COS 318: Operating Systems

Processes and Threads

Jaswinder Pal Singh
Computer Science Department
Princeton University

(http://www.cs.princeton.edu/courses/cos318/)

Today' s Topics

- Concurrency
- Processes
- Threads

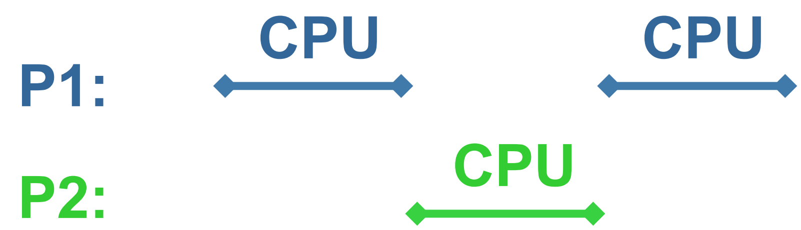
- Reminder:
  - Hope you're all busy working on your implementations

Concurrency and Processes

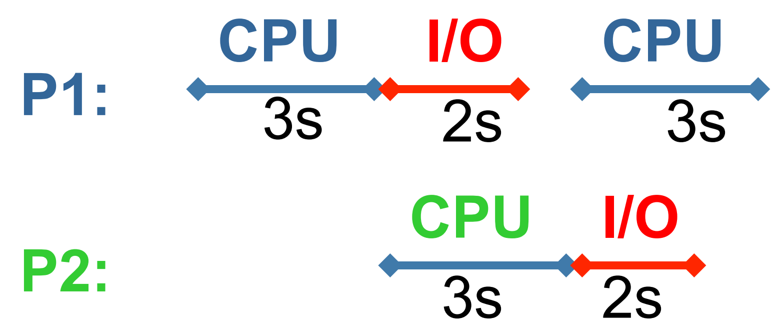
- Concurrency
  - Hundreds of jobs going on in a system
  - CPU is shared, as are I/O devices
- Concurrency via Processes
  - Decompose complex problems into simple ones
  - Make each simple one a process
  - Processes run ‘concurrently’ but each process feels like it has its own computer
- Example: gcc (via “gcc –pipe –v”) launches the following
  - /usr/libexec/cpp | /usr/libexec/cc1 | /usr/libexec/as | /usr/libexec/elf/ld
  - Each instance of cpp, cc1, as and ld running is a process

Process Concurrency

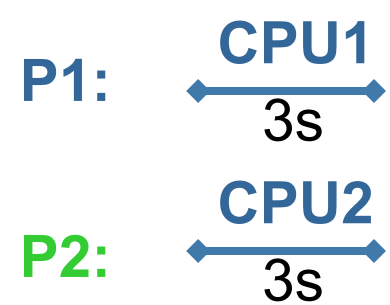
- Virtualization
  - Processes interleaved on CPU



- I/O concurrency
  - I/O for P1 overlapped with CPU for P2
  - Each runs almost as fast as if it has its own computer
  - Reduce total completion time



- CPU parallelism
  - Multiple CPUs (such as SMP)
  - Processes running in parallel
  - Speedup



