CS162
Operating Systems and Systems Programming
Lecture 3

Concurrency:
Processes, Threads, and Address Spaces

September 7, 2005

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

Review: History of OS

- **Why Study?**
  - To understand how user needs and hardware constraints influenced (and will influence) operating systems
- **Several Distinct Phases:**
  - Hardware Expensive, Humans Cheap
    » Eniac, … Multics
  - Hardware Cheaper, Humans Expensive
    » PCs, Workstations, Rise of GUIs
  - Hardware Really Cheap, Humans Really Expensive
    » Ubiquitous devices, Widespread networking
- **Rapid Change in Hardware Leads to changing OS**
  - Batch ⇒ Multiprogramming ⇒ Timeshare ⇒ Graphical UI ⇒ Ubiquitous Devices ⇒ Cyberspace/Metaverse/??
  - Gradual Migration of Features into Smaller Machines
- **Situation today is much like the late 60s**
  - Small OS: 100K lines/Large: 10M lines (5M browser!)
  - 100-1000 people-years

Review: Migration of OS Concepts and Features
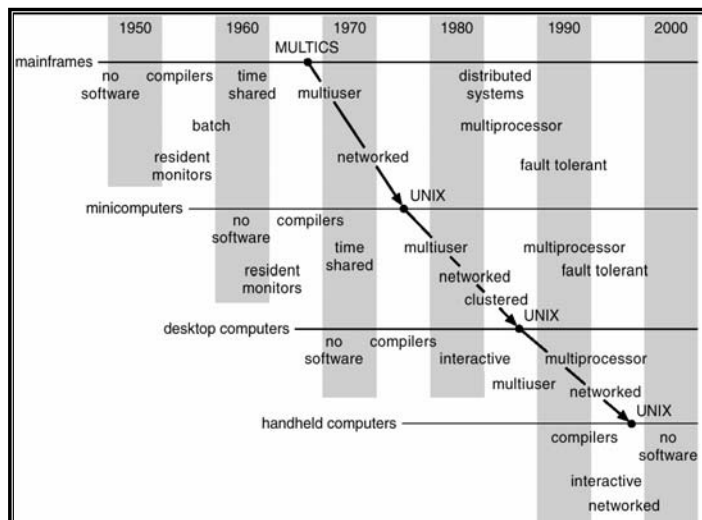
Review: Implementation Issues
(How is the OS implemented?)

- **Policy vs. Mechanism**
  - Policy: **What** do you want to do?
  - Mechanism: **How** are you going to do it?
  - Should be separated, since policies change
- **Algorithms used**
  - Linear, Tree-based, Log Structured, etc…
- **Event models used**
  - threads vs event loops
- **Backward compatability issues**
  - Very important for Windows 2000
  - POSIX tries to help here
- **System generation/configuration**
  - How to make generic OS fit on specific hardware

Goals for Today

- **How do we provide multiprogramming?**
- **What are Processes?**
- **How are they related to Threads and Address Spaces?**

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

Concurrency

- **"Thread" of execution**
  - **Independent Fetch/Decode/Execute loop**
  - **Operating in some Address space**
- **Uniprogramming:** *one thread at a time*
  - **MS/DOS, early Macintosh, Batch processing**
  - **Easier for operating system builder**
  - **Get rid concurrency by defining it away**
  - **Does this make sense for personal computers?**
- **Multiprogramming:** *more than one thread at a time*
  - **MULTIX, UNIX, OS/2, Windows NT/2000/XP**
  - **Often called "multitasking", but multitasking has other meanings (talk about this later)**
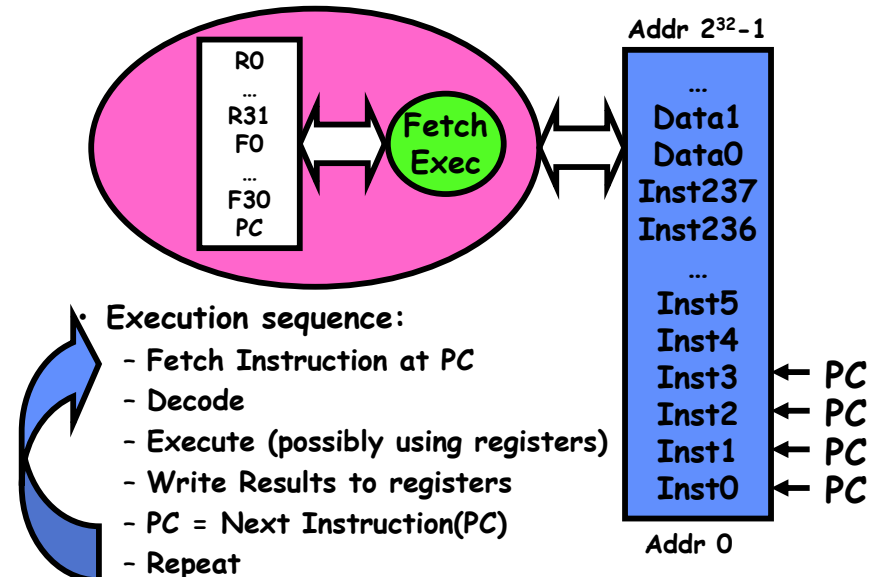
The Basic Problem of Concurrency

