Lecture #2 OS Structure, Process abstraction

Anatomy of an OS
Anatomy of a process

****

Review -- 1 min

****

OS: 2 basic functions

- Coordination (security, communication, resource mgmt)
- Abstraction/Standard Services
  - Physical Reality → More convenient interface
  - -- easier for programmer
  - -- interoperability among programs

    I argued that "dual mode operation" would let us do these things

OS as complex system

- keeping things simple is key to getting them to work

We'll see these themes over and over again throughout this class.

History – change driven by technology

At first HW expensive, humans cheap → OS optimized for HW
(concurrency, overlap IO, batch system…)
Then, humans expensive, HW cheap → OS optimized for human (windows system, PC, interactive system,..)
Future: HW "free", networks everywhere
→ distributed services, communication, entertainment,…

Evaluation criteria/goals
-- reliability (challenging!)
-- performance (many metrics)
-- portability (many dimensions)

****

Outline - 1 min

****

(1) OS Structure: What is an OS? How do you run it?

- Key idea: Supervisor mode v. user mode
- {By end of day – understand 3 ways to invoke OS}

If time:
(2) OS structure: monolithic, microkernel, virtual machine
How to transfer control between process and OS; how to transfer control between processes
(3) Putting it together: Boot
(4) Abstraction: Process

- {By end of day – understand anatomy of a process}
- {By end of day – understand differences among process, program, address space, thread}
- linking and loading

****

Preview - 1 min
****

Today: OS structure, process abstraction
Then: Concurrency abstraction for several weeks:
implementation of this abstractions – how threads are implemented
Then – 2 weeks – how to program with threads
Then: Memory: relocation and protection

****

Lecture - 33 min
****

1. OS Structure:

What is an OS? How do you run it?

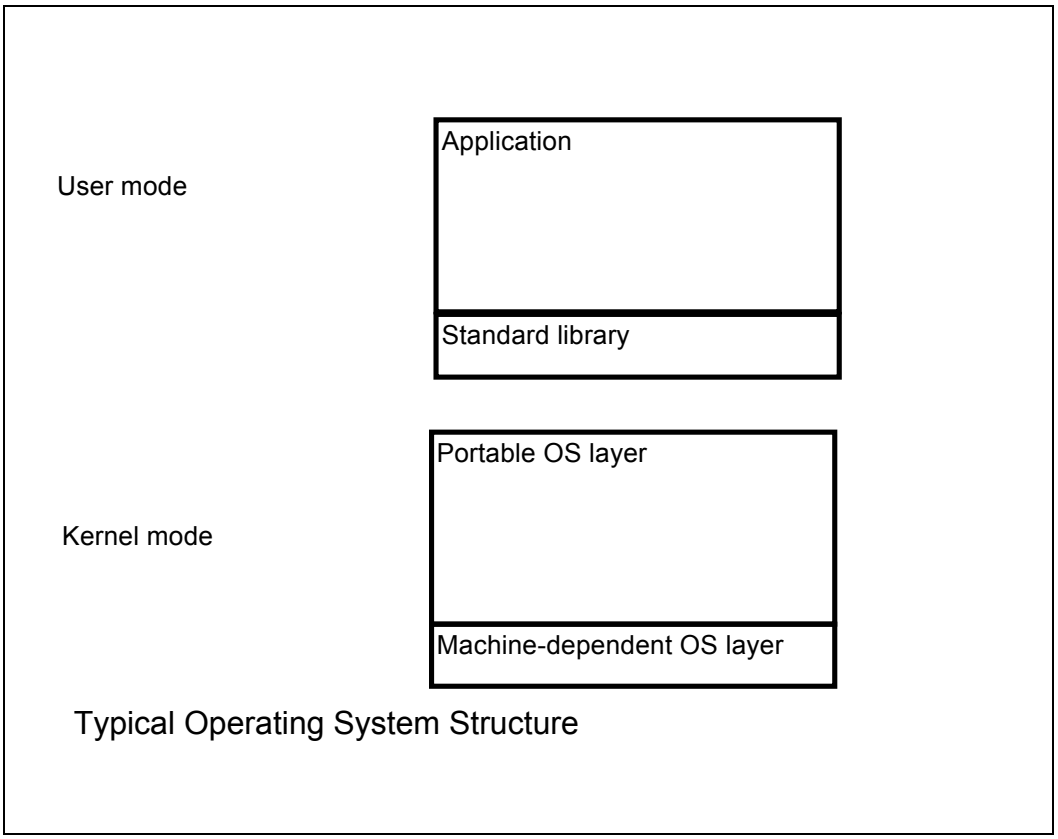
- Key idea: Supervisor mode v. user mode
- Case studies: internal OS structure

Note on class organization: often we will describe a "key idea" and then use examples/case studies to (a) better understand the key idea (see how the idea is applied, test understanding of the idea), (b) important knowledge in its own right

2. Dual mode operation

OS must control actions of applications
- (1) Communication, (2) protection, (3) resource mgmt

→ dual mode operation
"user mode" -- restricted (applications)
"supervisor mode" -- do anything you want (OS kernel)



Typical Operating System Structure

How do the security, communication, resource management facilities get reflected in dual-mode operation? (*draw in picture above)*

- Security: application can access its memory but not OS or other apps
- But OS can access any application's memory
- Communication: application can call OS (how does that work?) – then OS can read one application's memory and write another's → communication

- Resource management: OS can access hardware registers to, for instance, grow an application's memory allocation…

QUESTION: examples of things that kernel can do that applications should not be able to do/when do you want OS code to run?

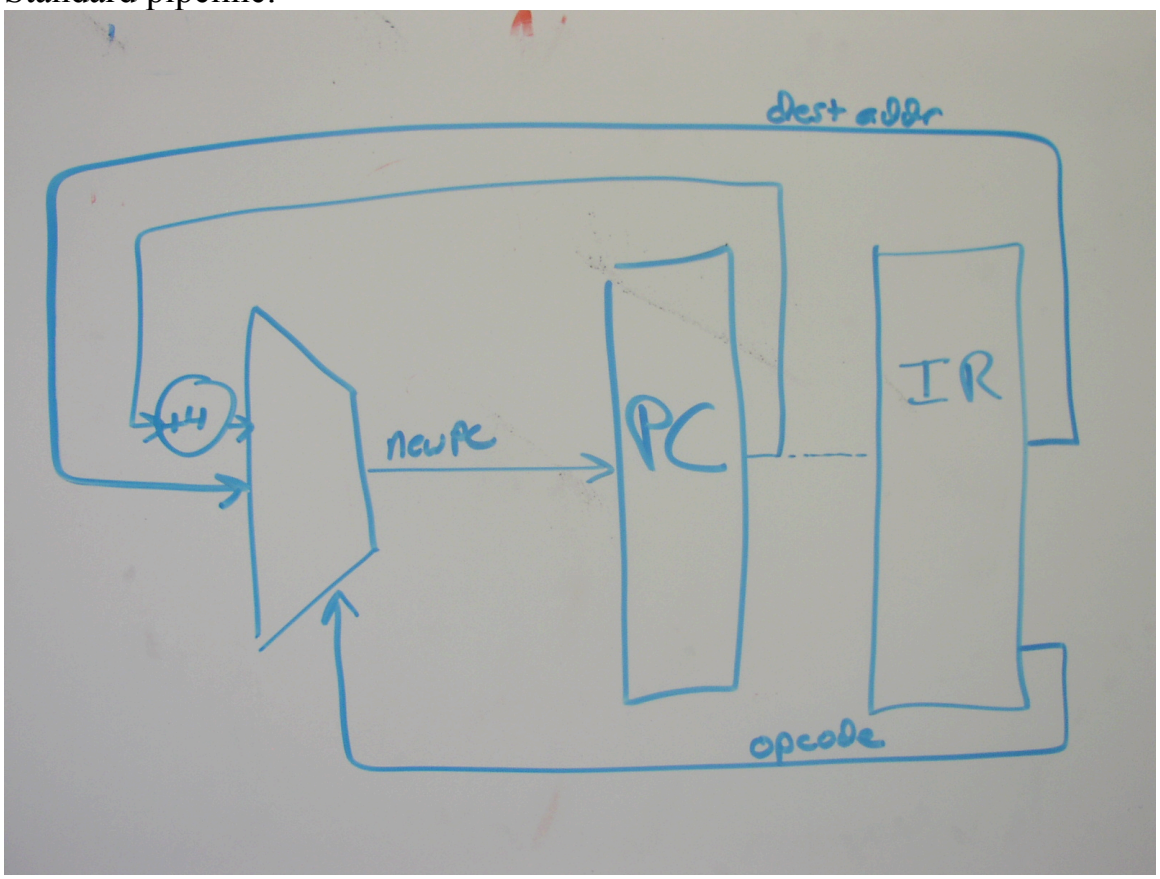
- *Security examples…*
- *Communication examples…*
- *Resource mgmt examples…*