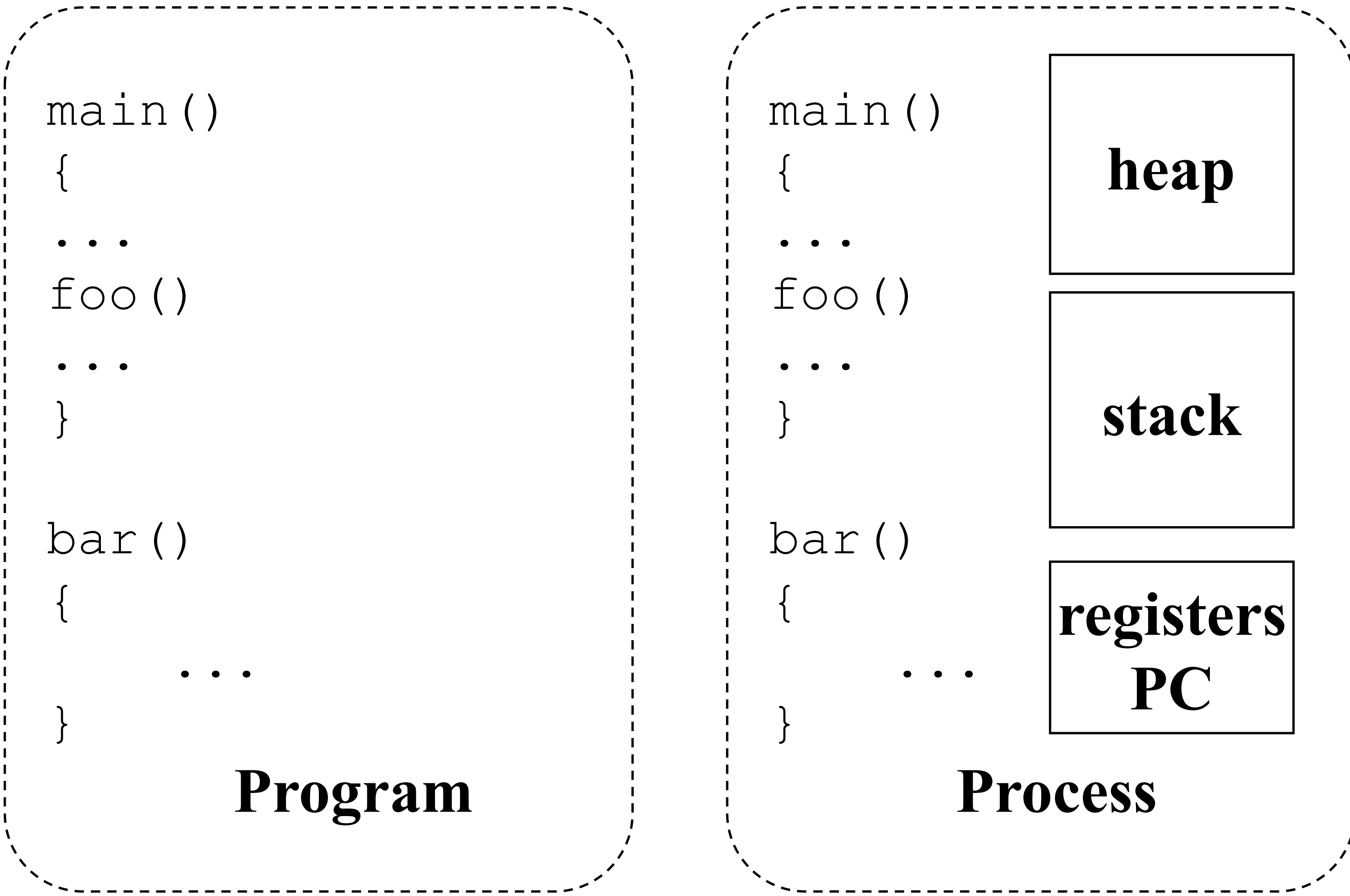
Parallelism

- Parallelism is common in real life
  - A single sales person sells $1M annually
  - Hire 100 sales people to generate $100M revenue
- Speedup
  - Ideal speedup is factor of N
  - Reality: bottlenecks + coordination overhead reduce speedup
- Questions
  - Can you speed up by working with a partner?
  - Can you speed up by working with 20 partners?
  - Can you get super-linear (more than a factor of N) speedup?