- **The basic problem of concurrency involves resources:**
  - **Hardware: single CPU, single DRAM, single I/O devices**
  - **Multiprogramming API: users think they have exclusive access to machine**
- **OS Has to coordinate all activity**
  - **Multiple users, I/O interrupts, …**
  - **How can keep this straight?**
- **Basic Idea: Use Virtual Machine abstraction**
  - **Decompose hard problem into simpler ones**
  - **Abstract the notion of an executing program**
  - **Then, worry about multiplexing these abstract machines**

Recall (61C): What happens during execution?

- **Execution sequence:**
  - **Fetch Instruction at PC**
  - **Decode**
  - **Execute (possibly using registers)**
  - **Write Results to registers**
  - **PC = Next Instruction(PC)**
  - **Repeat**

How can we give the illusion of multiple processors?
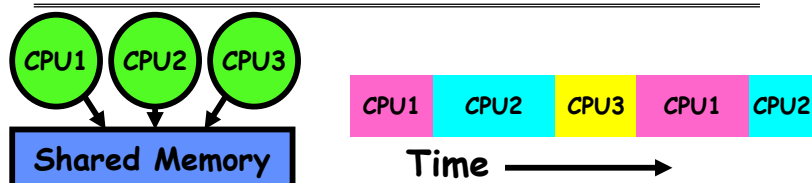
- **How do we provide the illusion of multiple processors?**
  - Multiplex in time!
- **Each virtual "CPU" needs a structure to hold:**
  - Program Counter (PC)
  - Registers (Integer, Floating point, others…?)
- **How switch from one CPU to the next?**
  - Save PC and registers in current state block
  - Load PC and registers from new state block
- **What triggers switch?**
  - Timer, voluntary yield, I/O, other things

Properties of this simple multiprogramming technique

- **All virtual CPUs share same non-CPU resources**
  - I/O devices the same
  - Memory the same
- **Consequence of sharing:**
  - Each thread can access the data of every other thread (good for sharing, bad for protection)
  - Threads can share instructions (good for sharing, bad for protection)
  - Can threads overwrite OS functions?
- **This (unprotected) model common in:**
  - Embedded applications
  - Windows 3.1/Machintosh (switch only with yield)
 - Windows 95/ME? (switch with both yield and timer)

Modern Technique: SMT/Hyperthreading

- **Hardware technique**
  - Exploit natural properties of superscalar processors to provide illusion of multiple processors
  - Higher utilization of processor resources
- **Can schedule each thread as if were separate CPU**
  - However, not linear speedup!
  - If have multiprocessor, should schedule each processor first
- **Original technique called "Simultaneous Multithreading"**
  - See http://www.cs.washington.edu/research/smt/
  - Alpha, SPARC, Pentium 4 ("Hyperthreading"), Power 5
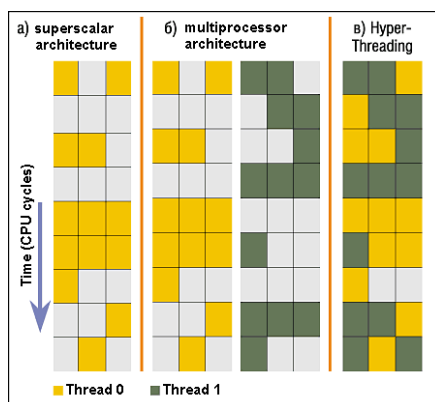
How to protect threads from one another?