To really understand this, need to know how to switch between user and supervisor mode

2.1 Switching from user to supervisor mode

1) interrupt: HW device needing service
2) exceptions: user program acts silly (e.g. divide by 0, bus error, etc)
3) trap: user program requires OS service (system call)

Standard pipeline:



Basic procedure:
(1) Switch to kernel
(2) kernel handler handles event
(3) resume suspended process

(1) Switch to supervisor mode

QUESTION: How would you implement dual mode (hint: hardware)?

→ *e.g., "supervisor" bit*

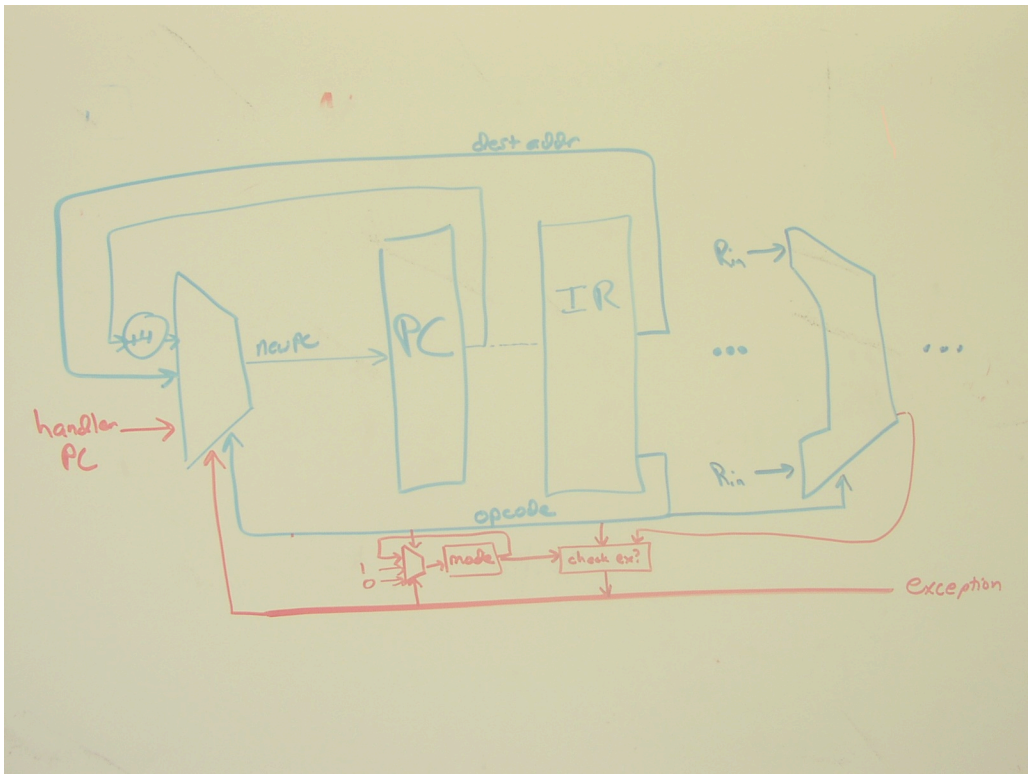
When bit on, HW can do "anything"; when bit off, restrict what HW can do.

- HW detects need to go to OS (for one of these reasons)

  → jump to well know OS address (handler) and switch on supervisor bit

QUESTION: How would hardware detect that an exception has occurred (e.g., user did divide-by-zero)? Interrupt? Trap?

QUESTION: How would hardware "jump to well known OS address and switch on supervisor bit"?



- HW/OS save user process's state
  What state? Registers (PC, stack pointer, other registers).
  Save it where?

- OS handler executes (with supervisor bit set)

QUESTION: How might HW help save state? How might HW get handler to execute?

(2) Supervisor handles request

- Dispatch: OS looks at program or machine state and decides what has been requested/needs to be done

QUESTION: How would this work? What arguments should handle_trap(), handle_interrupt(), handle_exception() receive? How would these arguments be supplied?

- OS jumps to appropriate handler routine and does its job

(3) Resume suspended process

- HW/OS restore user process's state and execute a RTI exception to jump back to user

QUESTION: What should RTI do? How differ from regular "return"?

QUESTION: So, what does it mean to "run an operating system"?
ANSWER: Start running, initialize a bunch of data structures, then just sit there until some event happens. In some old OS, there was not even a "kernel thread" – everything was reactive; new stack created on each event; now, there may be threads in OS, but they all are waiting for some event to trigger work…

x86 case study

Boot/OS initialization:

initially in supervisor mode
use supervisor-only instructions to specify
-- register that points to array of handlers
(handler i handles interrupt/exception of type i)
-- register that points to exception stack
-- register that points to supervisor page tables

[[big **picture**: hardware [PC, stack pointer, registers, mode, exception stack pointer, interrupt vector pointer], kernel [interrupt vector, handler code, stack], user code [code, stack]]

On exception i
-- HW:
set supervisor bit,
start using supervisor page tables,
save a few key registers on exception stack (e.g., stack pointer (`esp`, `ss`), `eflags` [includes supervisor/user bit, processor status], PC (`eip, cs`))
jump to handler[i],
set stack to exception stack,
-- typical SW for handler[i]:
save remaining registers to stack [`pusha` instruction],
run code to handle i,
restore registers from stack [`popa` instruction]
`reti`
-- HW (`reti` instruction)
restore `eflags` (includng supervisor/user bit), PC (`eip, cs`), stack ponter (`esp, ss`) from stack to processor registers

2.2 System calls, signals -- view from a process

*QUESTION: Using this mechanism, how does a process do a **system call**?*

system call -- request by user-level process to call a function in the kernel
e.g., gettimeofday(), read(), fork(), exit()

QUESTION: Using this mechanism, how does a process communicate with another process?

signal -- request by kernel to call a function in the user-level process
e.g., SIGALRM, SIGHUP, SIGUSR1, ...)

