Lecture #2 OS Structure, Process abstraction

Anatomy of an OS
Anatomy of a process
****

Review -- 1 min

****

2 basic functions
- Coordination (security, communication, resource mgmt)
- Abstraction/Standard Services
  - Physical Reality → More convenient interface

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,…

- Key idea: Supervisor mode v. user mode
  3 ways to invoke OS

****

Outline - 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}~~

(1) OS structure: monolithic, microkernel, virtual machine
(2) 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

(3) Putting it all together: Booting the OS

****

Preview - 1 min

****

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

****

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

Recall from last time:

OS must control actions of applications
- protection, resource mgmt

→ dual mode operation
"user mode" -- restricted (applications)

"supervisor mode" -- do anything you want (OS kernel)

PICTURE

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)

Basic procedure:

(1) Switch to supervisor mode

QUESTION: How would you implement dual mode (hint: hardware)?
→ e.g., "supervisor" bit
QUESTION: How would hardware detect that an exception has occurred (e.g., user did divide-by-zero)? Interrupt? Trap?

- HW detects need to go to OS (for one of these reasons)
  → jump to well know OS address (handler) and switch on supervisor bit
- HW/OS save user process's state
  What state? Registers (PC, stack pointer, other registers).
- 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
- 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: Using this mechanism, how does a process communicate with another process?

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

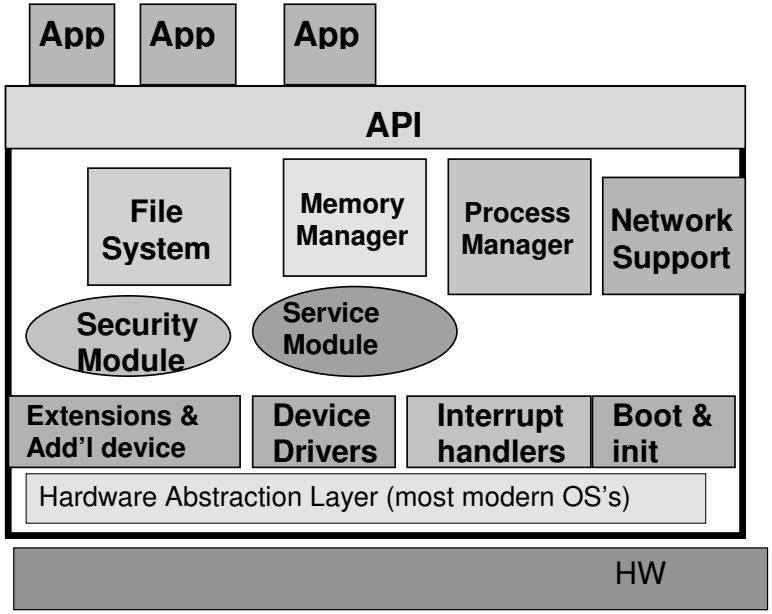
3. Internal OS Structure

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

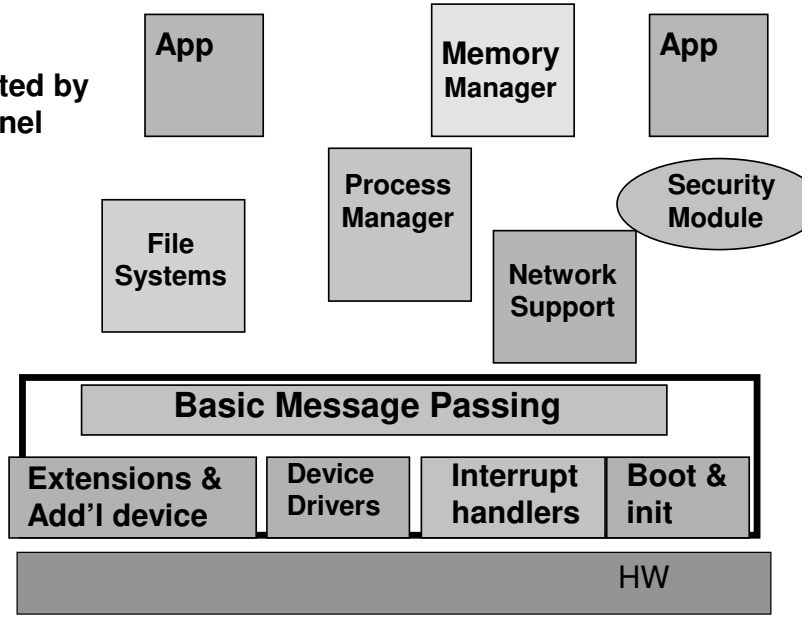
Question: Suppose OS is written in C or C++, then handler needs a stack.
- Where should stack be (is it OK to use application stack?) What has to happen on trap?
- What should handler stack look like on call? (handler call is first stack frame, empty below that)
- What happens to stack if reti happens 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

