Processes and Address Spaces

COMS W4118

Prof. Kaustubh R. Joshi

krj@cs.columbia.edu

http://www.cs.columbia.edu/~krj/os

References: Operating Systems Concepts (9e), Linux Kernel Development, previous W4118s

Outline

- Processes
- Address spaces
- Mechanisms
- Process lifecycle

Multiprogramming

- OS requirements for multiprogramming
  - Scheduling: what to run? (later)
  - Dispatching: how to switch? (today)
  - Memory protection: how to protect from one another? (today + later)

- Separation of policy and mechanism
  - Recurring theme in OS
  - Policy: decision making with some performance metric and workload (scheduling)
  - Mechanism: low-level code to implement decisions (dispatching, protection)

What is a process

- Process: an execution stream in the context of a particular process state
  - “Program in execution” “virtual CPU”
- Execution stream: a stream of instructions
- Process state: determines effect of running code
  - Registers: general purpose, instruction pointer (program counter), floating point, …
  - Memory: everything a process can address, code, data, stack, heap, …
  - I/O status: file descriptor table, …

Program v.s. process

- Program != process
  - Program: static code + static data
  - Process: dynamic instantiation of code + data + more

- Program ⇔ process: no 1:1 mapping
  - Process > program: more than code and data
  - Program > process: one program runs many processes
  - Process > program: one process can run multiple programs (exec)

Why use processes?

- Express concurrency
  - Systems have many concurrent jobs going on
    - E.g. Multiple users running multiple shells, I/O, …
  - OS must manage

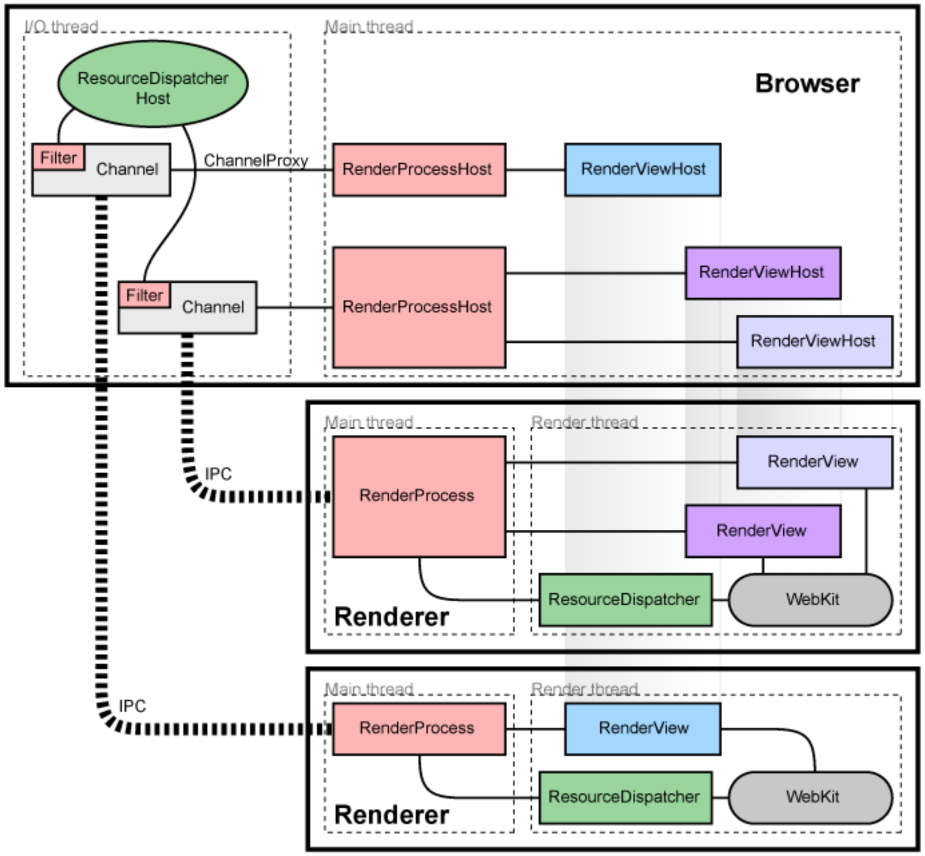
- General principle of divide and conquer
  - Decompose a large problem into smaller ones → easier to think of well contained smaller problems

- Isolated from each other
  - Sequential with well defined interactions

Example: The Chrome Browser

- Multiple processes, one for each plugin, webpage
- If one webpage unresponsive, doesn't crash browser

New! Download Chrome (BETA) - the new browser from Google



Source: http://www.chromium.org/developers/design-documents/multi-process-architecture