QUESTION: Many OS's allow user-level exception handlers (explain idea). How would we implement user-level exception handler?(called "signal handler" in unix)

process is running "normal code"
[something happens; os does WHAT?]
process's signal handler executes
process resumes running "normal code"

Notice -- signal:process::interrupt:kernel

[[picture -- timer interrupt]]
[[HW -- PC, SP, interrupt vector, exception stack pointer; kernel handler code, stack; user-level normal code, normal stack, handler code, handler stack]]

[[details:]]

setup: process tells kernel: signal stack, handler address
invoke: kernel saves process's interrupted state to signal stack and then starts executing process's signal handler with signal stack

QUESTION: how would you use reti to do this?

finish: process handler code restores state from context saved to signal stack

QUESTION: In posix, OS can save process state on current stack or "exception stack" within process, but OS exception handler **always** switches stacks and uses a dedicated signal stack rather than current stack. Why?

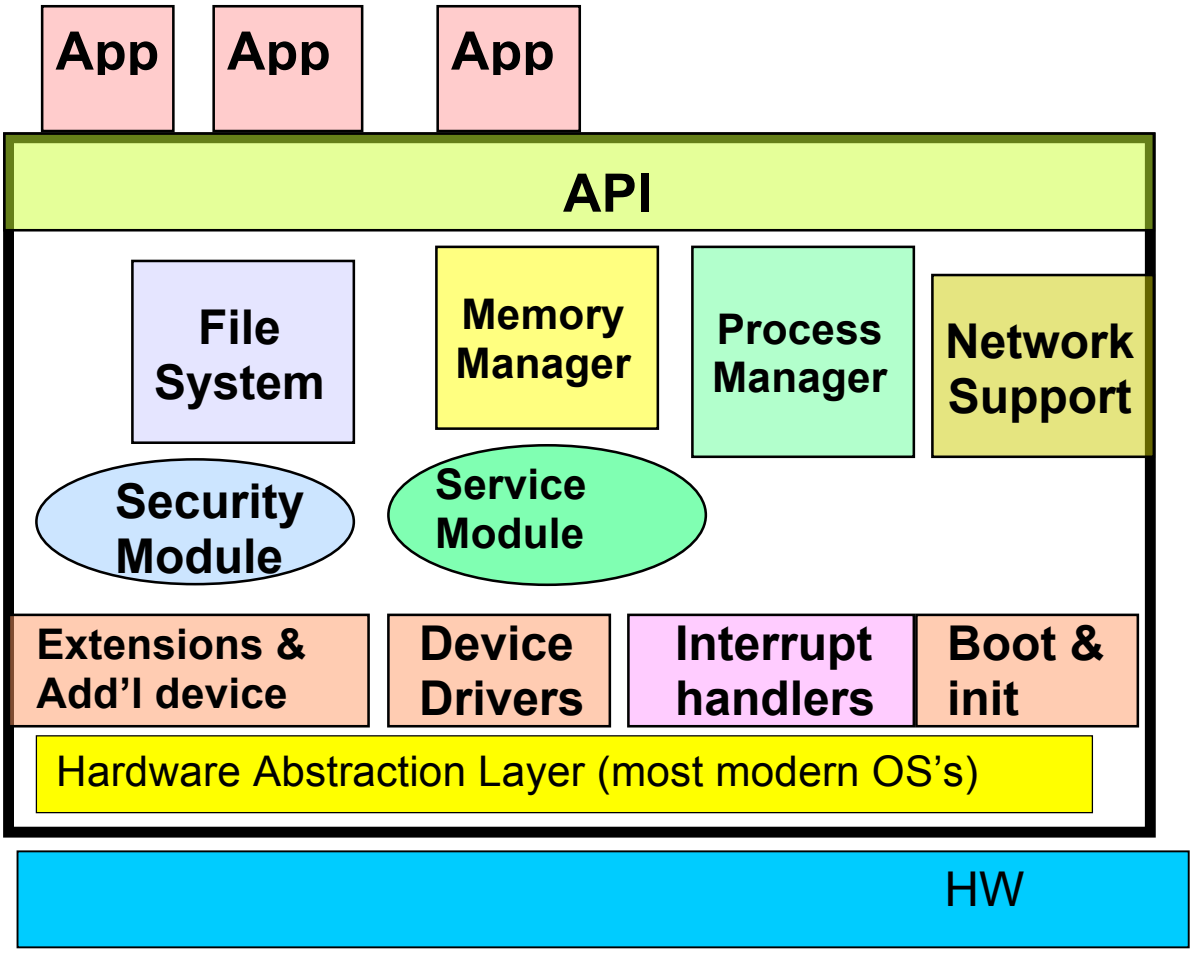
3. Internal OS Structure

OS mediates between HW and applications
Provide “Nicer” abstraction of physical HW

Structure of OS affects abstractions, implementation

Examine 3 common structures to (a) understand switching between kernel/user, (b) understand how “abstraction” function of an OS can be built and (c) discuss design choices in OS structure.

3.1 Monolithic structure (e.g., Unix)



- Question: Suppose application wants to access OS service (e.g., read block from file, allocate page of memory, etc.)
- **System call** – application-to-kernel request

Abstraction: procedure call
Implementation:
(1) application makes library procedure call *read(fd, buffer, length)*
(2) library converts to system call
  - syscall.h defines system call numbers *e.g., SYS_read = 3*
  → *syscall(SYS_read, arg1, arg2, arg3)*
  each OS will have calling convention (e.g., "put system call number in R1, args 1-2 in registers 2-3, and any remaining arguments on the stack")
  lw R1, callNum
  lw R2, arg1

lw R3, arg2
push arg3
trap

(3) trap jumps to OS handler, which grabs arguments, handles call, and returns

Question: Suppose OS is written in C or C++, then handler needs a stack.

