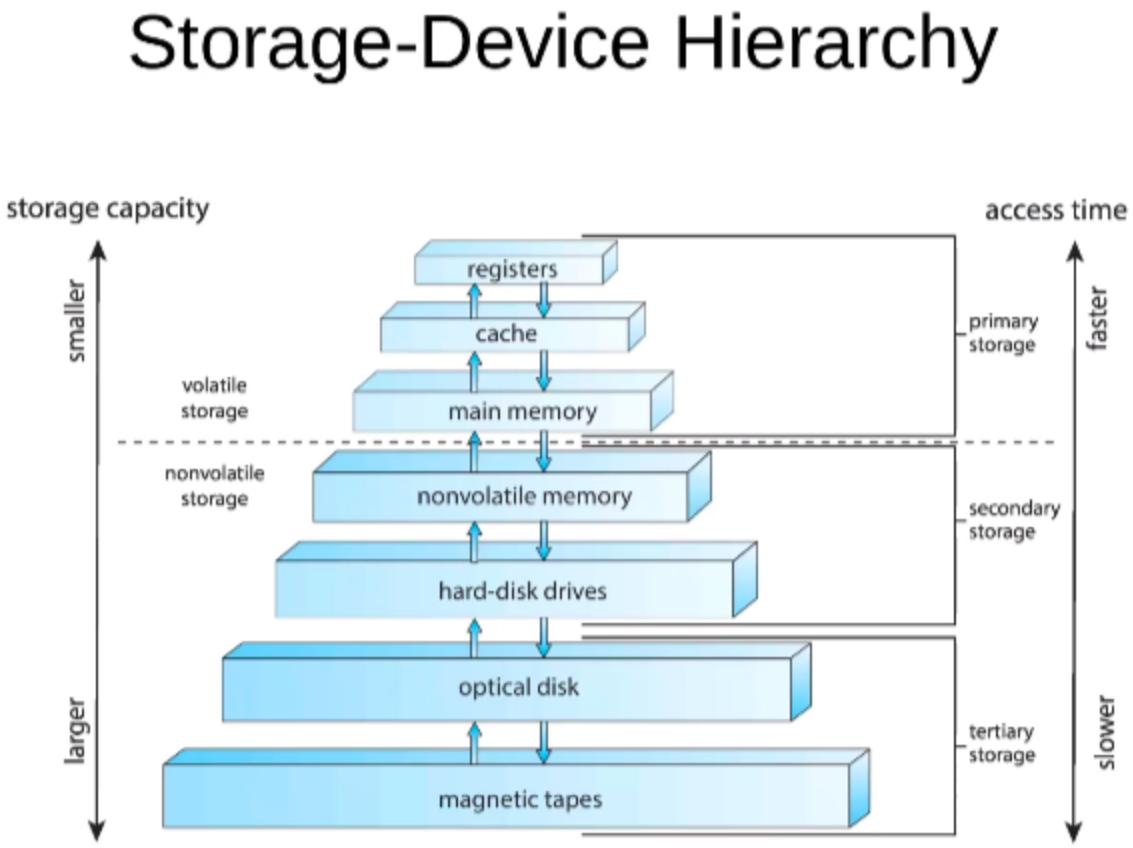
slides

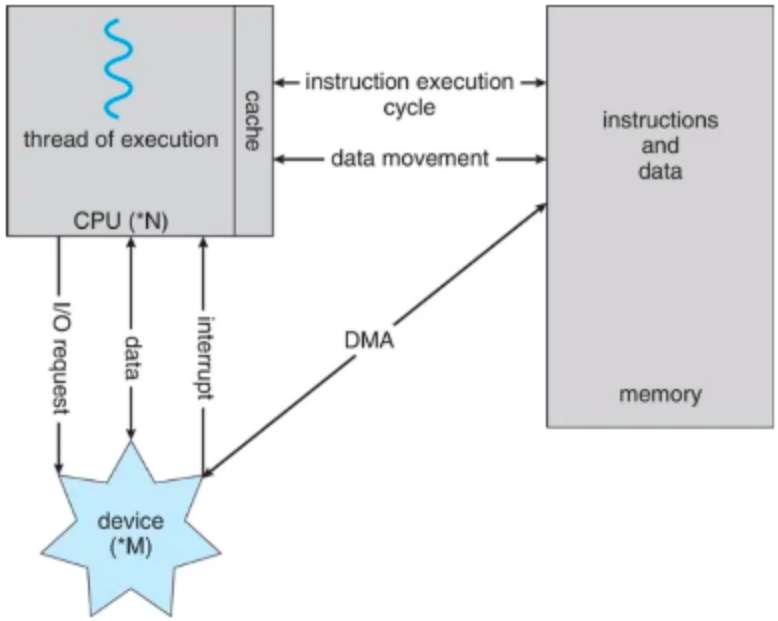
cpu works independently of the OS, it does not ask OS which instructions to execute

processor is not a part of OS



system bus is the main wire which connects all the components of modern computer

How a Modern Computer Works



A von Neumann architecture

left box is CPU,it has various threads of execution, and cache is put on cpu chip itself, the cache is entirely handled in the hardware.

RAM is a chip connected on motherboard, cpu and ram are connected by system bus, there is continuous movement of data and instruction between cpu and ram.

hardware devices are connected to cpu using intterupt,data, i/o request pin

there is a direct connection between hardware and RAM using DMA

Important Facts

A most important slide
What does the processor do?

From the moment it's turned on until it's turned off, the processor simply does this

1)Fetch the instruction from RAM (Memory).

Location is given by Program Counter (PC) register

2)Decode the instruction and execute it

While doing this may fetch some data from the RAM

3)While executing the instruction change/update the Program Counter

4)Go to 1

hence CPU is a dumb device

BIOS is hardwired into ROM on motherboard by manufacturer.

Bios loads boot loader,code loaded by boot loader is kernel initially when computer loads,

kernel takes the control of the hardwares. you do not see kernel with naked eye.

what is there in program counter initially?