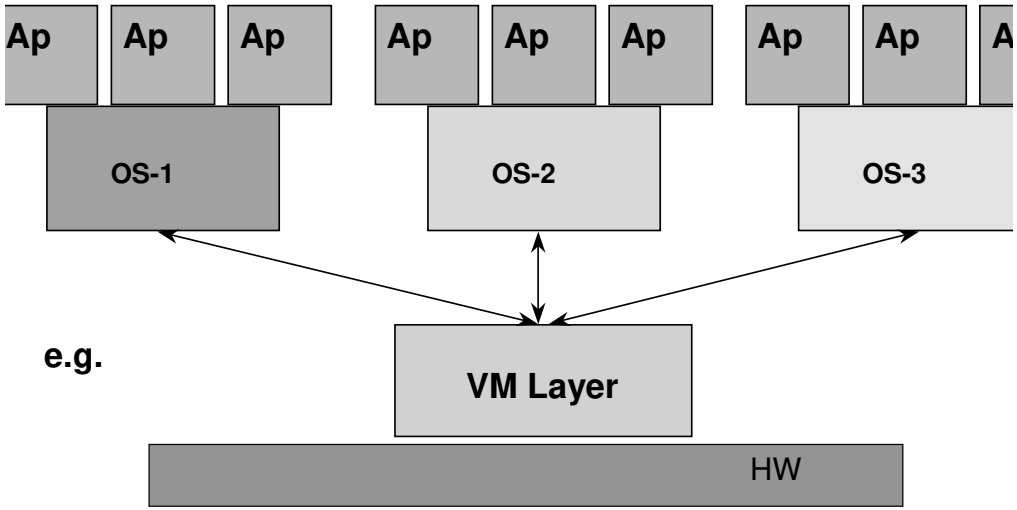
QUESTION: What are advantages of microkernel? Of monolithic?

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)

e.g.



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
Project 1 due Wednesday Feb 4 before class
Chris's office hours:
Th 12:30-2:00, Fri 3:00-4:00 PAI 5.36C
This Friday – "discussion section" on some aspects of lab

4. Process abstraction

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

4.1 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)

4.2 Processes

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

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

4.2.1 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

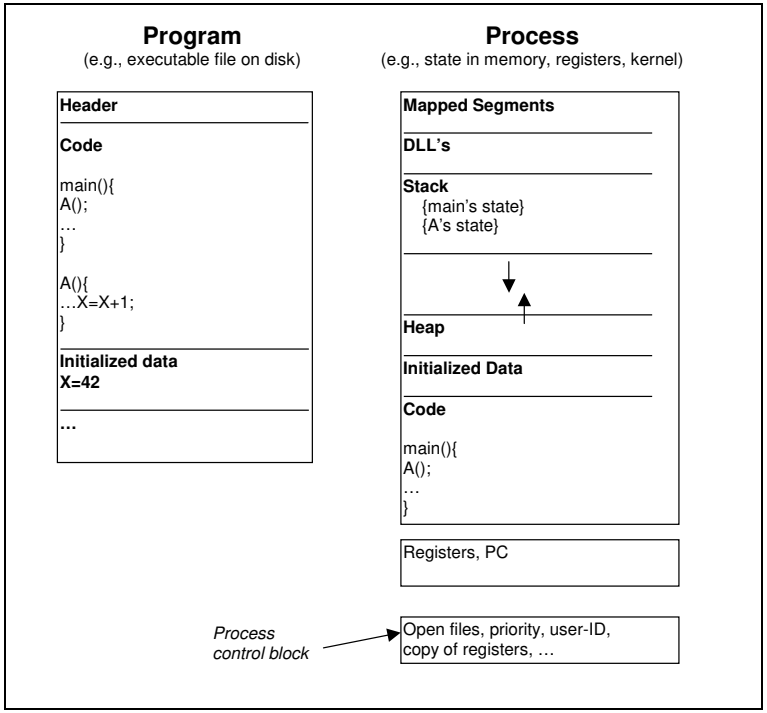
QUESTION: I'm running emacs on my Sun. What state is part of that process?
ANSWER: registers, main memory, open files (in Unix), …

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

4.2.2 Process =? Program

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

Anatomy of a process



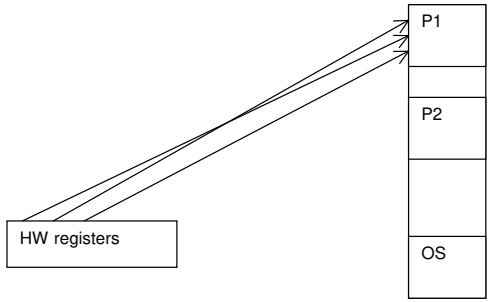
1) More to a process than just a program
program is *part* of process state

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)

4.2.3 OS implementation of process abstraction

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

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:
(1) storage space for "inactive" variables – *process control block*
(2) 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)

4.2.4 Linking and loading

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

Solution: **loading**
(1) In header portion of file, list offsets of symbols (e.g., "A" = start+178), list where symbols referenced (e.g., "A used at start+47")
(2) Loader program given offset, reads header, replaces all references with correct final value

Generalization: linking
What if we have 4 ".o" files to link together?
- each file has list of exported symbols and locations and a list of imported sysbols and locations they are used

Generally 2 phases: linking + loading

Variations:
- Unix has combined linker/loader
- Modern OS's – dynamic linking and loading
- Alternative: position independent code (alternative to linking or loading?)

4.2.5 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)

5. 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)
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 sector of hard disk)
     - To "well-known" memory location
   - Jump to bootloader code (well-known entry point)
3) Bootloader
   - Initialize stuff
     - SP = Initial stack at well-known location
     - 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
   - Read OS from disk, jump to well-known entry point
4) OS initialization
   - Initialize kernel data structures
   - Initialize state of HW devices
   - Initialize virtual memory system
   - Creates a number of processes to start operation (e.g. daemons, tty in Unix/windowing system in NT)
   - How does this work?

Boot creates first proc ("sysproc" in unix)
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…)

5) 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

5.1 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…
}
```

****

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}

Abstraction: Process

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

Transferring between user and kernel mode

OS Structure

- monolithic
- microkernel
- virtual machine

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.