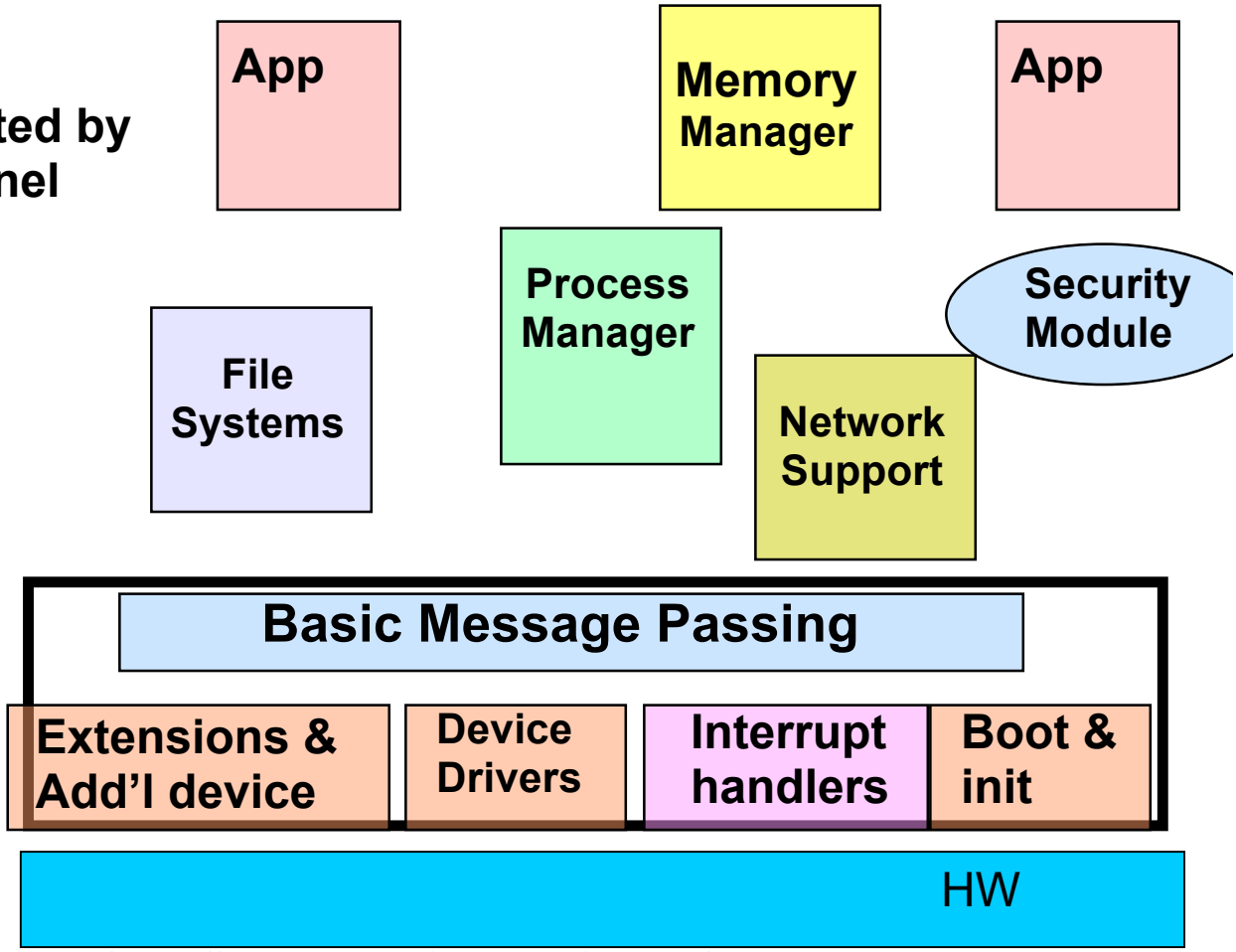
- Where should stack be (is it OK to use application stack?) What has to happen on trap?
- What should handler stack look like on call? (handler call is first stack frame, empty below that)
- What happens to stack if reti happens when we are several layers deep in procedure calls?
- What if kernel is handling an interrupt and another interrupt arrives? (reset stack pointer? Block interrupt? Keep building on kernel stack?)

3.2 Microkernel (e.g., Mach, QNX, …)

Main OS functions implemented by out-of-kernel servers

OS main functions implemented by out-of-kernel servers

e.g.: QNX,



Note: only small amount runs in "supervisor mode"

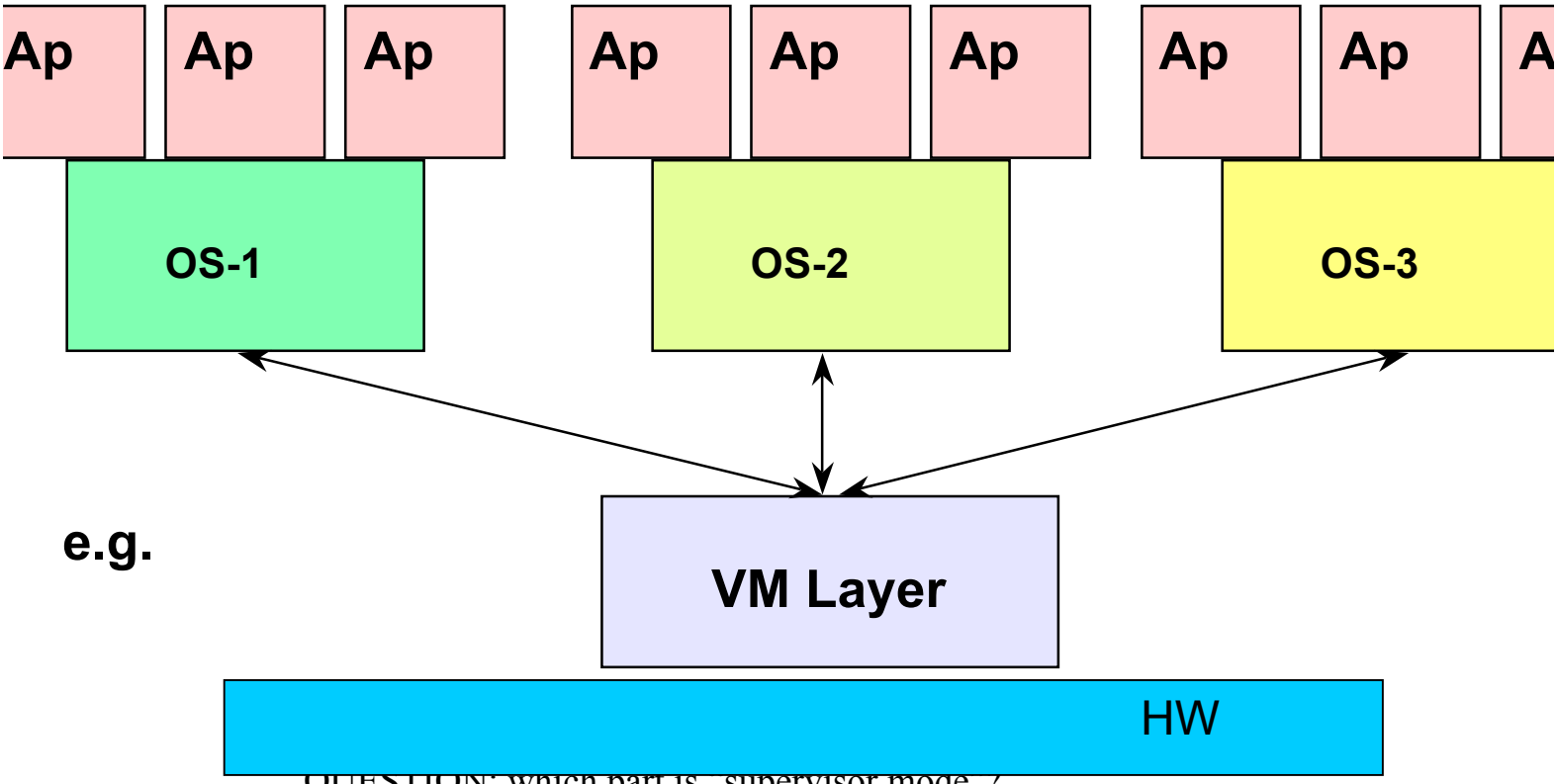
QUESTION: how can file system get access to disk drives?
How can file system security be maintained?
→ out-of-kernel servers are still "privileged" even though they don't run in supervisor mode

QUESTION: What are advantages of microkernel? Of monolithic?

Windows NT is "modified microkernel" – microkernel modularity as design methodology, but implement modules in kernel process for performance.

3.3 Virtual machine (e.g., OS360)

Virtual machine presents exactly the HW interface, but virtualizes the HW for protection
Independent OS's run on these virtual machines (e.g., Linux, NT, Solaris all in one box)



QUESTION: which part is "supervisor mode"?

QUESTION: what happens if os1 tries to execute a privileged instruction?

QUESTION: what happens if app1 in os1 calls trap?

****

Announcements

HW1 "due" Friday

TA office hours -- see web page

Newsgroup

HW1

Project 1 – discuss

Software engineering attitude -- discuss

---

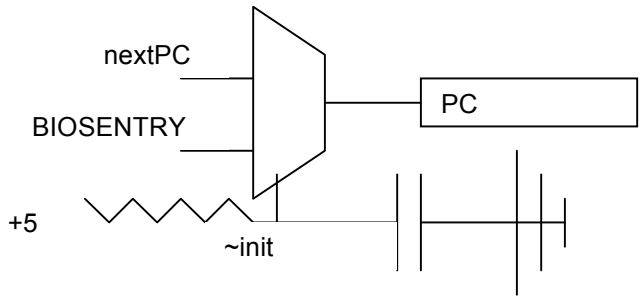
4. Putting it all together: OS in action

OS is just a program. What happens when you turn power on?

