Lecture 2:

Intro to CPU Mechanisms

Virtualization: The CPU [Fall '11] ①

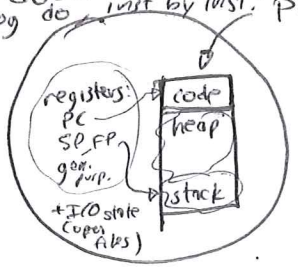
goal: run N processes @ once, even though only M CPUs (N >> M) [multiprogramming]
⇒ each thinks it has its own isolated machine [illusion]

CPU: A | B | C | A | B | C → time sharing

Q why? [ease of use, etc.]
⇒ also, it's own restricted machine
⇒ it's own efficient machine

abstraction: the Process (running program)

Q what does a running prog do inst by inst? private memory: address space (kind of ignore for now)



virtualizing CPU ⇒ virtualizing registers of machine

what is OS to a process? ⇒ set of APIs (std library)

to support abstraction: need
focus: mechanisms: how to virtualize
policies: which Proc to run? (next time)
general mech: limited direct execution

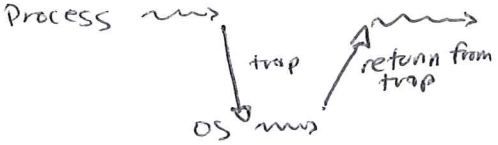
How to virtualize CPU:
start simple: when you want to run a program, just run it!
(load into memory, point PC @ first inst, go!)
⇒ direct execution

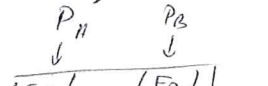
SHOW PROTOCOL HERE (CLI)

Three Problems:
1) what if process wants to do something restricted?
2) what if OS wants to stop P_A, run P_B?
3) what if running P does something slow, like I/O?

Problem #1: restricted ops (e.g. I/O) ⇒
solution: privilege levels + way to change it (e.g. the system call)
1) user process: "user" mode (restricted) 2) OS: "kernel" mode (not restricted)

need way to change between them: sys call





to read: io instruction (read block 10)
could trust processes not to read other user's files...
⇒ save state of running process
trap: changes privilege ⇒ level to kernel mode
⇒ runs "trap handler" ⇒ as code for handling this trap
rett: ⇒ restore state, run back ... privilege

Prob #1: cont [Fall '11, (2)]

inst: trap

1) save state: done by h/w (+ s/w) sometimes
   logically: save PC, registers to some special place
   then, can run OS code w/o worry of corrupting user code registers
2) raise priv. level : allows fun stuff
3) jump to correct kernel code : how does h/w know which PC to jump to?

(super powerful instruction!)

Q some save state must be done by h/w: why?

Q special instruction: privileged?

protocol:

kernel boots (kernel mode)
installs trap tables ⇒ list of addresses of functions to run when certain traps/interrupts happen
eventually runs a process (user mode)
syscalls: → kernel, ← user

on trap: h/w consults trap table to know which code to run

OS issue: where to put saved state? ⇒

instruction: return-from-trap

1) lower priv. level
2) restore state of registers of calling process
3) jump back to PC after trap in user code

Q what if user-mode process tries to do something restricted?
⇒ killed

Problem #2: how to stop currently running P_A, run P_B?
(only 1 CPU: OS is not running when P_A is running)

approach #1: cooperative — just wait for process to call into kernel or yield explicitly

#2: non-cooperative — forcibly reclaim CPU via timer interrupt

issue: must

[turn on @ boot, before running any user process]
goes off every X ms: decide to continue P_A, or switch to P_B?

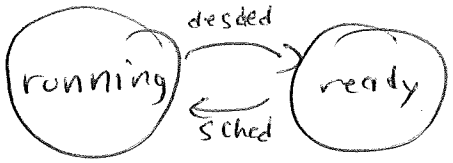
if switch: ⇒ ~~add~~ context switch
save P_A state, stop it
:
restore P_B's state, run it

Prob #2 : cont [Fall '11, ③]

OS issues:

1/trust

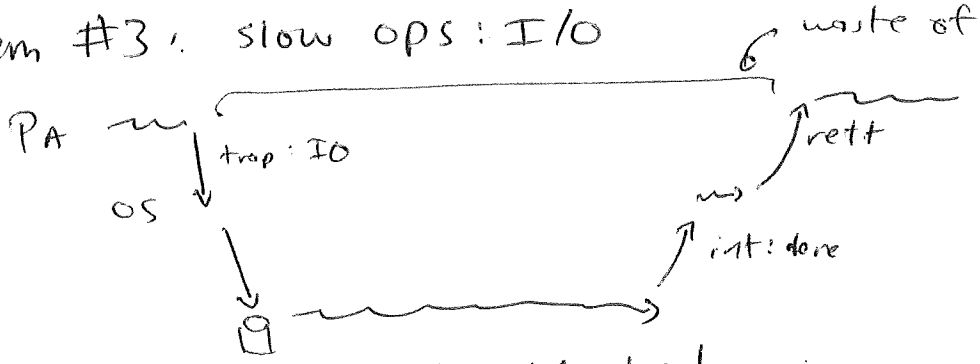
2) have to track state of processes



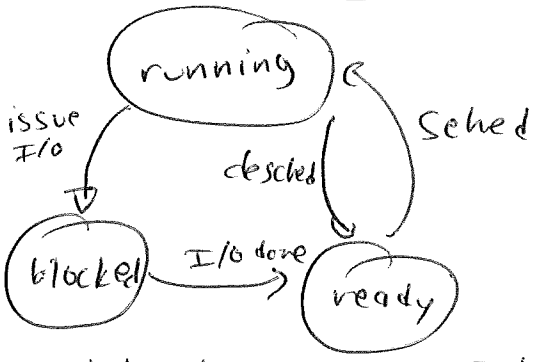
3) need data struct to track this: process list

per-process info: saved registers, state: ready, running

Problem #3: slow ops: I/O



new process state: blocked



when trap into kernel for I/O, OS switches to another process ⇒ efficiency

Summary: Limited Direct Execution

direct: because usually, just runs

limited: at certain key points, OS + h/w get in way to retain control of system

result: ~~SMPL~~ virtualized CPU

⇒ w/ safety

⇒ w/ efficiency