Boot Order

The “Boot” Order

- **Which code does the CPU run initially and in which order ?**

1)BIOS

- Hardwired into ROM on motherboard by motherboard manufacturer. The CPU runs it “automatically” when turned on.
- BIOS *locates* the “Boot loader” code on PD/HD/CD/Network, etc. Loads it and “jumps” to it

2)Boot Loader

- Was put on the sector-0 of PD/HD/CD/Network etc by some end human user!
- When run, finds all OS installed on your HD and shows you options to boot
- You select some OS and press enter.
- Now Boot loader copies OS code into RAM and “jumps” to OS code

3)OS

- Takes “control” of hardware using hardware provided features
- Creates an “environment” for applications to run
- Loads some initial application (e.g. “init”) from HD into RAM and passes control to it (while retaining overall control)

4. Applications

grub’s code is written in C language, but is converted into machine code is different

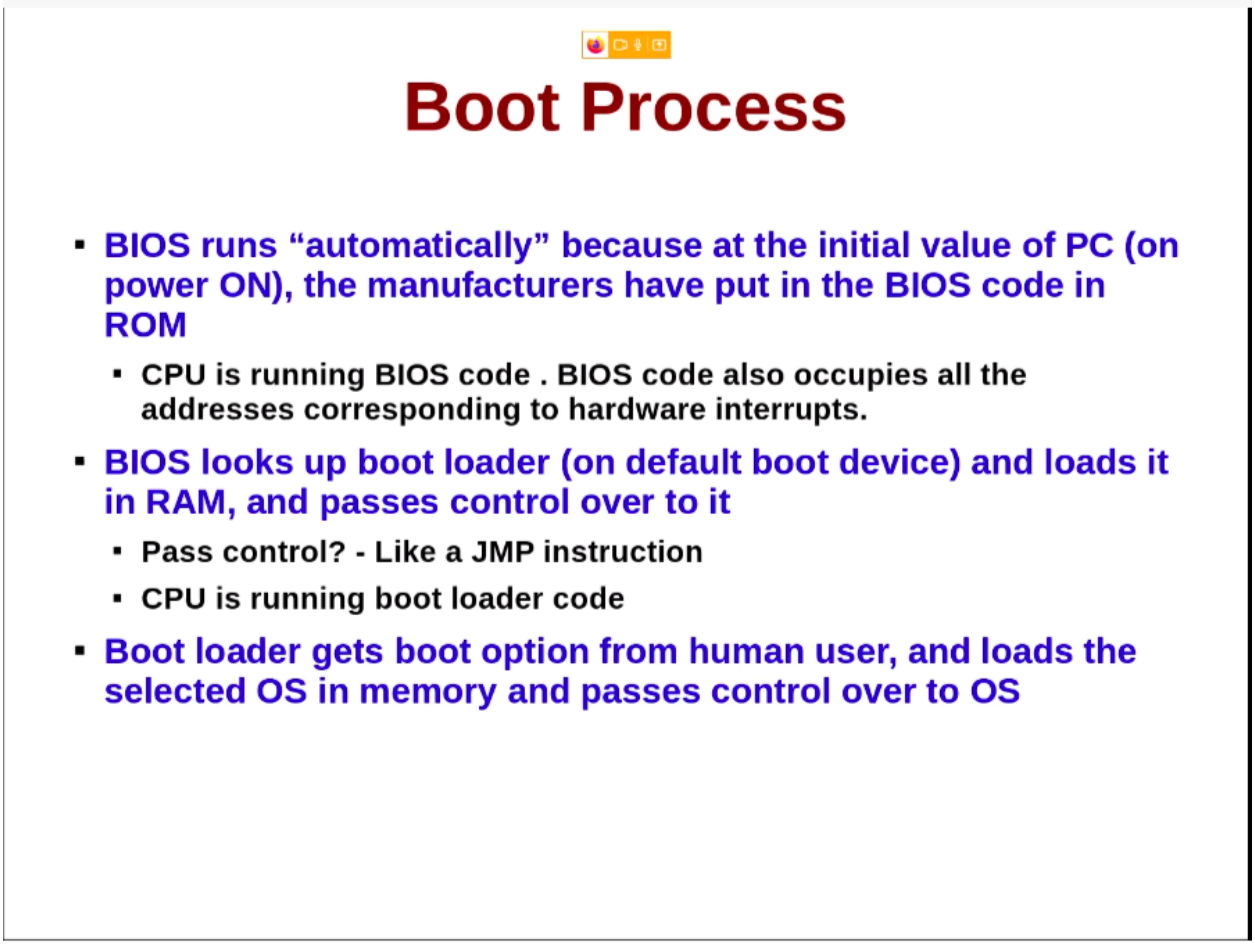
vmlinuz-5.11.0.41/vmlinuz-5.11.0.43 is the file in machine code that is loaded as kernel

the running of BIOS is done in hardware, in form of machine code

boot loader resides in sector 0, you can load OS directly by bios, if your OS is small enough to be in sector 0

Hardware Interrupt

the pc changes to a location(location in the Interrupt vector table) pre determined by processor manufacturer, at this address is Interrupt service routine which is part of OS



time shared cpu

Time Shared CPU

- Normal use: after the OS has been loaded and Desktop environment is running
- The OS and different application programs keep executing on the CPU alternatively (more about this later)
  - The CPU is time-shared between different applications and OS itself
- Questions to be answered later
  - How is this done?
  - How does OS control the time allocation to each?
  - How does OS choose which application program to run next?
  - Where do the application programs come from? How do they start running ?
  - Etc.