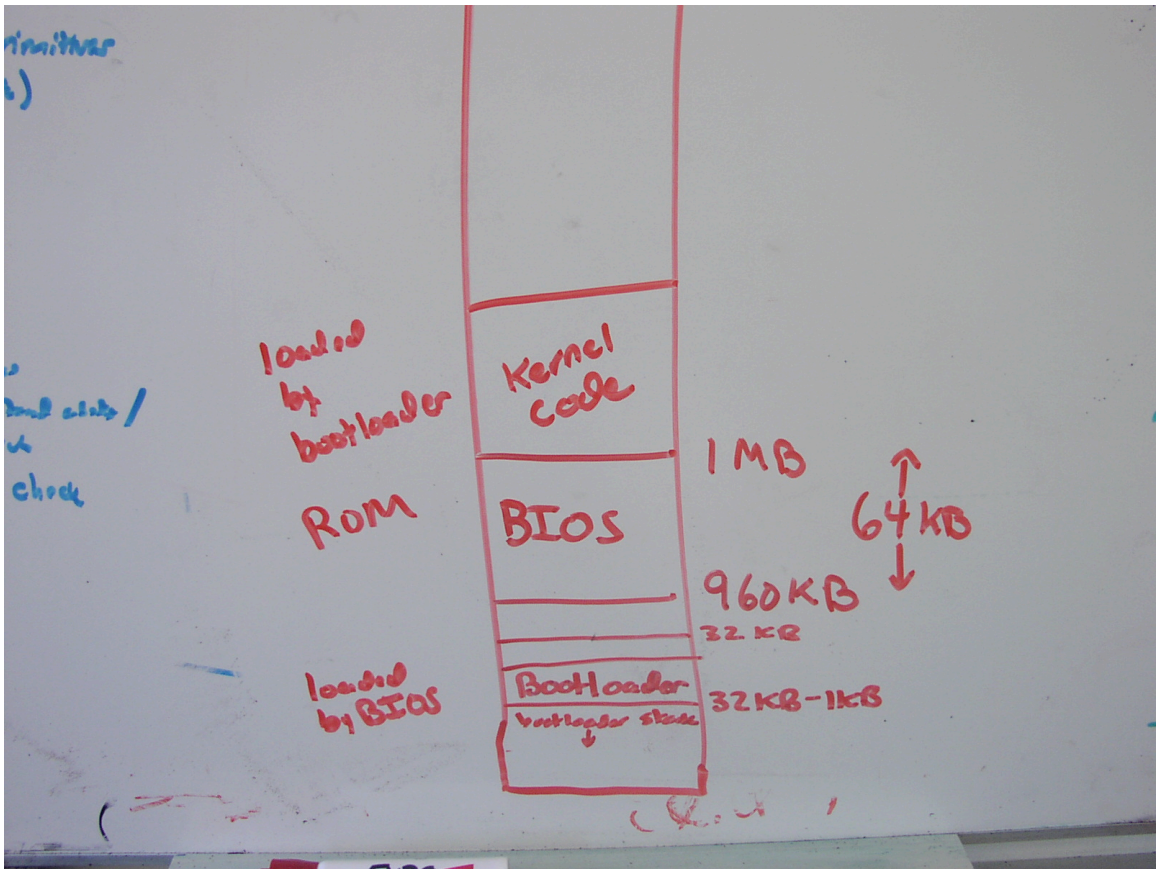
1) CPU loads boot program from ROM (e.g. BIOS in PCs)

You're on a desert island and need to build a computer…

- ROM is easy (solder some wires…)
- Need to arrange so that first instruction the CPU executes is BIOS entry point in ROM
  (Alternatively, on boot, state machine copies ROM to pre-specified location in RAM…exercise for the reader…)
  - Entry point is just another hard-wired (literally) constant
  - Initialization circuit to set PC…



Physical address layout for JOS boot:



2) Boot program (BIOS)
   - Examine/check machine configuration (# CPUs, how much memory, # and type of HW devices, etc)
   - Build configuration structure describing the HW
   - Load the bootloader
     - From "well-known" location (e.g., first 512 bytes of hard disk)
     - To "well-known" memory location (0x7c00 for x86 … 32KB – 1KB)
   - Jump to bootloader code (well-known entry point…0x7c00 for x86)
3) Bootloader (JOS: see boot/boot.S, boot/main.c)
   - Initialize stuff
     - SP = Initial stack at well-known location (JOS: 0x7c00 – code starts here and goes up. Stack starts here and grows down.)
     - X86 cruft: boot in "real mode" – only 1MB of memory visible, no virtual memory → make more than 1MB visible and set up initial segmentation table

- JOS Note: first part of bootloader is assembly (boot.S) but once we've got 32-bit mode set and a stack, we can start running C code, so we jump to the function bootmain in boot/main.c
- Read OS from disk, jump to well-known entry point
  - JOS bootloader
    - Read sector 2 of disk (elf header) to 0x10000 (65KB) (scratch space)
    - Parse *elf header* (see inc/elf.h) to determine size of rest of OS on disk and where to load the rest of the OS in memory
    - Load OS to specified location from sectors 2…size on disk to elf-specified locations in memory (code segment goes to 0x00100000 → 1MB)
    - Jump to elf-specified entry point (0x0010000c – corresponds to `_start` in `kernel/entry.S`)

4) OS initialization
   - Initialize registers (stack, frame pointers)
     - now we can run C!
   - Initialize virtual memory system
     - See below
   - Initialize kernel data structures
   - Initialize state of HW devices
   - Creates a number of processes to start operation (e.g. daemons, tty in Unix/windowing system in NT)
   - How does this work?

5) Start initial processes

Boot creates first proc ("sysproc" in unix)

Q: How would an OS create a new process?

The first process creates other processes
The creator is called the parent
The new process is called a child
The relationships among processes can be expressed as a tree

In Unix, the second process is called *init*
It creates the ttys (terminals)
It creates daemons (rlogin, smtp (mail), DNS, finger, …)
It controls system configuration
It should never die

When you log in (to a daemon/tty), it creates a *shell*
your *shell* process can create new sub-processes (login->shell->emacs…)

(6) Once OS is initialized
- Run user programs if available; else run low-priority user-level *idle-process*
  - In the idle process
    - infinite loop (Unix)
    - system mgmt & profiling
    - halt processor and enter low-power mode (notebooks)
    - compute some function (DEC's VAX VMS computed Pi)
  - OS wakes up on:
    - Interrupts from HW devices
    - Traps from user programs
    - Exceptions from user programs

4.1.1 [372H] JOS Linking and loading

Program source has *symbols* "main()", "A()", "X"
Hardware knows *addresses* (absolute or relative)
What address should program store for main, A, or X?
Depends on where program eventually loaded

Point is – compiler, linker need to translate symbols (procedure names, constant names, loop boundaries) to addresses

See call to memset() in lab1/kernel/init.c:i386_init() v. lab1/obj/kernel/kernel.asm;

```
f010015d:   e8 8c 13 00 00          call   f01014ee <memset>
```

try objdump lab1/obj/kernel/kernel | grep memset

→ Code all depends on certain stuff landing at certain addresses
Linker encodes these dependencies in the code
Loader is supposed to honor these dependencies by loading at the specified address