- **Need three important things:**
  - Protection of memory
    - » Every task does not have access to all memory
  - Protection of I/O devices
    - » Every task does not have access to every device
  - Preemptive switching from task to task
    - » Use of timer
   - » Must not be possible to disable timer from usercode

Recall: Program's Address Space

- Address space $\Rightarrow$ the set of accessible addresses + state associated with them:
  - For a 32-bit processor there are 2^{32} = 4 billion addresses
- What happens when you read or write to an address?
  - Perhaps Nothing
  - Perhaps acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes I/O operation
    - » (Memory-mapped I/O)
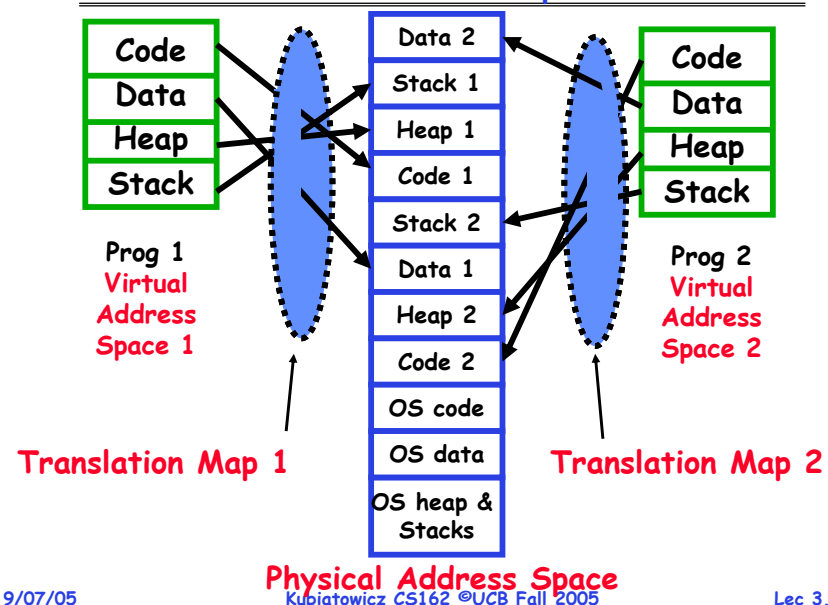 - Perhaps causes exception (fault)
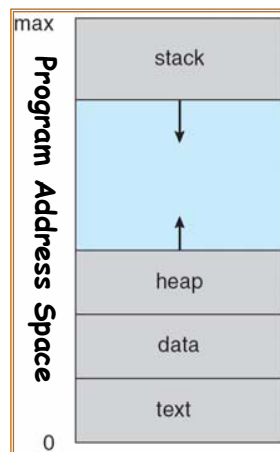
Providing Illusion of Separate Address Space: Load new Translation Map on Switch

Prog 1 Virtual Address Space 1

Prog 2 Virtual Address Space 2

Translation Map 1

Translation Map 2

Physical Address Space

Administriva: Time for Project Signup

- Project Signup:
  - Only submit once per group!
  - Everyone in group must have logged into their cs162-xx accounts before you register the group
  - Make sure that you select at least 2 potential sections
  - Due date: Today by 11:59pm
- Will have sections assigned by end of week
  - Go to new sections next week!

Section	Time	Location	TA
101	Tu 1:00-2:00P	310 Hearst Mining	Dominic
102	W 10:00-11:00A	2 Evans	Rajesh
103	W 11:00-12:00P	85 Evans	Rajesh
104	W 1:00-2:00P	85 Evans	Chris
105	W 2:00-3:00P	85 Evans	Chris

Administrivia (2)

- Cs162-xx accounts:
  - Make sure you got an account form
  - If you haven't logged in yet, you need to do so
- Email addresses
  - We need an email address from you
  - If you haven't given us one already, you should get prompted when you log in again
- Nachos readers:
  - Available from Northside Copy Central
  - Includes printouts of all of the code
- Next Week: Start Project 1
  - Go to Nachos page and start reading up

Traditional UNIX Process

- **Process:** *Operating system abstraction to represent what is needed to run a single program*
  - Often called a "HeavyWeight Process"
  - Formally: a sequential stream of execution in its *own* address space
- **Two parts:**
  - Sequential Program Execution Stream
    - » Code executed as a *single, sequential* stream of execution
    - » Includes State of CPU registers
  - Protected Resources:
    - » Main Memory State (contents of Address Space)
    - » I/O state (i.e. file descriptors)
- **Important: There is no concurrency in a heavyweight process**

How do we multiplex processes?