Outline

- Processes

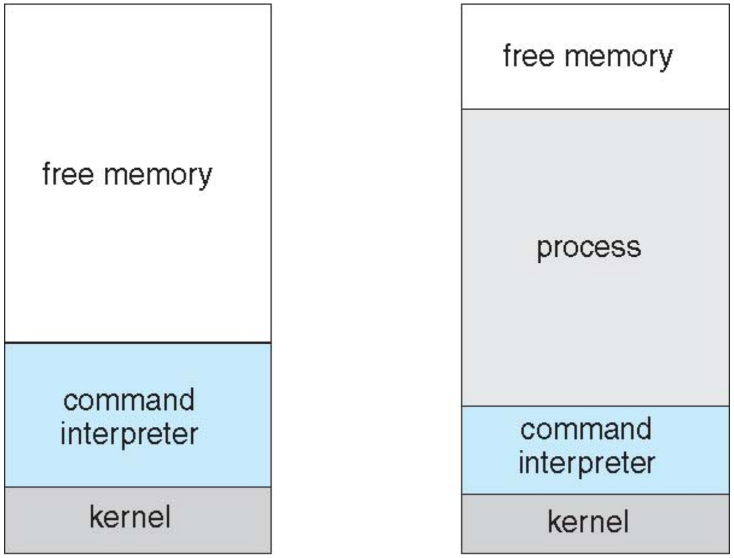
- Address spaces

- Mechanisms

- Process lifecycle

How to assign memory to processes?

- Uniprogramming: one process at a time
  - Eg., early main frame systems, MSDOS
  - Good: simple
  - Bad: poor resource utilization, inconvenient for users
  - Application can overwrite OS



(a) (b)

Supporting Multiprogramming

- Want Multiprogramming: multiple processes, when one waits, switch to another
  - Can’t have one process overwriting other’s memory. What to do?

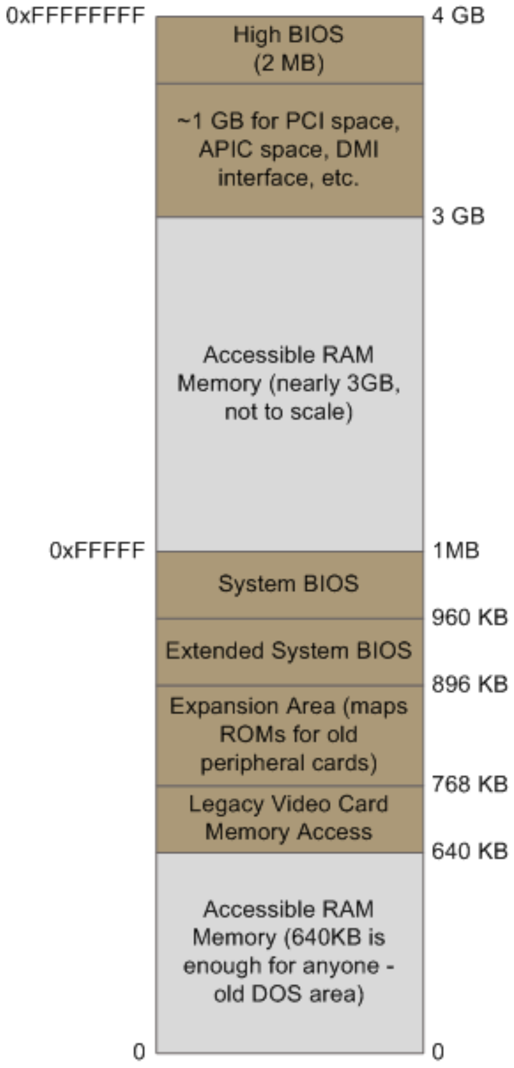
Supporting Multiprogramming (2)

- Solution 1: enforce in programming language
  - The “Java” approach
  - No pointers, runtime array bounds checks
  - Compiler can statically optimize many checks
  - Forces programmers to write in high level language
- Solution 2: runtime checks on every memory access
  - Solves security problem. Expensive, but hardware support can help.
  - Memory addresses change every time program is loaded
  - Can’t move program once its loaded (to compact space)
- Solution 3: add a level of indirection!
  - Each memory address is really a pointer
  - Table maps “virtual” memory address to “physical” or real address
  - Hardware usually provides support to speed up (more later)

Address Space

- Address Space (AS): all memory a process can address
    - Really large memory to use
    - Linear array of bytes: [0, N), N roughly 2^32, 2^64
- Process ⇔ address space: 1:1 mapping
- Address space = protection domain
    - OS isolates address spaces
    - One process can't even see another's address space
    - Same pointer address in different processes point to different memory
    - Can change mapping dynamically