Linking and loading

Link address – address where program **expects** to be loaded (compile-time)
Load address – address where actually is loaded (run-time)

Simple case: compile one source file

- linker can assign base address for "start" (e.g., 0x01000)
- all symbols assigned an address (e.g., lw A; A stored at start+0x100 → lw 0x01100)
- Put link addres in header as expected load address

Variation: Position-independent code

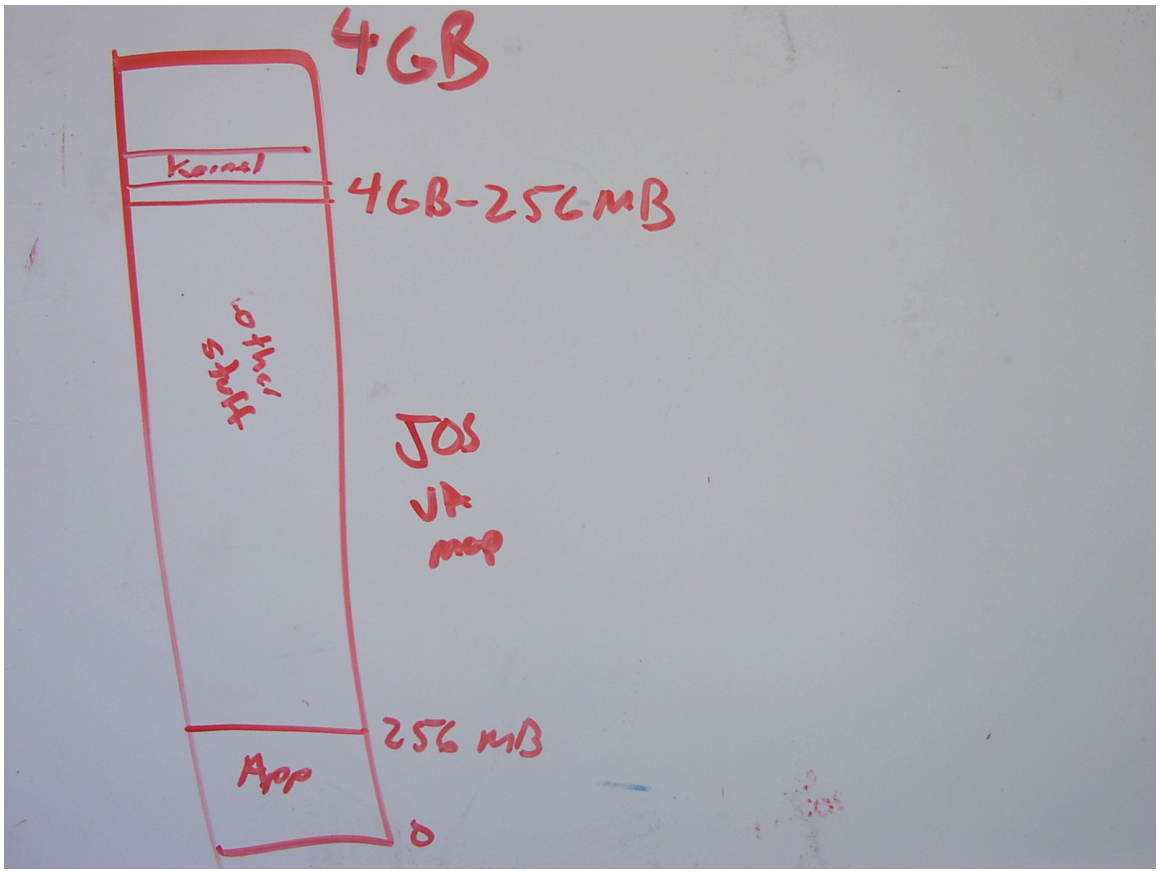
More general case: Link multiple .o files

- Create each .o file by compiling one source file
- Instead of hard-coding link/load address at compile time, **symbol table** (in header or pointed to in header) has list of offsets of symbols (e.g., "A" = start+178), list where symbols referenced (e.g., "A used at start+47")
- Linker assembles .o files into one executable starting at some link address; update each file accordingly
- Put link address in header as expected load address

[More general case – instead of specifying link address in header, specify symbol table in header; loader also updates addresses in binary]

[Another key variation: Dynamic linking – link shared library into running binary.]

Issue in JOS – want kernel to run at high *virtual address* 0xf0100000 (4GB – 256MB + 1MB) so that applications can run at low virtual addresses (first 256MB) and have disjoint addresses from kernel



We'll discuss virtual addressing more starting next week.

I bring this to your attention now b/c – do you see the issue?
Where did bootloader load kernel? (load address)
Where does kernel code expect to be loaded (link address)

Why is this OK? Short answer

- bootloader jumps to load address of kernel after loading kernel
- first thing the kernel does is run some gnarly assembly code (entry.S) that is aware of both the link and load address and of the discrepancy. This code sets up virtual addressing so that the VA 0xF0100000 (the link address) maps to PA 0x00100000 (the load address) and then jumps to "regular" C code that runs using link addresses (and

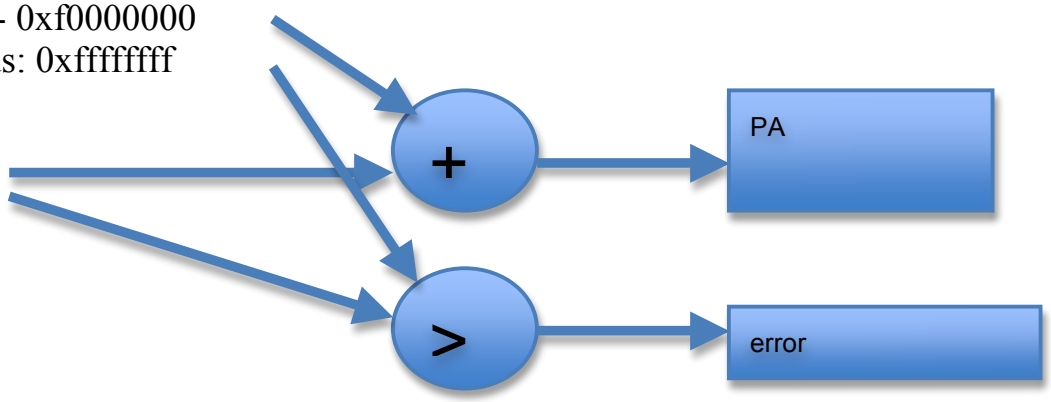
the VA->PA translation means that link addresses refer to the expected data)

- You will walk through this more carefully in the lab, but I wanted to give a high-level picture now…