- **The current state of process held in a process control block (PCB):**
  - This is a "snapshot" of the execution and protection environment
  - Only one PCB active at a time
- **Give out CPU time to different processes (Scheduling):**
  - Only one process "running" at a time
  - Give more time to important processes
- **Give pieces of resources to different processes (Protection):**
  - Controlled access to non-CPU resources
  - Sample mechanisms:
    - » Memory Mapping: Give each process their own address space
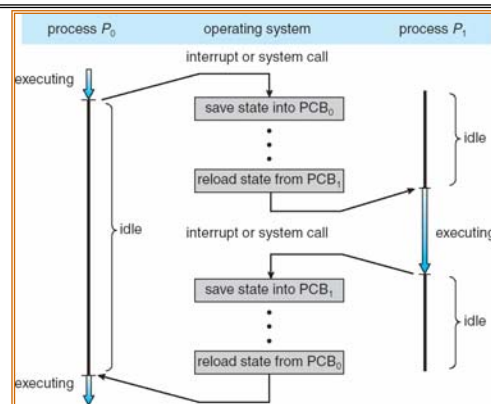   - » Kernel/User duality: Arbitrary multiplexing of I/O through system calls

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

PCB

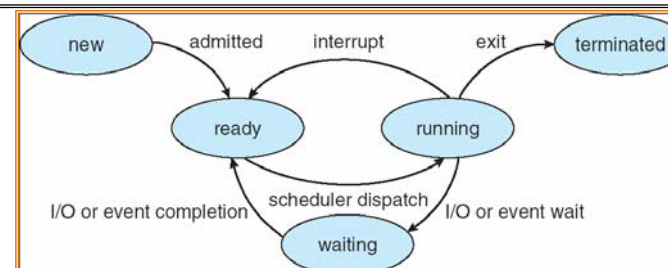## CPU Switch From Process to Process

- **This is also called a "context switch"**
- **Code executed in kernel above is overhead**
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/hyperthreading, but... contention for resources instead

Diagram of Process State

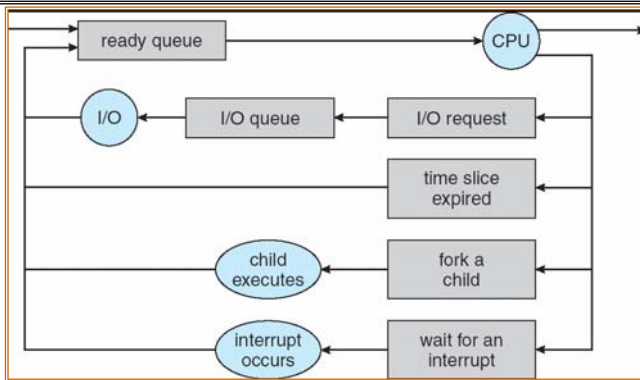- **As a process executes, it changes *state***
  - new: The process is being created
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - ready: The process is waiting to run
  - terminated: The process has finished execution

Process Scheduling

- **PCBs move from queue to queue as they change state**
  - Decisions about which order to remove from queues are **Scheduling** decisions
  - Many algorithms possible (few weeks from now)

What does it take to create a process?

- **Must construct new PCB**
  - Inexpensive
- **Must set up new page tables for address space**
  - More expensive
- **Copy data from parent process? (Unix `fork()`)**
  - Semantics of Unix `fork()` are that the child process gets a complete copy of the parent memory and I/O state
  - Originally *very* expensive
  - Much less expensive with "copy on write"
- **Copy I/O state (file handles, etc)**
  - Medium expense