Multiprogramming

- **Multiprogramming**
  - A system where multiple processes(!) exist at the same time in the RAM
  - But only one runs at a time!
    - Because there is only one CPU
- **Multi tasking**
  - Time sharing between multiple processes in a multi-programming system

A multiprogramming system is not necessarily multi tasking, whereas a multitasking system is always multiprogramming

OS code executes only when

1. hardware interrupts
2. software interrupts
3. exceptions like page fault, divide by 0

OS is always running in the background → FALSE

OS is not always running, it is there in RAM, but not every time executing on CPU

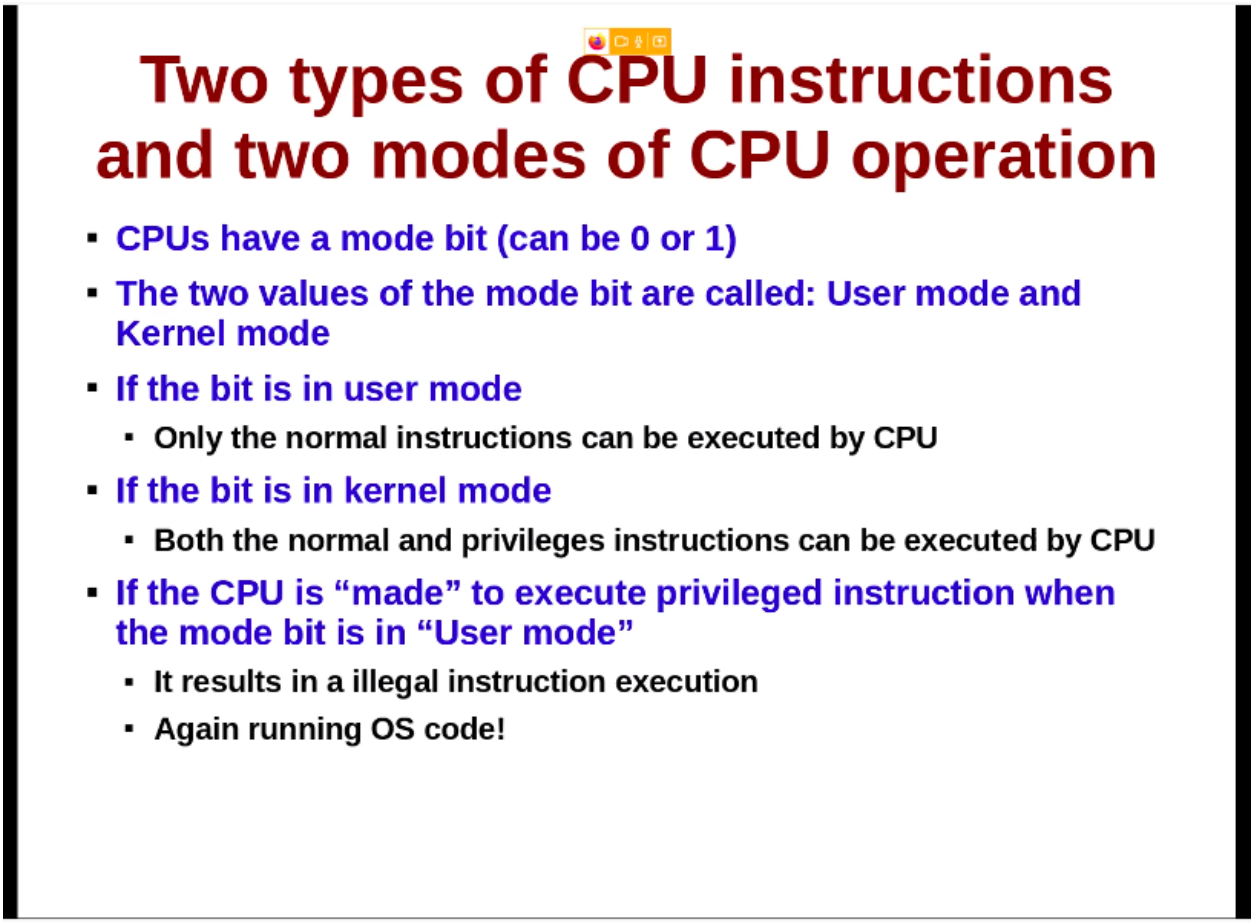
OS is event driven, events is written in list above