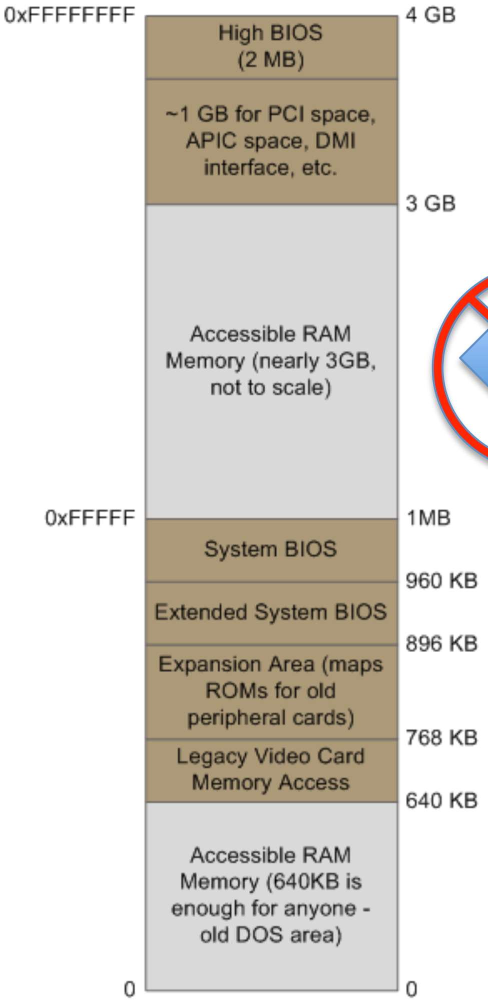
x86 PC Physical Memory Layout

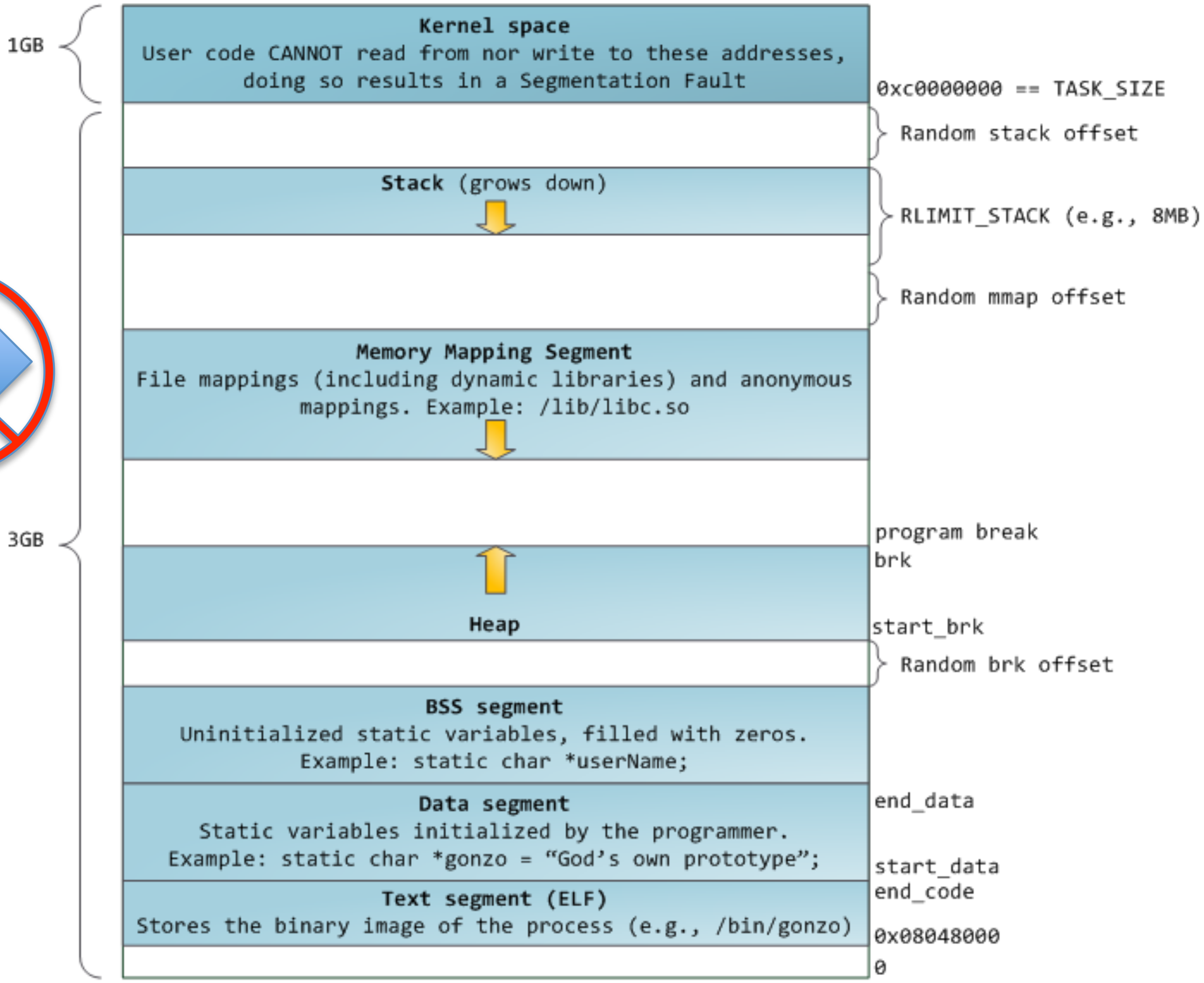


- Specific to each platform
- Different across architectures
- Different for machines with the same processor
- Firmware knows exact layout
- Passes to kernel at boot time (in Linux through atag_mem structures)