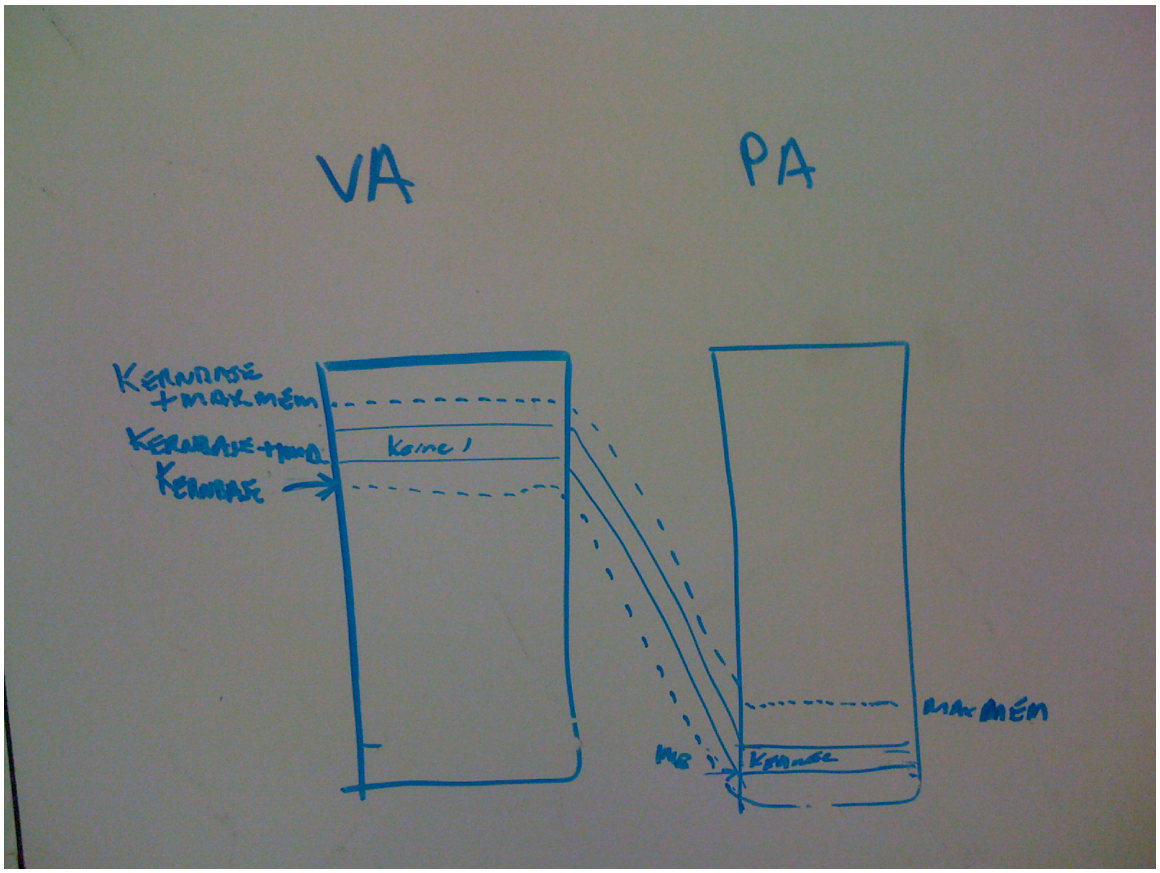
Note: JOS kernel sets up virtual memory in two steps

Lab1:
Initially kernel loaded at VA = PA = 1MB
But kernel "wants" to run at VA 0xf0100000 (4GB – 256MB)
→ entry.S sets segmentation table to shift all addresses by 0xf0000000

Seg. Register
Base: - 0xf0000000
Bounds: 0xffffffff



Picture:



LAB 2
Kernel starts up paging
Creates page table so that kernel VAs still maps to same PAs

5. Process abstraction

Main Point: What are processes?
How are process, programs, threads, and address spaces related?

Up until now, we've drawn lots of pictures with "user level process" as a black box. Let's talk about what is inside. 2 key ideas – state and concurrency – will be major focus of next 2 main blocks of this class

~~5.1 Motivation: Concurrency Abstraction~~

~~Hardware – single CPU, I/O interrupts~~
~~API – users think they have machine to themselves~~

OS has to coordinate all of the activity on a machine -- multiple users, I/O interrupts, etc

Picture: A bunch of jobs running and doing I/O -- os executes a few instructions here, then a couple of those, and now the data showed up from disk, so give it to the job, …

Challenge: How can it keep all these things straight?
Solution: Decompose hard problem into simpler ones. Instead of dealing with everything going on at once, separate them and deal with them one at a time
You've seen something like this before:
Analogy -- procedure calls "local variables" → registers, stack

- within procedure insert(), "r1" might mean "pointer to object to be inserted"; sp - 32 might mean "pointer to current queue"
- within procedure printf(), "r1" might mean "pointer to format string"; sp-32 might mean "the third variable to be inserted into the string"
- But insert() can call printf() {and vice versa} without being confused

QUESTION: How does it work?

What are key ideas?

1) storage space for "inactive" variables
2) standard procedures to move "inactive" variables from storage space to registers and vice versa

Switching between threads/processes is similar (but a little more involved because there is more state to swap back and forth)

5.2 Processes

Last few meetings, I've drawn picture of a bunch of boxes over an OS over hardware.

These boxes are meant to represent "processes"

I've talked about properties of the processes -- "keep in box" (isolation), "let out of box (in controlled ways)" (communication), "multiplex boxes on hw" (resource mgmt)

I've talked about how you can go from running process to running OS and back (dual mode operation, trap/interrupt/exception, etc.)

Have not formally defined the box/process. Do that now.

Process: Operating system abstraction to represent what is needed to run a single program
(This is the traditional UNIX definition)

more formally (traditional)
Process: a sequential stream of execution in its own address space

5.2.1 Two parts to a process

1) **sequential execution**: no concurrency inside a process – everything happens sequentially [we will generalize this later]
2) **address space**: all *process state*; everything that interacts with a process

In practice, this means that "inside each box" [**DRAW**] -- registers (including PC, stack pointer), code, stack, globals, heap, ...

--> I can reason about each process independently
-- each process has *everything* I need to reason about the running program
    -- almost everything -- "controlled communication" e.g., system calls etc.
--> Ignore other processes
-- I told you how to transparaently switch between process and OS
--> Ignore OS

QUESTION: I'm running emacs on my linux box. What state is part of that process?

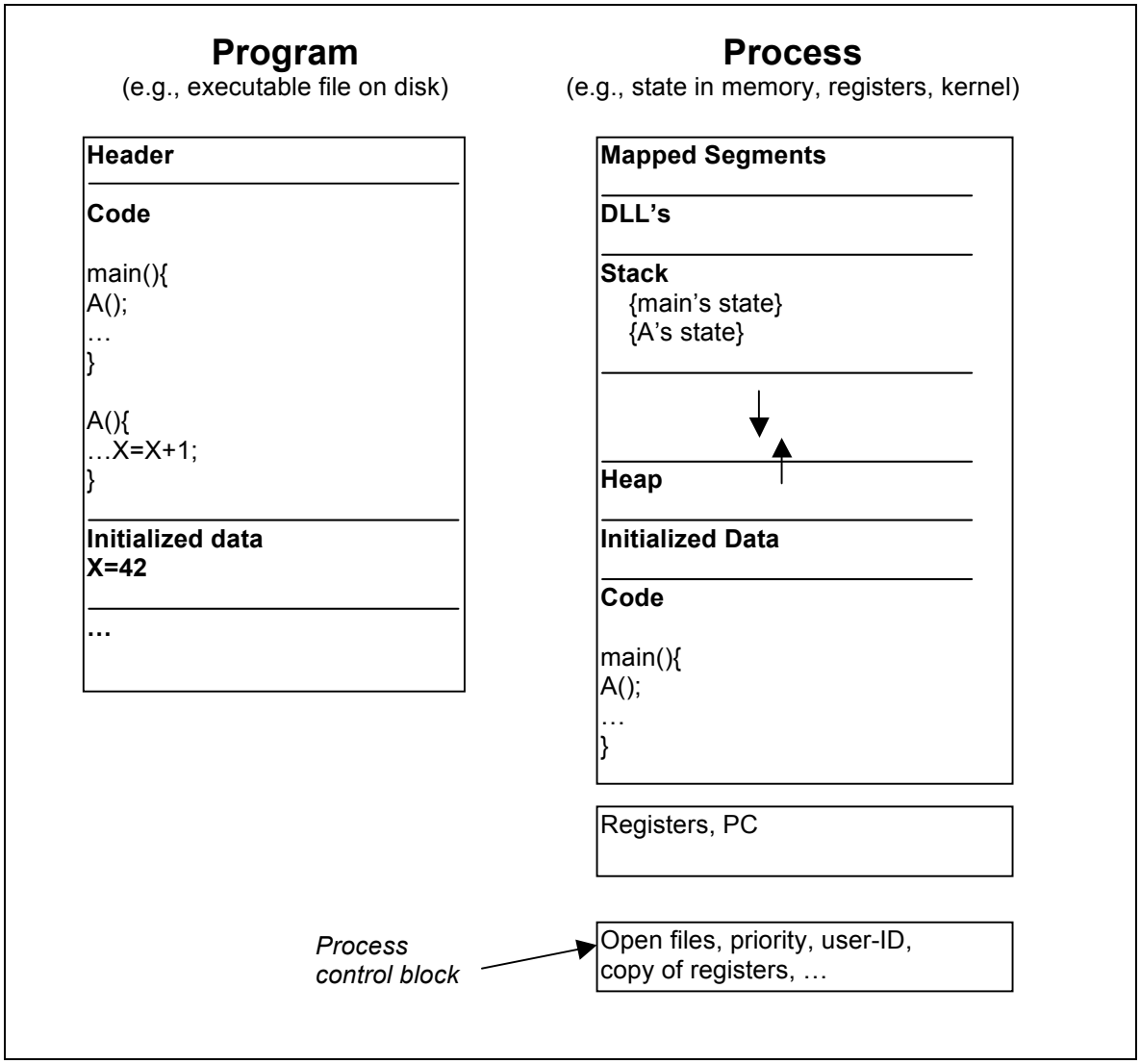
ANSWER: registers, main memory, open files (in Unix), …

