COSC 301: Weds-Fri, 12-14 Sept 2012

Learning goals

- Identify the attributes and purposes of a process.
- Describe the actions that take place on a context switch.

Virtualizing the CPU

Goal: run N processes "at once" even through only M CPUs (N » M)

- This is termed "multiprogramming"
  - Multiple programs residing in memory at once, possibly running simultaneously
- Each process thinks it has its own isolated machine
- Also, each thinks it has its own *restricted* and *efficient* machine
  - That is, the cost of creating the illusion shouldn't be high
  - And we want to be sure to protect/isolate processes from each other
- **Why**? Why do we want to do any of this?

Key abstraction: the process (a running program)

- What does a running program do?
  - Has a private memory ("address space")
  - Simple operations on memory and registers
    - Virtualizing the CPU will require us to (in a sense) virtualize the registers!
- To support the process abstraction, need

  Mechanisms *How* to virtualize Name of general mechanism: "limited direct execution"
  Policies Which process to run and how long? (scheduling, next week)

How to virtualize CPU?

- Start simple: when you want to run a program, just run it
  - Load it into memory, point PC at first instruction, go!
  - This is *direct execution*
- What's wrong? What are the problems that can arise?
  1. What if process wants to do something that is restricted?
  2. What if OS wants to stop process A and run process B?
  3. What if the running process does something *slow*, like I/O?

Problem 1: Restricted operations (like I/O, device operations, etc.)

We could trust processes not to read the files of other processes, but that's probably just a little bit naive.

- Solution: operating system privilege levels
  - Process can be running in one of two modes:
    1. "User" mode (restricted)
    2. "Kernel" mode (not restricted)
  - Need a way to change between these two modes: the **system call**
  - Scenario: process runs, calls syscall (e.g., `fopen`), causes trap (implicit change in privilege), does restricted op, returns.
- On `trap` instruction (`INT` on x86)
  1. Save state: done by hardware (sometimes also software)
     - logically, save PC, registers to a special place, then run OS code w/o worry of corrupting user registers
     - why must the saving of state be done (at least partially) by hardware?
     - OS issue: where to put the saved state? (don't answer Q yet...)
  2. Raise privilege level - allows us to do some fun stuff
  3. Jump to correct kernel code (trap handler, or interrupt handler)
     - How does hardware know which PC to jump to?
     - Protocol:
       * Kernel boots (into kernel mode)
       * Installs trap tables (list of addresses of functions to run when certain traps/interrupts happen
       * OS eventually runs a process (in user mode)
     - Process makes sys calls
     - So on trap, the hardware consults the trap table (interrupt vector) to know which code to run
- On return from trap
  1. Lower privilege level
  2. Restore state of registers of calling process
  3. Jump back to PC after trap in user code

  What if a process running in user mode tries to do something restricted? E.g., call `halt` instruction?
  - Typically the OS will kill it

Problem 2: How to stop a currently running process A and run process B? (Assume only 1 CPU: when A is running, B cannot be running)

Cooperative just wait for process to call into kernel or yield explicitly

Non-cooperative

forcibly reclaim the CPU via a **timer interrupt**

- turn on timer interrupt at boot, before *any* user process
- goes off every x milliseconds
- at timer interrupt, decide to continue A or switch to B

What's the main issue for deciding whether we should use *cooperative* or *non-cooperative* scheduling? (trust!)

If we decide to switch the running process: **context switch**

1. Save A state, stop it

2. Restore B's state, run it

OS issues

trust processes may misbehave or may be buggy

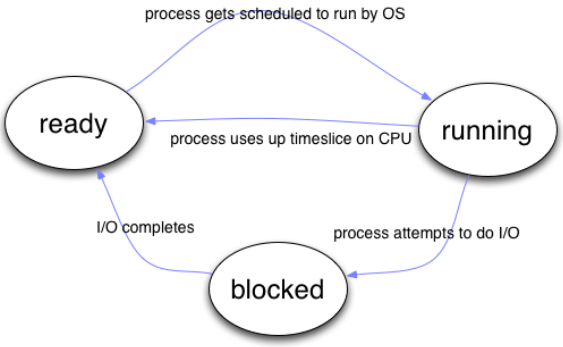
bookkeeping need to keep track of state of each process (whether running, or ready to be run) need a data structure to track this info: *process list*

- Per-process info stored: saved registers (answers question from above on where we save process state on a syscall), state: ready/running

---

Problem 3: Slow operations like I/O

- Basic solution: keep more detailed information about process state



Summary: Limited Direct Execution

direct Want process to run on the CPU directly as much as possible

limited At certain key points, OS and hardware impose themselves to regain/retain control of the system

Result: a virtualized CPU

- With safety (why?)
- With efficiency (why?)