two types of CPU instructions and 2 modes of CPU

CPU instructions are divided in two types➖

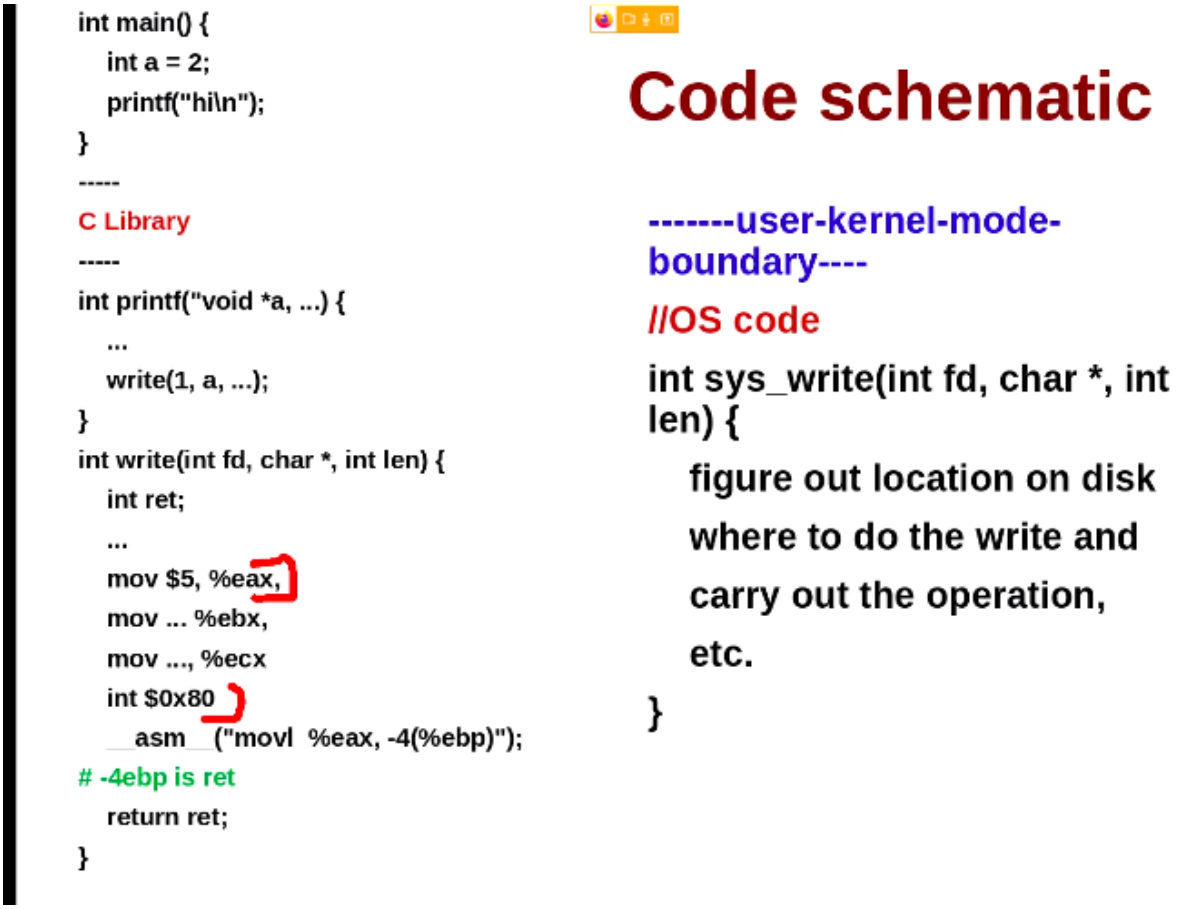
normal instructions → JMP,ADD,MV

privilliged instruction→ IN,OUT, INT



scanf runs a INT software interrupt to read value from keyboard

POSIX systemcall



fork() returns a value of 0 to the child process and returns the process ID of the child process to the parent process.

while executing exec, it overwrites the in memory image of currently running process by the new process.

After calling execl("/usr/bin/ls"), does the pid of ls remain same as pid of the caller process of execl

you can use -lm and -lncurses to include maths and ncurses library, while linking

linker,assembler and compiler are application program

gcc invokes these linker→ ld, assembler →as, for doing the job

.so is shared object, it is combination of many .o files and some extra information for the kernel.

.so is a dynamically linked file

Executable file format

An executable file needs to execute in an environment created by OS and on a particular processor

Contains machine code + other information for OS

Need for a structured-way of storing machine code in it

Different OS demand different formats

Windows: PE, Linux: ELF, Old Unixes: a.out, etc.

ELF : The format on Linux.

Try this

$ file */bin/ls*

$ file /usr/lib/x86_64-linux-gnu/libc-2.31.so

$ file a.out # on any a.out created by you

Exec() and ELF

When you run a program

$ ./try

Essentially there willl be a fork() and exec("./try", ...")

So the kernel has to read the file "./try" and understand it.

So each kernel will demand it's own object code file format.

Hence ELF, EXE, etc. Formats

ELF is used not only for executable (complete machine code) programs, but also for partially compiled files e.g. main.o and library files like libc.so.6

What is a.out?

"a.out" was the name of a format used on earleir Unixes.

It so happened that the early compiler writers, also created executable with default name 'a.out'

objdump can be used to disassemble machine code

Utilities to play with object code files

objdump

$ objdump -D -x /bin/ls

Shows all disassembled machine instructions and "headers"

hexdump

$ hexdump /*bin/*ls

Just shows the file in hexadecimal

readelf

Alternative to objdump

ar

To create a "statically linked" library file

$ ar -crs libmine.a one.o two.o

Gcc to create shared library

$ gcc hello.o -shared -o libhello.so

To see how gcc invokes as, ld, etc; do this

$ gcc -v hello.c -o hello

Calling Convention of functions-LIFO

- the only way to store local variables is on stack
- you cannot compile C program on a processor that doesnt support stack pointer
- Activation record=local var+parameters+return address
- Calling convention is really for→compiler
- when you store a variable at -4(ebp) how do you know it is from-4(ebp) to -8(ebp)
- who would set value for code segment, data segment, and offset registers if we ran OS on a 8086 processor
- EAX→32 bits, RAX→ 64 bits
- in linux int 0x80 is used for system calls
- IDT is in memory whereas IDTR register is in cpu

- why are global variables present in ELF file? → because they should be in memory before main executes as they are globals, as oppose to stack and heap which may keep on shrinking/growing in the program
- leave→ movl ebp,esp; pop ebp.[ebp=old ebp]
- ret→ Pop ‘return IP’ and go back in old function
- call → pushl (returnIP) into stack
- initial in calle function→ pushl %ebp; movl %esp,%ebp;

Function calls

LIFO

Last in First Out

Must need a “stack” like feature to implement them

Processor Stack

Processors provide a stack pointer

%esp on x86

Instructions like push and pop are provided by hardware

They automatically increment/decrement the stack pointer. On x86 stack grows downwards (substract from esp!)

Unlike a “stack data type” data structure, this “stack” is simply implemented with only the esp (as the “top”). The entire memory can be treated as the “array”.

Function calls

System stack, compilers, Languages

Compilers generate machine code with the 'esp'. The pointer is initialized to a proper value at the time of fork-exec by the OS for each process.

Langauges like C which provide for function calls, and recursion also assume that they will run on processors with a stack support in hardware

Convention needed

How to use the stack for effective implementation of function calls ?

What goes on stack?

Local variables

Function Parameters

Return address of instruction which called a function !