Linux Address Space Layout

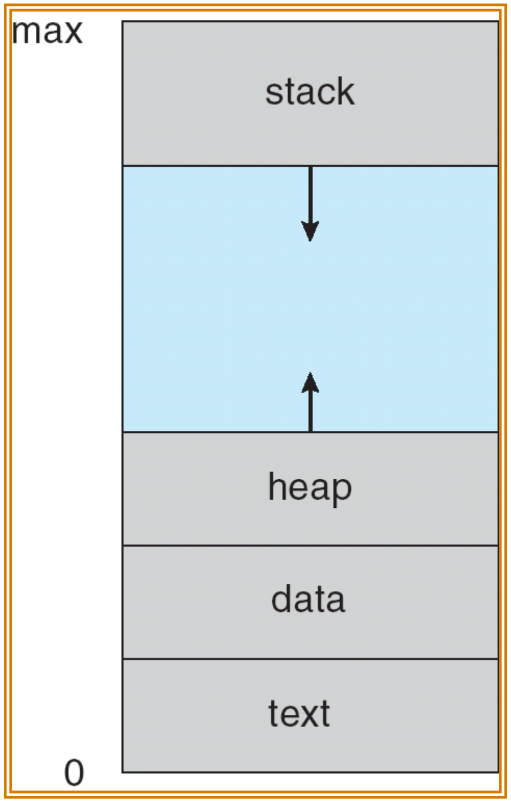
- Same address layout for all processes



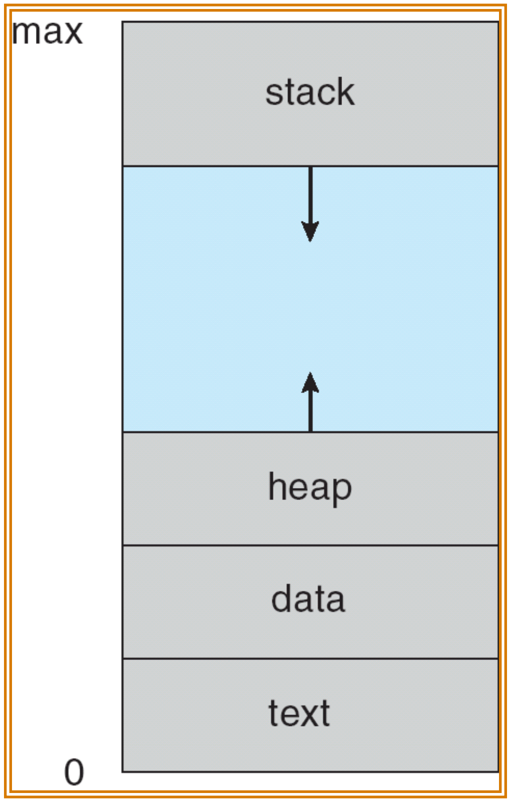


Read: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory

Address space illustration



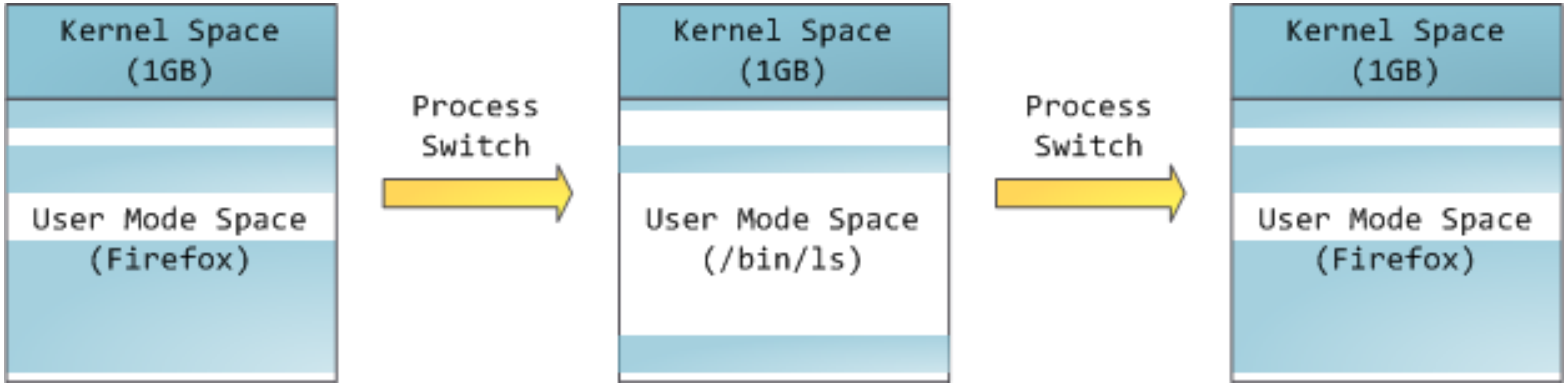
Process A



Process B

Process Switching

- Process switch? Just change memory map!
  - Therefore, also called context switch
  - All CPUs with memory management unit (MMU)
  - Special register points to active map
  - On x86, cr3 register (is this privileged?)