Process =? Program

```
main ()
{
    …;
}
A() {
    …
}          Program
```

```
main ()          Heap
{
    …;
}                Stack
A() {
                   A
    …             main
}          Process
```
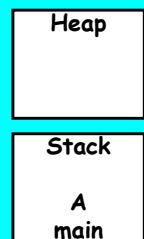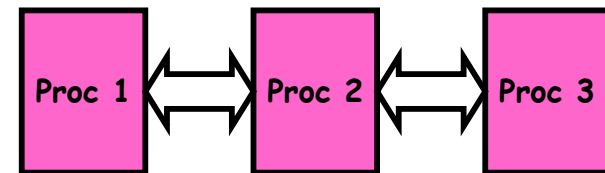
- **More to a process than just a program:**
  - Program is just part of the process state
  - I run emacs on lectures.txt, you run it on homework.java – Same program, different processes
- **Less to a process than a program:**
  - A program can invoke more than one process
  - cc starts up cpp, cc1, cc2, ld, etc

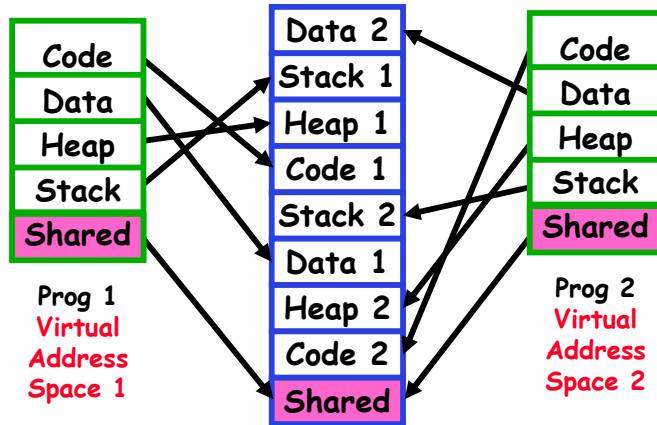## Multiple Processes to Contribute on Task

- **High Creation/memory Overhead**
- **(Relatively) High Context-Switch Overhead**
- **Need Communication mechanism:**
  - Separate Address Spaces Isolates Processes
  - Shared-Memory Mapping
    » Accomplished by mapping addresses to common DRAM
    » Read and Write through memory
  - Message Passing
    » `send()` and `receive()` messages
    » Works across network

Shared Memory Communication

Prog 1 Virtual Address Space 1		Prog 2 Virtual Address Space 2
Code	Data 2	Code
Data	Stack 1	Data
Heap	Heap 1	Heap
Stack	Code 1	Stack
Shared	Stack 2	Shared
	Data 1	
	Heap 2	
	Code 2	
	Shared	

- Communication occurs by "simply" reading/writing to shared address page
  - Really low overhead communication
  - Introduces complex synchronization problems

Inter-process Communication (IPC)

- **Mechanism for processes to communicate and to synchronize their actions**
- **Message system – processes communicate with each other without resorting to shared variables**
- **IPC facility provides two operations:**
  - send(*message*) – **message size fixed or variable**
  - receive(*message*)
- **If *P* and *Q* wish to communicate, they need to:**
  - **establish a *communication link* between them**
  - **exchange messages via send/receive**
- **Implementation of communication link**
  - **physical (e.g., shared memory, hardware bus)**
  - **logical (e.g., logical properties)**

Modern "Lightweight" Process with Threads

- **Thread: *a sequential execution stream within process* (Sometimes called a "Lightweight process")**
  - **Process still contains a single Address Space**
  - **No protection between threads**
- **Multithreading: *a single program made up of a number of different concurrent activities***
  - **Sometimes called multitasking, as in Ada…**
- **Why separate the concept of a thread from that of a process?**
  - **Discuss the "thread" part of a process (concurrency)**
  - **Separate from the "address space" (Protection)**
  - **Heavyweight Process ≡ Process with one thread**

Single and Multithreaded Processes
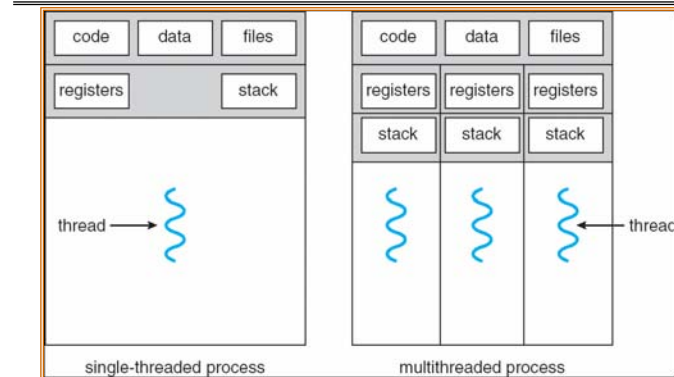
- **Threads encapsulate concurrency**
  - **"Active" component of a process**
- **Address spaces encapsulate protection**
  - **Keeps buggy program from trashing the system**
  - **"Passive" component of a process**