Simplest Process

- Sequential execution
  - No concurrency inside a process
  - Everything happens sequentially
  - Some coordination may be required
- Process state
  - Registers
  - Main memory
  - I/O devices
    - File system
    - Communication ports
  - …

Program and Process



Process vs. Program

- Process > program
  - Program is just the code; just part of process state
  - Example: many users can run the same program

- Process < program
  - A program can invoke more than one process
  - Example: Fork off processes
  - Many processes can be running the same program

Managing Processes: Process Control Block (PCB)

- Process management info
  - Identification
  - State
    - Ready: ready to run.
    - Running: currently running.
    - Blocked: waiting for resources
  - Registers, EFLAGS, EIP, and other CPU state
  - Stack, code and data segment
  - Parents, etc
- Memory management info
  - Segments, page table, stats, etc
- I/O and file management
  - Communication ports, directories, file descriptors, etc.
- Resource allocation and accounting information

API for Process Management

- Creation and termination
  - Exec, Fork, Wait, Kill
- Signals
  - Action, Return, Handler
- Operations
  - Block, Yield
- Synchronization
  - We will talk about this a lot more later

Create A Process

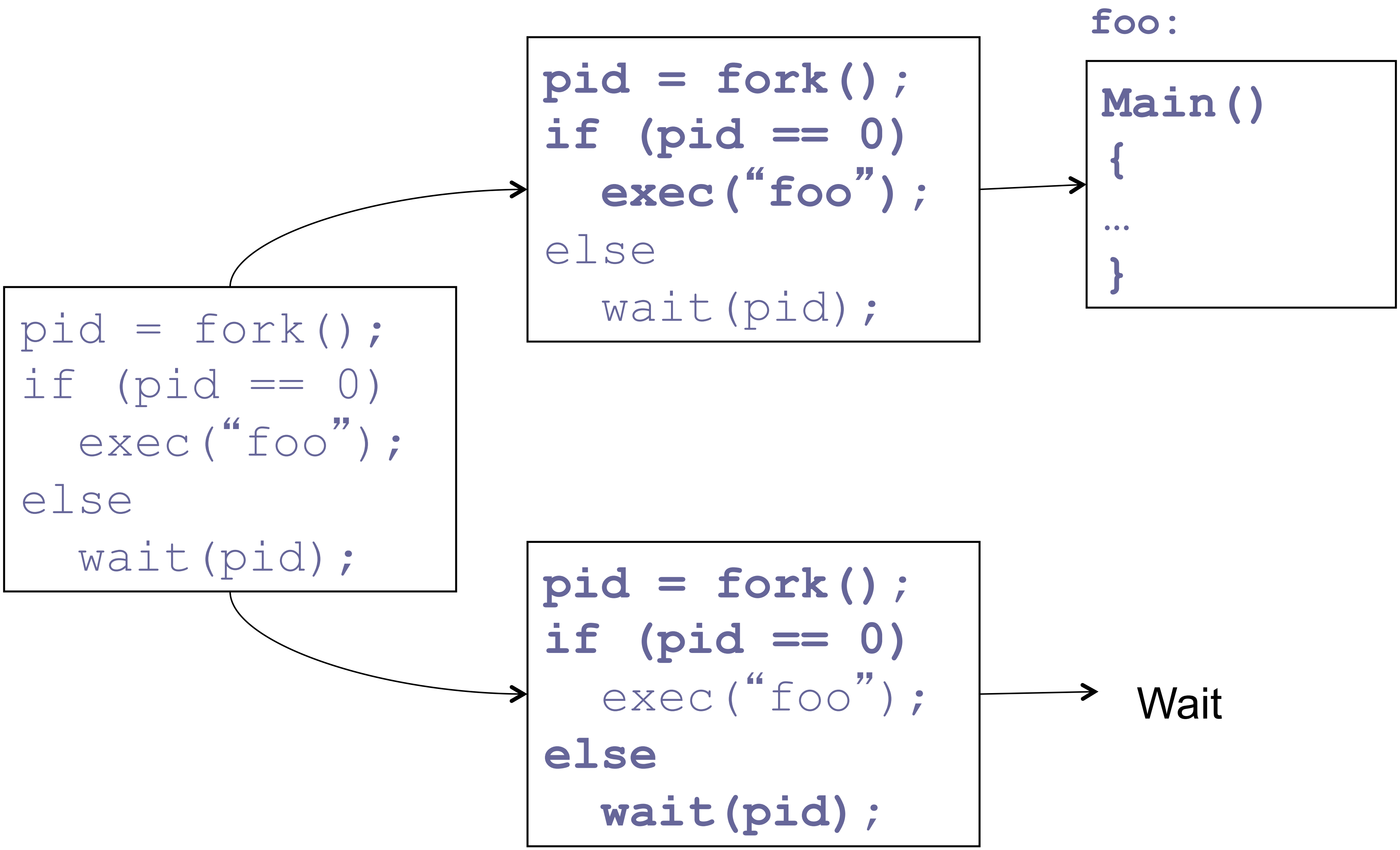
- Creation
  - Load code and data into memory
  - Create an empty call stack
  - Initialize state
  - Make the process ready to run
- Cloning a process
  - Save state of current process
  - Make copy of current code, data, stack and OS state
  - Make the process ready to run