Source: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory

Linux Address Space Demo

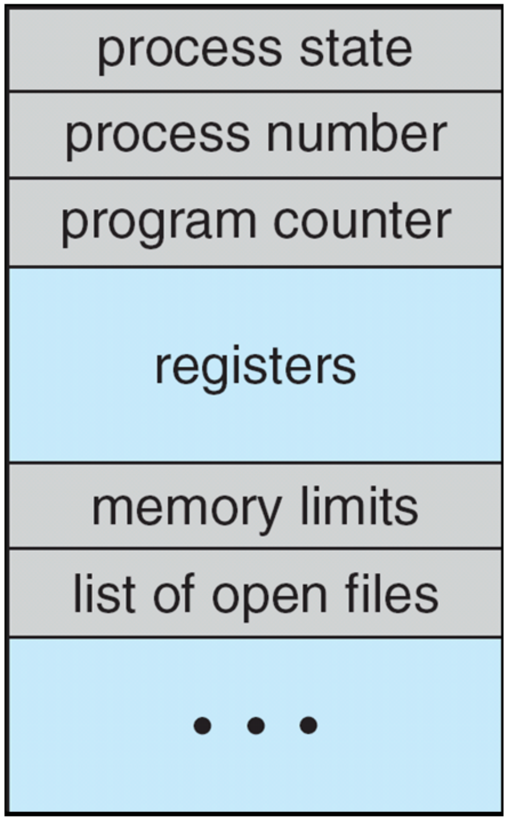
- ssh –Y clic-lab.cs.columbia.edu
- The /proc and /sys filesystems
  - Another abstraction: live data structure as a file
- /sys/firmware/memmap
  - Raw physical memory regions reported by BIOS
- /proc/iomem (/proc/ioports while at it)
  - Additional information filled in by OS drivers
- /proc/<own_process_pid>/
  - cmdline: name of program
  - maps: address space

Outline

- Processes
- Address spaces
- Mechanisms
- Process lifecycle

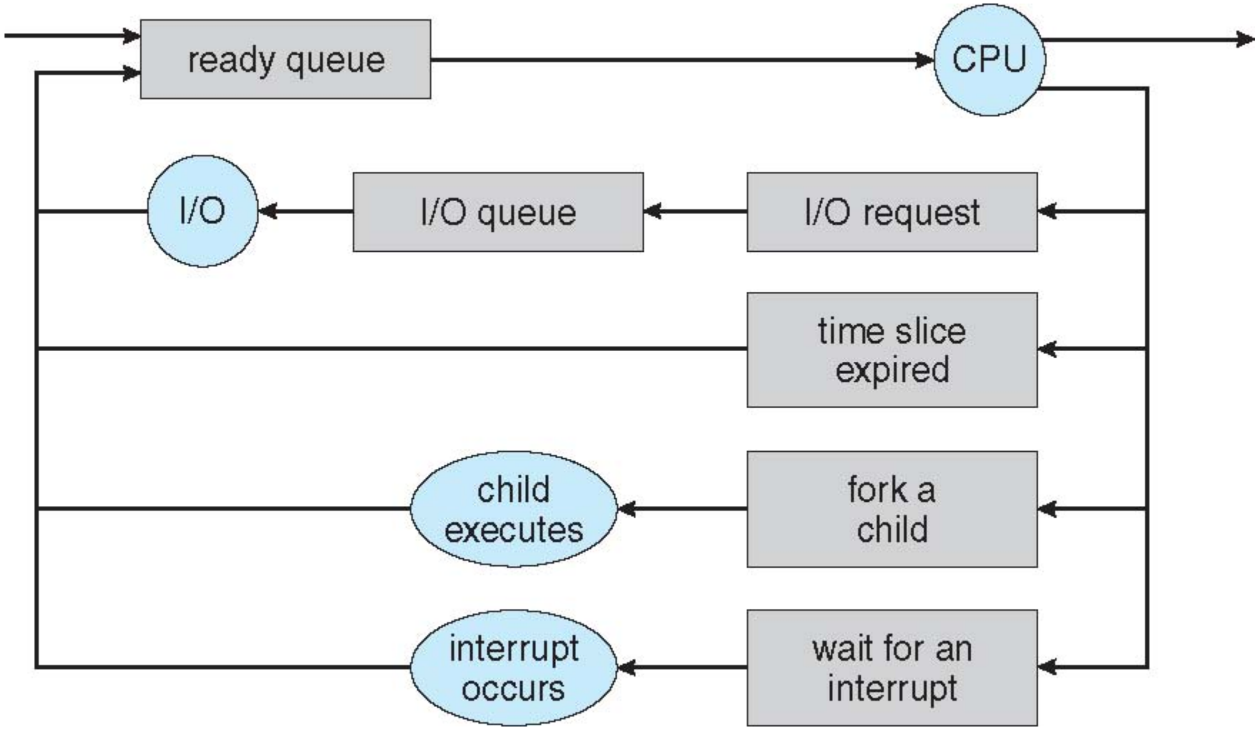
Process management

- Process control block (PCB)
  - Process state (new, ready, running, waiting, finish …)
  - CPU registers (e.g., %eip, %eax)
  - Scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information
- OS often puts PCBs on various queues
  - Queue of all processes
  - Ready queue
  - Wait queue



process state
process number
program counter
registers
memory limits
list of open files
• • •