X86 instructions

leave

Equivalent to

```
mov   %ebp, %esp     # esp = ebp
pop   %ebp
```

ret

Equivalent to

```
pop %ecx
Jmp %ecx
```

Comparison

	MIPS	x86
Arguments:	First 4 in `%a0`–`%a3`, remainder on stack	Generally all on stack
Return values:	`%v0`–`%v1`	`%eax`
Caller-saved registers:	`%t0`–`%t9`	`%eax`, `%ecx`, & `%edx`
Callee-saved registers:	`%s0`–`%s9`	Usually none

Figure 6.2: A comparison of the calling conventions of MIPS and x86

From the textbook by Misruda

8086 Registers

- 16 bit ought to be enough ! (?)
- General purpose registers
  - four 16-bit data registers: AX, BX, CX, DX
  - each in two 8-bit halves, e.g. AH and AL
- Registers for memory addressing
  - SP, BP, SI, DI : 16 bit
  - SP stack pointer, BP base pointer, SI Source Index, DI Destination Index
  - IP instruction Pointer
  - Addressible memory: 2^16 = 64 kb

32 bit 80386

- Still possible to access 16 bit registers using AX or BX
  - Specifix coding of machine instructions to tell whether operands are 16 or 32 bit
  - prefixes 0x66/0x67 toggle between 16-bit and 32-bit operands/addresses respectively
  - in 32-bit mode, MOVW is expressed as 0x66 MOVW
  - the .code32 in bootasm.S tells assembler to generate 0x66 for e.g. MOVW
- 80386 also changed segments and added paged memory...

Summary of registers in 80386

- **Expected usage of Segment Registers**
  - **CS: Holds the Code segment in which your program runs.**
  - **DS : Holds the Data segment that your program accesses.**
  - **ES,FS,GS : These are extra segment registers**
  - **SS : Holds the Stack segment your program uses.**
- **All 16 bit**

X86 Assembly Code

- **Syntax**
  - **Intel syntax: op dst, src (Intel manuals!)**
  - **AT&T (gcc/gas) syntax: op src, dst (xv6)**
  - **uses b, w, l suffix on instructions to specify size of operands**
- **Operands are registers, constant, memory via register, memory via constant**

Examples of X86 instructions

AT&T syntax	"C"-ish equivalent	Operands
movl %eax, %edx	edx = eax;	register mode
movl $0x123, %edx	edx = 0x123;	immediate
movl 0x123, %edx	edx = *(int32_t*)0x123;	direct
movl (%ebx), %edx	edx = *(int32_t*)ebx;	indirect
movl 4(%ebx), %edx	edx = *(int32_t*) (ebx+4)	displaced

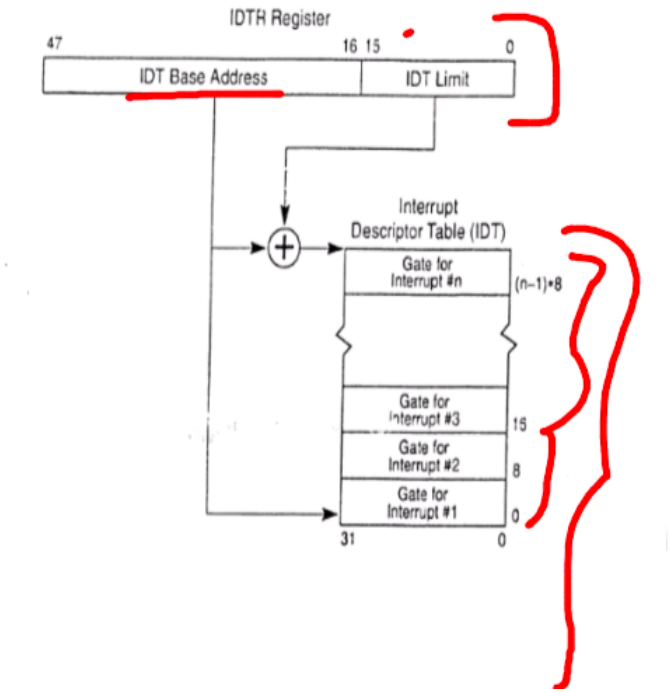
Privilege levels

- Changes automatically on
  - "int" instruction
  - hardware interrupt
  - exeception
- Changes back on
  - iret
- "int" 10 --> makes 10th hardware interrupt. S/w interrupt can be used to create hardware interrupt'
- Xv6 uses "int 64" for actual system calls

Interrupt Descriptor Table (IDT)

- IDT is an in memory table. IDT defines intertupt handlers
- Has 256 entries
  - each giving the %cs and %eip to be used when handling the corresponding interrupt.
- Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses.
  - Xv6 maps the 32 hardware interrupts to the range 32-63 and uses interrupt 64 as the system call interrupt

IDTR and IDT

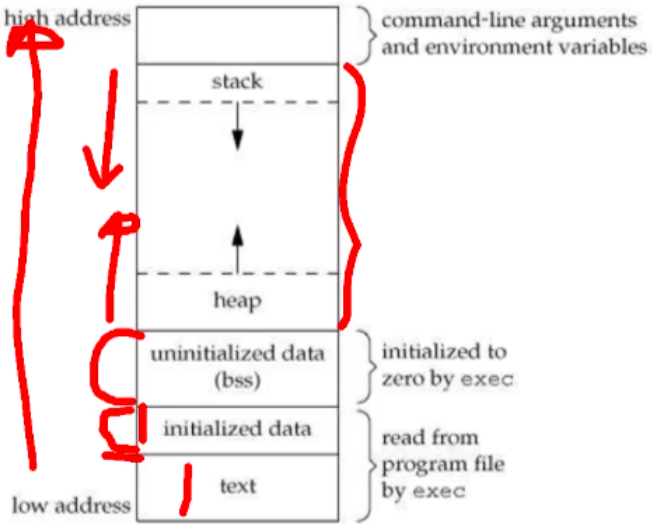


- **An entry in IDT is called a “gate”**
- **IDTR is a CPU register, IDT is in memory table**

memory layout as assumed by a compiler

- text is the code’s text that you write
- then is initialized data→ global and static variables
- uninitialized data (bss) , global variables that were not initialized are by default initialized to 0 and are stored in this region
- command-line arguments and environment variables are stored in high addresses
- text,data and bss are also present in ELF file

Memory Layout of a C Program



text, data, bss are also present in the ELF file
stack and heap are not present in ELF file

Text: object code of the program

Data: Global variables + (local/global) Static variables

BSS: Uninitialized data, globals that were not initialized

Stack: Region for allocating local variables, function parameters, return values

Heap: Region for use by malloc(), free()

Arguments, Environment variables: Initialized by kernel (during exec)