Examples of multithreaded programs

- **Embedded systems**
  - Elevators, Planes, Mecical systems, Wristwatches
  - Single Program, concurrent operations
- **Most modern OS kernel**
  - Internally concurrent because have to deal with concurrent requests by multiple users
  - But no protection needed within kernel
- **Database Servers**
  - Access to shared data by many concurrent users
  - Also background utility processing must be done

Examples of multithreaded programs (con't)

- **Network Servers**
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations
  - File server, Web server, and airline reservation systems
- **Parallel Programming (More than one physical CPU)**
  - Split program into multiple threads for parallelism
  - This is called Multiprocessing

- **Some multiprocessors are actually uniprogrammed:**
  - Multiple threads in one address space but one program at a time

Thread State

- **State shared by all threads in process/addr space**
  - Contents of memory (global variables, heap)
  - I/O state (file system, network connections, etc)
- **State "private" to each thread**
  - Kept in TCB ≡ Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack – what is this?

- **Execution Stack**
  - Parameters, Temporary variables
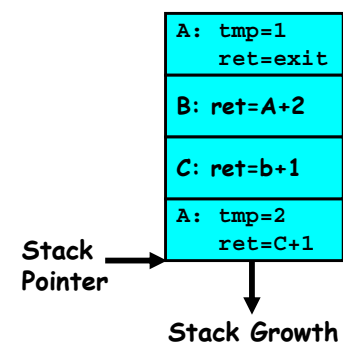 - return PCs are kept while called procedures are executing

Execution Stack Example

```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```


- **Stack holds temporary results**
- **Permits recursive execution**
- **Crucial to modern languages**

Classification

# threads Per AS:	# of addr spaces:	One	Many
One		MS/DOS, early Macintosh	Traditional UNIX
Many		Embedded systems (Geoworks, VxWorks, JavaOS,etc) JavaOS, Pilot(PC)	Mach, OS/2, Linux Windows 95??? Win NT to XP, Solaris, HP-UX, OS X

- **Real operating systems have either**
  - **One or many address spaces**
  - **One or many threads per address space**
- **Did Windows 95/98/ME have real memory protection?**
 - **No: Users could overwrite process tables/System DLLs**

Example: Implementation Java OS

- **Many threads, one Address Space**
- **Why another OS?**
  - **Recommended Minimum memory sizes:**
    » **UNIX + X Windows: 32MB**
    » **Windows 98: 16-32MB**
    » **Windows NT: 32-64MB**
    » **Windows 2000/XP: 64-128MB**
  - **What if want a cheap network point-of-sale computer?**
    » **Say need 1000 terminals**
    » **Want < 8MB**
- **What language to write this OS in?**
  - **C/C++/ASM? Not terribly high-level. Hard to debug.**
  - **Java/Lisp? Not quite sufficient – need direct access to HW/memory management**

Java OS Structure

Java APPS
OS
Hardware
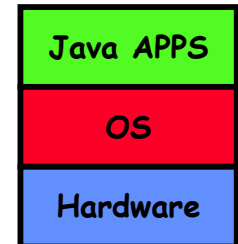
Summary

- **Processes have two parts**
  - **Threads (Concurrency)**
  - **Address Spaces (Protection)**
- **Concurrency accomplished by multiplexing CPU Time:**
  - **Unloading current thread (PC, registers)**
  - **Loading new thread (PC, registers)**
  - **Such context switching may be voluntary (yield(), I/O operations) or involuntary (timer, other interrupts)**
- **Protection accomplished restricting access:**
  - **Memory mapping isolates processes from each other**
  - **Dual-mode for isolating I/O, other resources**
- **Book talks about processes**
  - **When this concerns concurrency, really talking about thread portion of a process**
  - **When this concerns protection, talking about address space portion of a process**