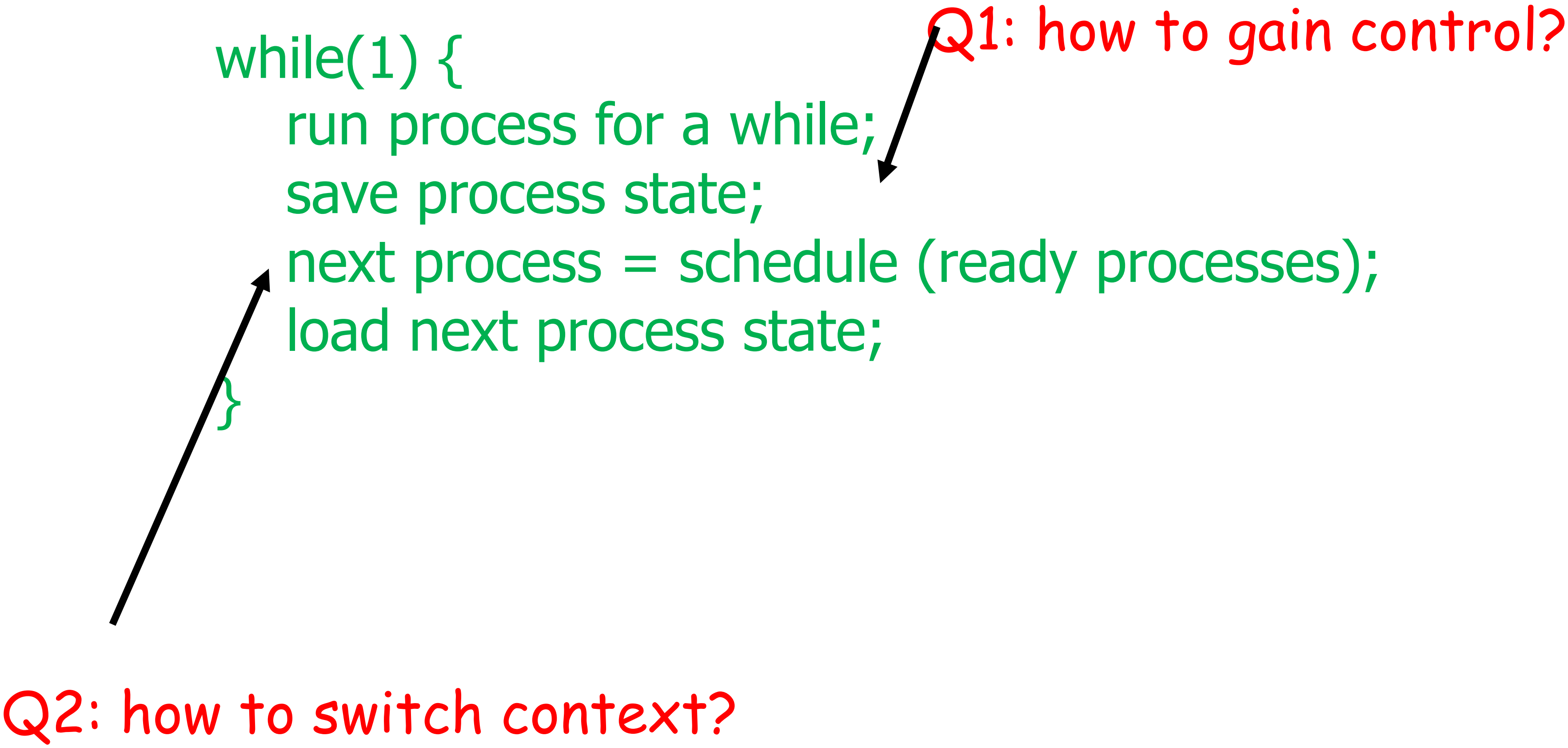
Process Scheduling Queues



- Process can be in one of many states: new, ready, waiting, running, terminated
- Scheduler only looks at ready queue (policy: later)
- I/O interrupts move processes from waiting to ready queues

Process dispatching mechanism

OS dispatching loop:



```
while(1) {
    run process for a while;
    save process state;
    next process = schedule (ready processes);
    load next process state;
}
```

Q1: how to gain control?

Q2: how to switch context?