Real mode and protected mode

- **Beware: note the same as user mode and kernel mode!**
- **Backward compatibility was desired by Intel**
  - **80286 was 32 bit, 8086 was 16 bit**
  - **Binary encoding for 80286 was different, registers were 32 bit, etc; also more speed, different memory management hardware, etc.**
  - **But still they wanted their customers to keep running earlier object code on new processor**
  - **So they ensured that the processor boots up as if it was 16 bit 8086/8088. REAL MODE!**
  - **On running particular machine instruction sequence, the CPU will change to 32 bit . PROTECTED MODE!**

More on real mode

- addresses in real mode always correspond to real locations in memory.
  - Memory address calculation done by MMU in real mode: segment*16+offset
- no support for memory protection, multitasking, or code privilege levels (user/kernel mode!)
- May use the BIOS routines or OS routines
- DOS like OS were written when PCs were in the era of 8088/real-mode.

run `make qemu 2>&1 | tee make-commands.txt` to output the commands run by make qemu in a separate file.

Files generated by Makefile

- **xv6.img**
  - Image of xv6 created

```
xv6.img: bootblock kernel
        dd if=/dev/zero of=xv6.img
count=10000
        dd if=bootblock of=xv6.img
conv=notrunc
        dd if=kernel of=xv6.img seek=1
conv=notrunc
```

dd will copy 512Bytes for each iteration, 512*100000=4.9MB of 0 is coppied into xv6.img

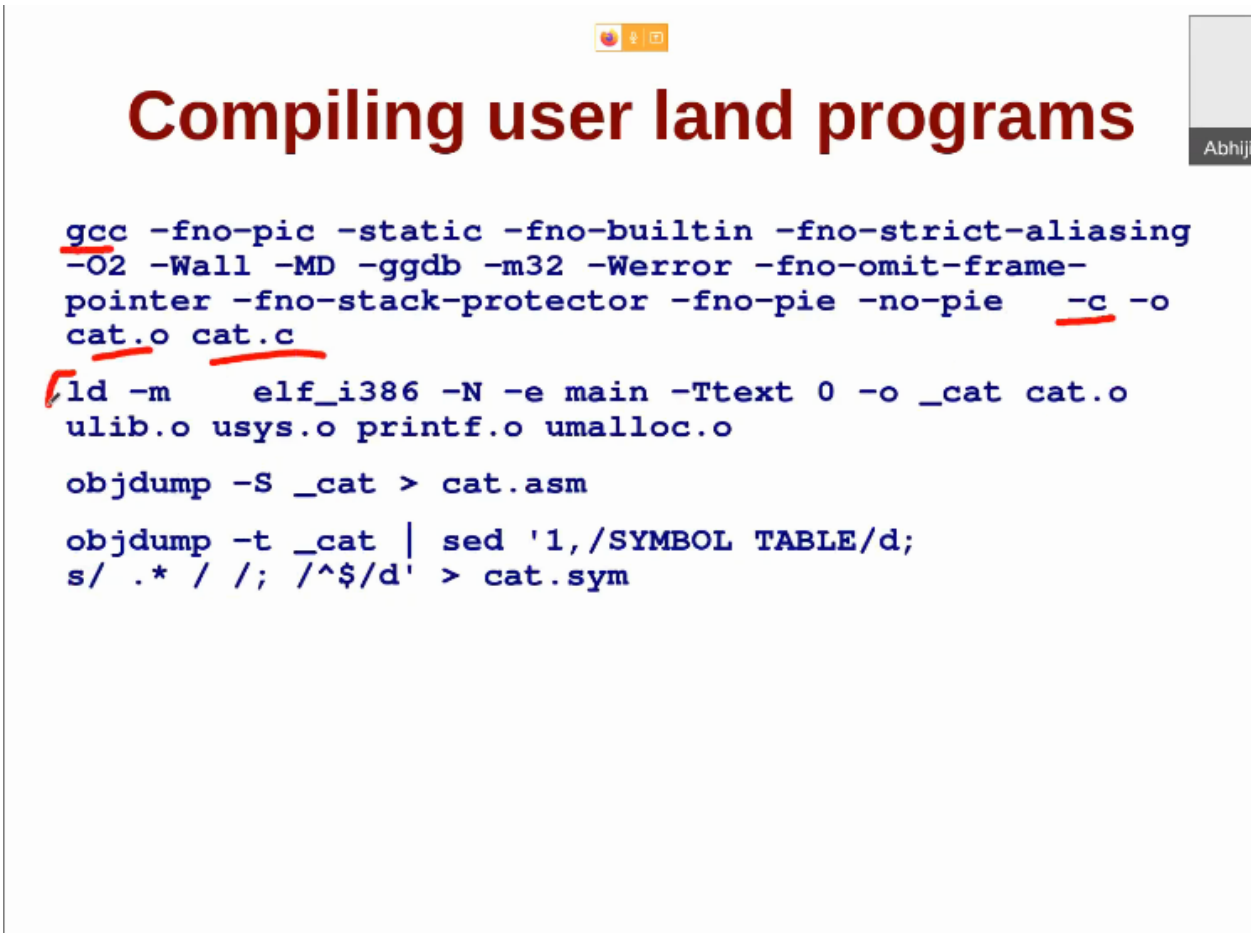
seek=1 is seek 1 block i.e 512Bytes,conv=notrunc is dont truncate the file,bootblock is overwritten on zeros

so xv6.img is concatenatioin of bootblock+kernel+ rest is zeroes copied from dev/zero