Point is – no other state on machine can affect that process; if that process wants to access some other state, it needs to get help from the OS

5.2.2 Process =? Program

program series of commands (e.g. C statements, assembly commands, shell commands)

Anatomy of a process



1) More to a process than just a program

program is *part* of process state

I run *ls*, you run *ls* – same program, different processes

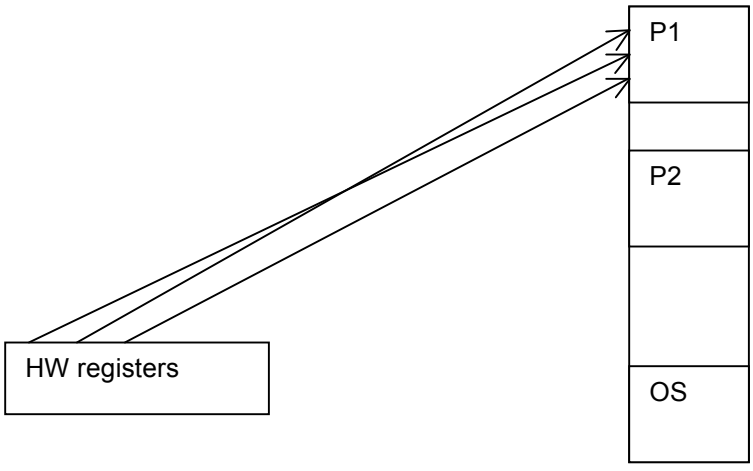
2) Less to a process than a program
A program can invoke more than one process to get the job done
e.g. cc starts up cpp, cc1, cc2, as (each are programs themselves)

5.2.3 OS implementation of process abstraction

2 things
(1) sequential stream of execution
(2) address space

Address space first (quickly now; come back to it in detail in 3 weeks). Then "sequential stream of execution" in detail -- we'll spend the next 3 weeks understanding it, generalizing it to multi-threaded processes.

~~Given process definition, machine can have a bunch of processes in memory at once. [PICTURE]~~



~~When P1's registers are loaded on CPU, registers contain P1's data, pointers, stack pointer, PC, etc. → P1 is running~~

Recall defn of process: sequential execution + address space
--> 2 challenges/mechanisms for above picture
(1) protect P1 from P2 when P2 is running -- **virtual addressing --**
address translation box [add to picture] [TLB]

(2) switch from running P1 to running P2

-- **context switch** -- mechanism to change HW registers
-- **process control block** -- place to save P1s state when P1 is not running (in memory data structure v. registers when running)

arm waving -- but we know how to do this.
Think about how we handled interrupt and transparently returned to interrupted process.
What if we restore some *other* process's registers instead?
voila!

~~Next few weeks, we'll talk about concurrency. What will we need to implement concurrency abstraction/context switch abstraction?~~
~~Analogy with procedure calls, which give you modularity so that you can think about local state and control w/o worrying too much about other procedures. Procedure calls had~~
~~(1) storage space for "inactive" variables -- *stack*~~
~~(2) standard procedures to move "inactive" variables from storage space to registers and vice versa -- *call/return*~~

~~OS has analogous things to let it switch between processes:~~
~~(2) storage space for "inactive" variables -- *process control block*~~
~~(3) standard procedures to move "inactive" variables from storage space to registers and vice versa -- *context switch*~~
~~(why can't we just store "old" register values on stack?)~~

~~Forward pointer to details~~

- ~~Lectures 7-13 will discuss concurrency abstraction and explain details of process control block and context switch~~
- ~~Lectures 3-6 will discuss virtual memory (above picture of memory a bit simplified)~~

5.3 Unix fork + exec

- Fork() creates a child process that is a (nearly) exact copy of parent
  - Identical copy of all parent's in-memory variables
  - Identical copy of all parent's kernel state (open files, etc)
  - Identical copy of all parents registers (except one)
    - Same state → same PC, same program, etc.
    - Child "wakes up" thinking it just called fork()!
    - Parent returns from calling fork()
    - For parent fork() returns child PID
    - For child, fork() returns 0
- Exec() load a new program over current address space
  - Overwrite current memory, registers with specified program
  - Begin execution at _start()

Typical code of a program

```
__start(args){
   ret = main(args);
   exit(ret);
}
```

Common usage:

```
Pid  = fork();
If(pid == 0){
  // I am child…
  // clean up – close files if needed
  exec("/bin/ls", arg0, arg1, …);
}
else{
  I am parent…wait for child or do something else…
}
```

~~5.3.1 Uniprogramming v Multiprogramming (definitions)~~

~~**Uniprogramming** – one process at a time (e.g. MS/Dos)~~

~~Easier for OS builder – get rid of problem of concurrency by defining it away. For PCs, idea was: one user does only one thing at a time~~
~~Harder for user – can't work while waiting for printer~~

~~**Multiprogramming** – more than one process at a time (e..g Unix, NT)~~
~~(often called multitasking, but multitasking sometimes has other meanings – see below – so we won't use that word in this course)~~

****

Summary - 1 min

****

OS Structure: What is an OS? How do you run it?

- Key idea: Supervisor mode v. user mode
- Case studies: internal OS structure
- {By end of day – understand 3 ways to invoke OS}

Transferring between user and kernel mode

OS Structure

monolithic

microkernel

virtual machine

Putting it all together: Boot

Abstraction: Process

- {By end of day – understand anatomy of a process }
- {By end of day – understand differences among process, program, address space, thread}

Processes have two parts – threads and address spaces

Book talks about processes – when this concerns concurrency, really talking about thread portion of a process; when this concerns protection, really talking about address space portion of a process.