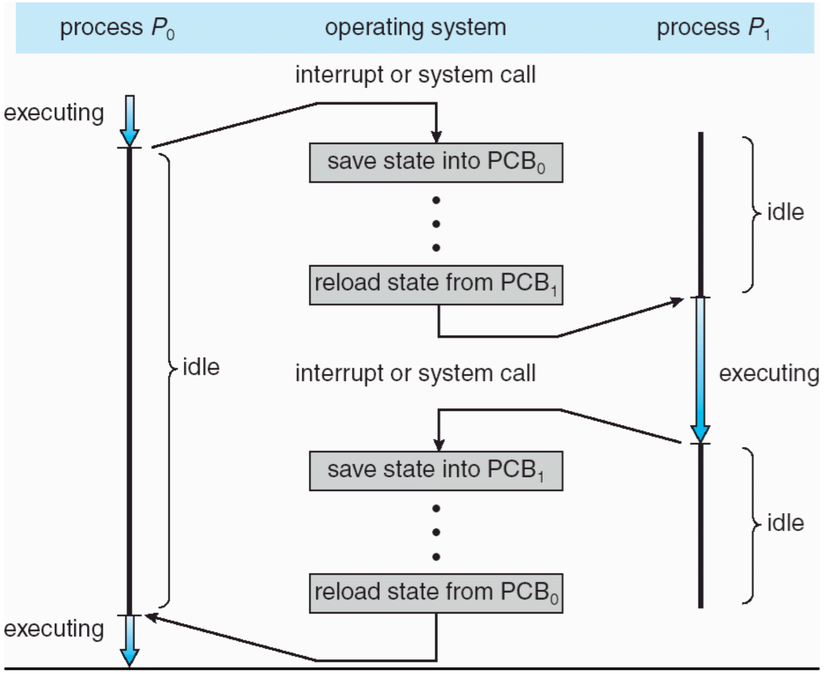
Q1: How does Dispatcher gain control?

- Must switch from user mode to kernel mode
- Cooperative multitasking: processes voluntarily yield control back to OS
  - When: system calls that relinquish CPU
  - OS trusts user processes!
- True multitasking: OS preempts processes by periodic alarms
  - Processes are assigned time slices
  - Counts timer interrupts before context switch
  - OS trusts no one!

Q2: how to switch context?

- Implementation: machine dependent
  - Tricky: OS must save state w/o changing state!
    - Need to save all registers to PCB in memory
    - Run code to save registers? Code changes registers
  - Solution: software + hardware

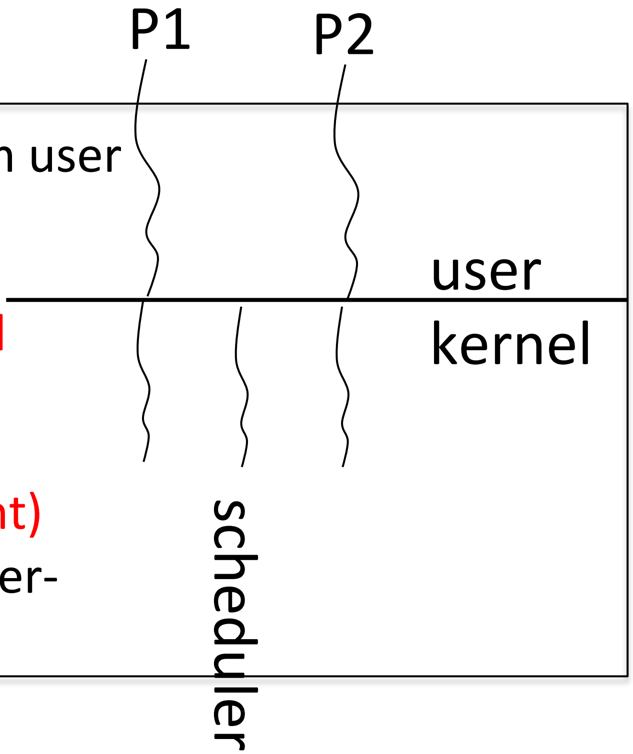
- Performance?
  - – Can take long. Save and restore many things. The time needed is hardware dependent
  - – Context switch time is pure overhead: the system does no useful work while switching
  - – Must balance context switch frequency with scheduling requirement



Example: Linux Context Switch

Contains both arch dependent and independent pieces

- Arch independent code in kernel/sched.c, context_switch()
- Arch dependent in include/asm/system.h and arch/x86/kernel/process_32.c in switch_to macro



1. Save P1’s user-mode CPU context and switch from user to kernel mode (need hw)
2. Scheduler selects another process P2
3. Switch to P2’s address space (need hw, but kernel memory stays same)
4. Save P1’s kernel CPU context (arch dependent)
5. Switch to P2’s kernel CPU context (arch dependent)
6. Switch from kernel to user mode and load P2’s user-mode CPU context (need hw)

- Change context by changing kernel stack
- When stack changes, all local variables change, including the identity of the previous and next PCB!
- Solution: maintain across process switch by storing in registers