bootblock is bootloader, which is stored in first sector

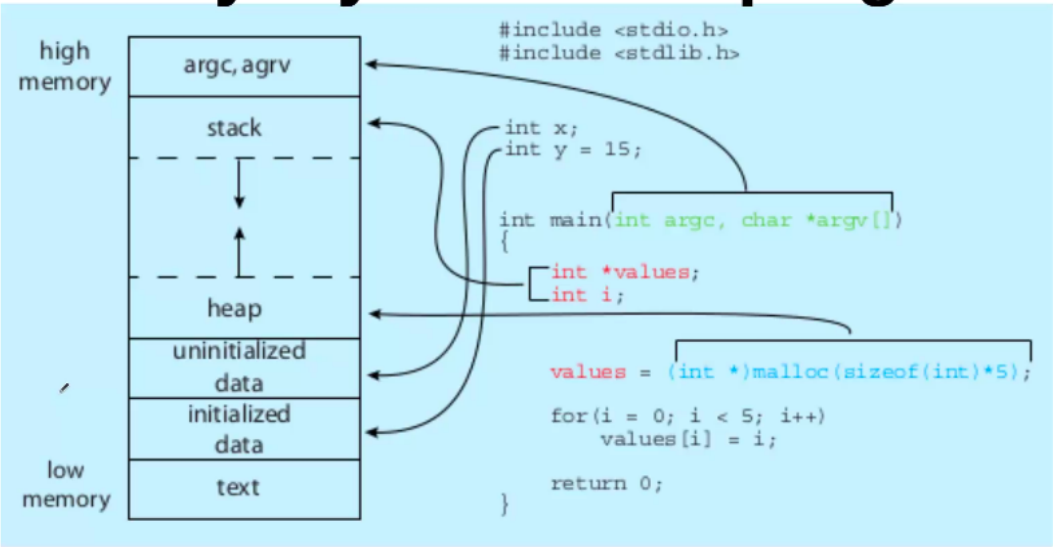
there are 2 hard disks for xv6, 1 containing kernel i.e xv6.img other is fs.img containing programs

read umalloc.c, printf.c, usys.c, ulib.c



memory management basics

Memory layout of a C program



$ size /bin/ls

text	data	bss	dec	hex	filename
128069	4688	4824	137581	2196d	/bin/ls

Desired from a multi-tasking system

- Multiple processes in RAM at the same time (multi-programming)
- Processes should not be able to see/touch each other's code, data (globals), stack, heap, etc.
- Further advanced requirements
  - Process could reside anywhere in RAM
  - Process need not be continuous in RAM
  - Parts of process could be moved anywhere in RAM

Different ‘times’

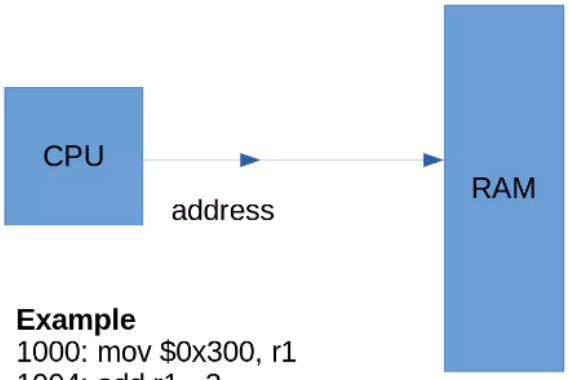
- Different actions related to memory management for a program are taken at different times. So let’s know the different ‘times’
- Compile time
  - When compiler is compiling your C code
- Load time
  - When you execute “./myprogram” and it’s getting loaded in RAM by loader i.e. exec()
- Run time
  - When the process is alive, and getting scheduled by the OS

Different types of Address binding

Which binding is actually used, is mandated by processor features + OS

- Compile time address binding
  - Address of code/variables is fixed by compiler
  - Very rigid scheme
  - Location of process in RAM can not be changed ! Non-relocatable code.
- Load time address binding
  - Address of code/variables is fixed by loader
  - Location of process in RAM is decided at load time, but can’t be changed later
  - Flexible scheme, relocatable code
- Run time address binding
  - Address of code/variables is fixed at the time of executing the code
  - Very flexible scheme , highly relocatable code
  - Location of process in RAM is decided at load time, but CAN be changed later also

Simplest case



Example
1000: mov $0x300, r1
1004: add r1, -3
1008: jnz 1000

- Problem with this solution
  - Programs once loaded in RAM must stay there, can't be moved
  - What about 2 programs?
    - Compilers being "programs", will make same assumptions and are likely to generate same/overlapping addresses for two different programs
    - Hence only one program can be in memory at a time !
    - No need to check for any memory boundary violations – all memory belongs to one process
- Example: DOS

Memory Management Unit (MMU)