Reference: Bovet and Cesati, Ch. 3.3

Outline

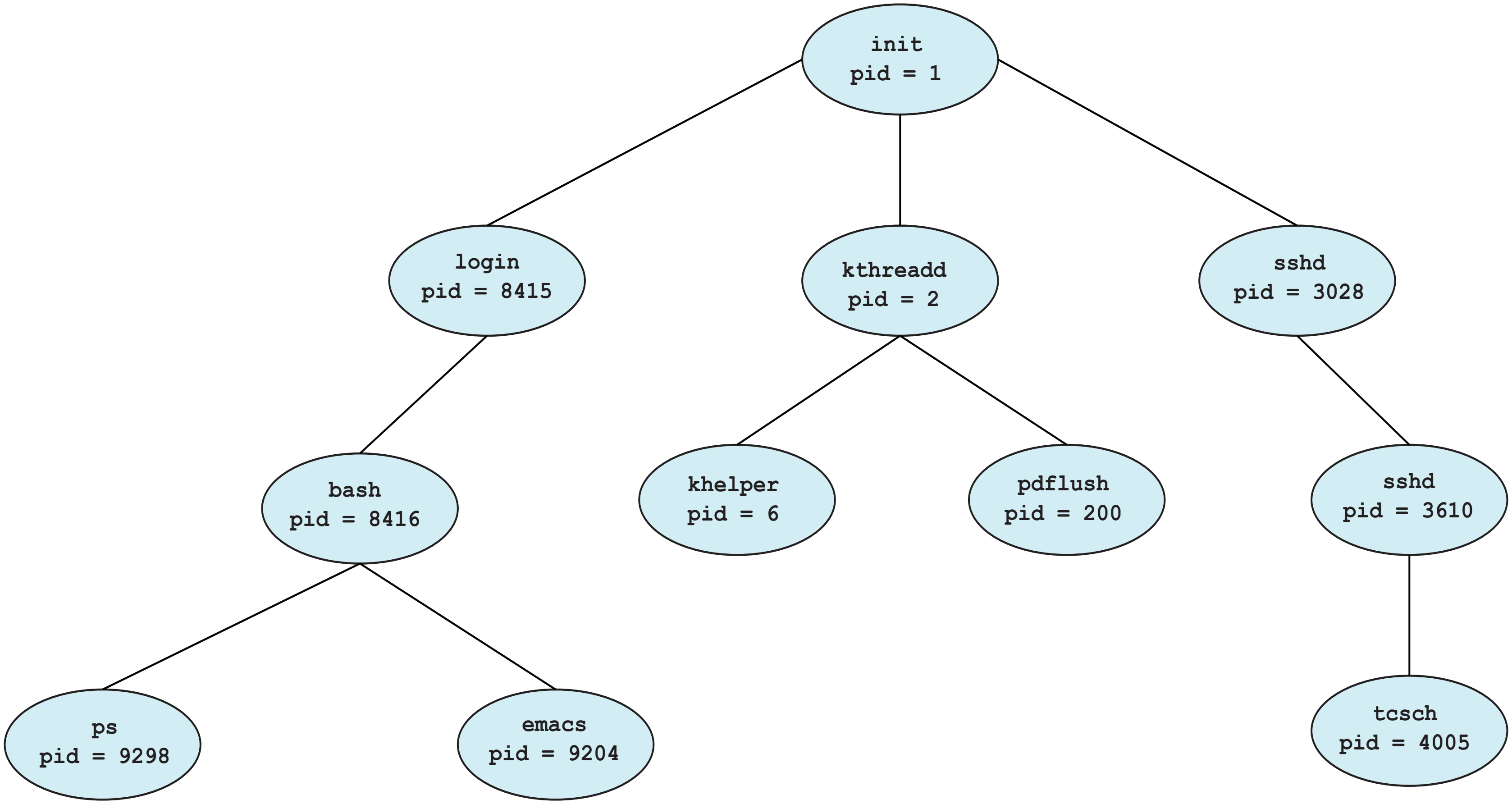
- Processes
- Address spaces
- Mechanisms
- Process lifecycle

Process creation

- Option 1: cloning (e.g., Unix fork(), exec())
  - Pause current process and save its state
  - Copy its PCB (can select what to copy)
  - Add new PCB to ready queue
  - Must distinguish parent and child

- Option 2: from scratch (Win32 CreateProcess)
  - Load code and data into memory
  - Create and initialize PCB (make it like saved from context switch)
  - Add new PCB to ready queue

A Process Tree



- On Linux: ps axjf to see process tree

Distinguished Processes

- The UNIX init process: /sbin/init
  - First and only user process instantiated by the kernel
  - Kernel forks init and goes idle
  - Responsible for forking all other processes
    - login screen, window manager
  - Can be configured to start different things
    - Read scripts in /etc/init.d on Linux
- The Android zygote process
  - Parent of all managed (Java) applications
  - Preloaded version of Dalvik runtime, libraries
  - fork() makes new application loading very efficient
  - Less memory, faster app start

Process termination

- Normal: exit(int status)
  - OS passes exit status to parent via wait(int *status)
  - OS frees process resources
- Abnormal: kill(pid_t pid, int sig)
  - OS can kill process
  - Process can kill process

Zombies and orphans

- What if child exits before parent?
  - Child becomes zombie
    - Need to store exit status
    - OS can’t fully free
  - Parent must call wait() to reap child

- What if parent exits before child?
  - Child becomes orphan
    - Need some process to query exit status and maintain process tree
  - Re-parent to the first process, the init process