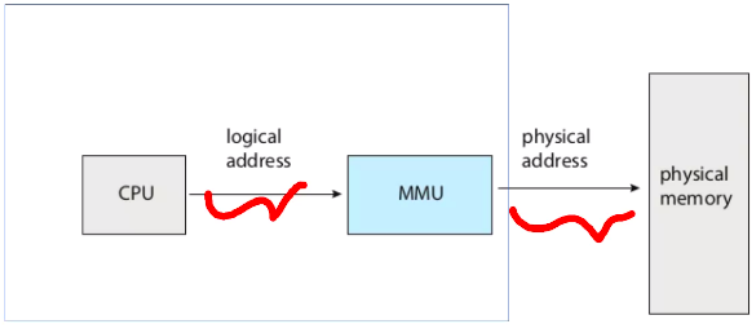
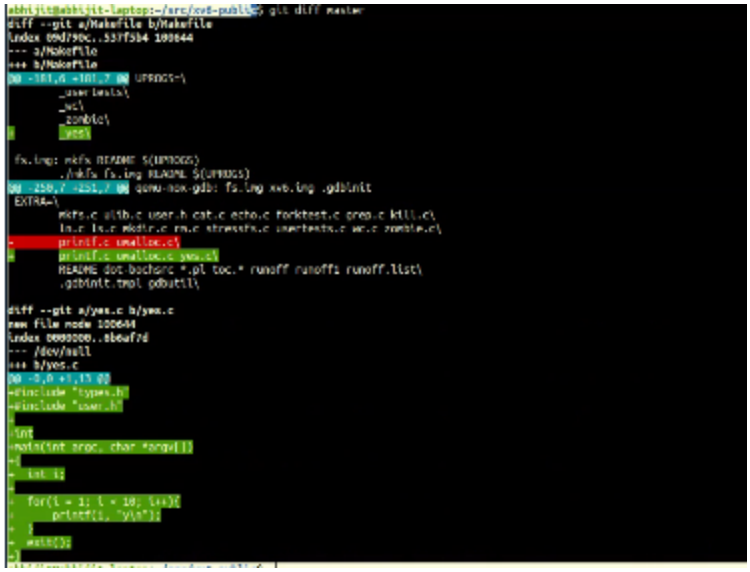
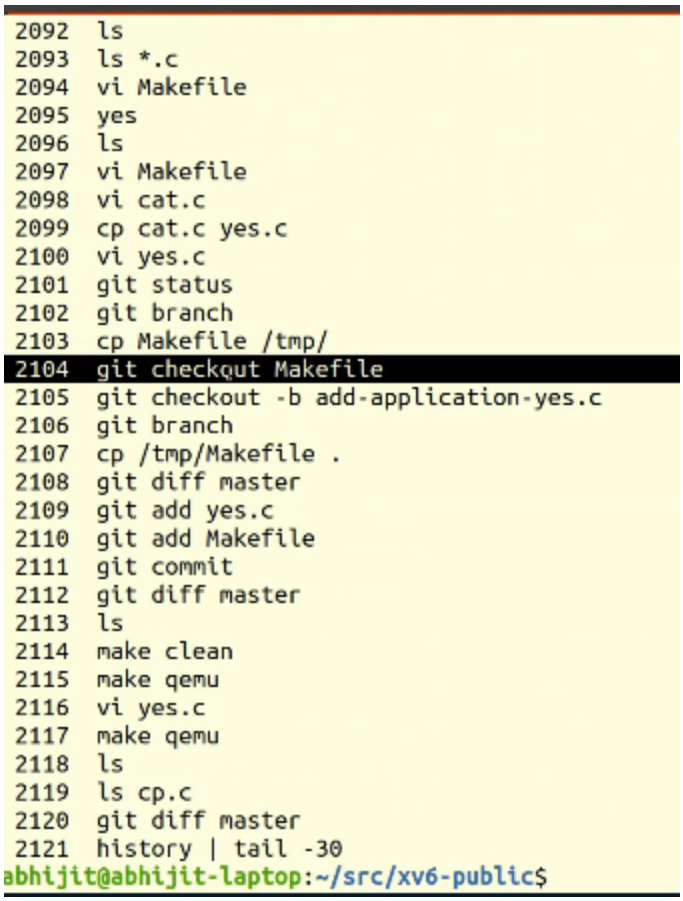


Figure 9.4 Memory management unit (MMU).

Problems with segmentation schemes

- OS needs to find a continuous free chunk of memory that fits the size of the “segment”
  - If not available, your exec() can fail due to lack of memory
- Suppose 50k is needed
  - Possible that mong 3 free chunks total 100K may be available, but no single chunk of 50k!
  - **External fragmentation**
- Solution to external fragmentation: **compaction** – move the chunks around and make a continous big chunk available. Time consuming, tricky.




2092 ls
2093 ls *.c
2094 vi Makefile
2095 yes
2096 ls
2097 vi Makefile
2098 vi cat.c
2099 cp cat.c yes.c
2100 vi yes.c
2101 git status
2102 git branch
2103 cp Makefile /tmp/
2104 git checkout Makefile
2105 git checkout -b add-application-yes.c
2106 git branch
2107 cp /tmp/Makefile .
2108 git diff master
2109 git add yes.c
2110 git add Makefile
2111 git commit
2112 git diff master
2113 ls
2114 make clean
2115 make qemu
2116 vi yes.c
2117 make qemu
2118 ls
2119 ls cp.c
2120 git diff master
2121 history | tail -30
abhijit@abhijit-laptop:~/src/xv6-public$

Effective memory translation in the beginning
At _start in bootasm.S:

%cs=0 %ip=7c00.

in bootasm.c , start is the 0x7c00 address

Real mode Vs protected mode

- Real mode 16 bit registers
- Protected mode
  - Enables segmentation + Paging both
  - 32 bit registers
  - can address upto 2^{32} memory
  - Can do arithmetic in 32 bits
  - Segment registers is index into segment descriptor table
  - Default segment registers used: for instructions CS, for data DS, for stack SS
  - Other segment registers need to be explicitely mentioned in instructions
    - Mov FS:$200, 30

lgdt is load into gdt register,

sta_r → read permission

X86 paging

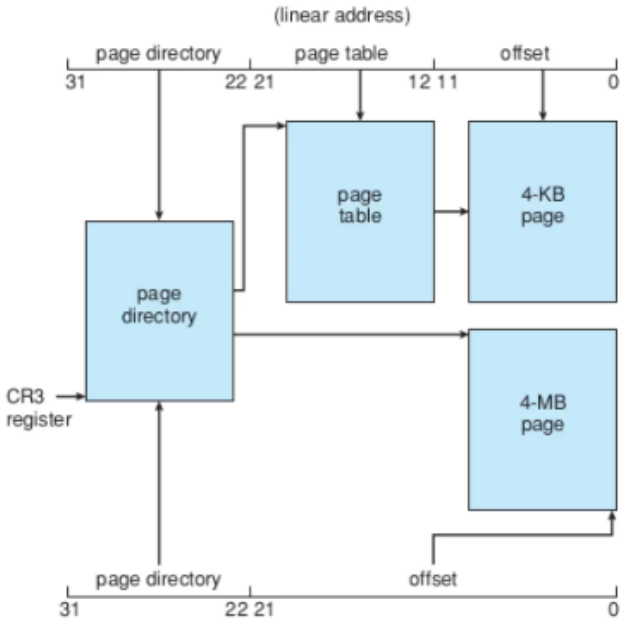


Figure 8.23 Paging in the IA-32 architecture.

last bit of CR0 enables protected mode

ljmp CS,IP

lline 65 is copying 7c00 to esp

at line 56 we move 2 to ax, as the segment register starts from bit 3,

insl copies multiple bytes from locatin given by disk controller