Unix Example

- Methods to make processes:
  - fork clones a process
  - exec overlays the current process

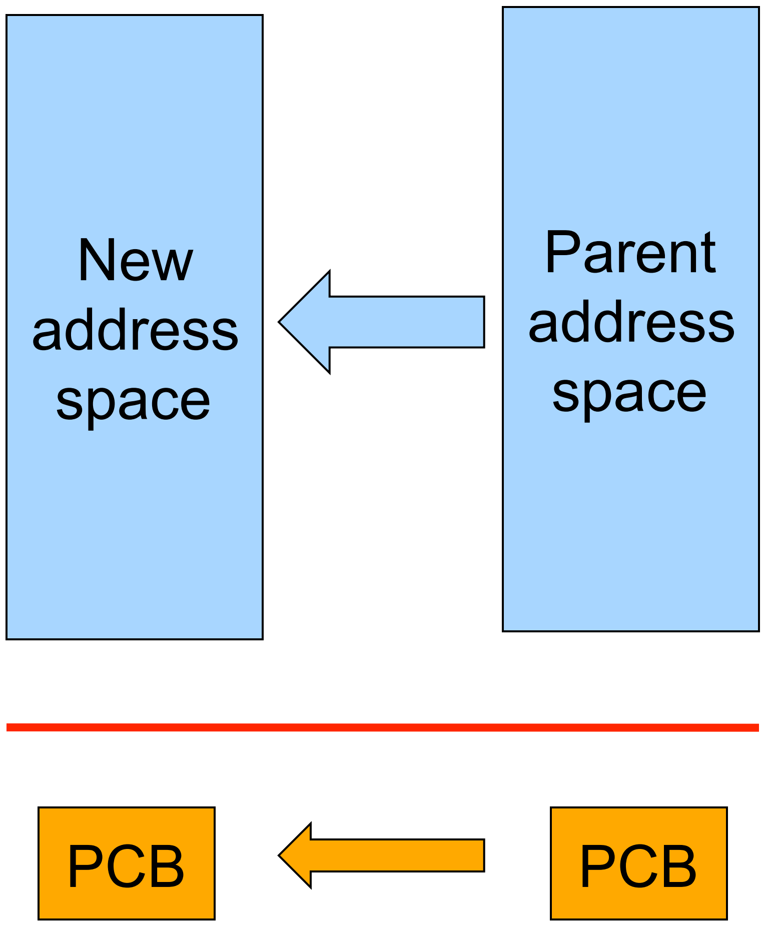
```
pid = fork();
if (pid == 0)
   /* child process */
   exec("foo");   /* does not return */
Else
   /* parent */
   wait(pid);     /* wait for child to die */
```

Fork and Exec in Unix



More on Fork

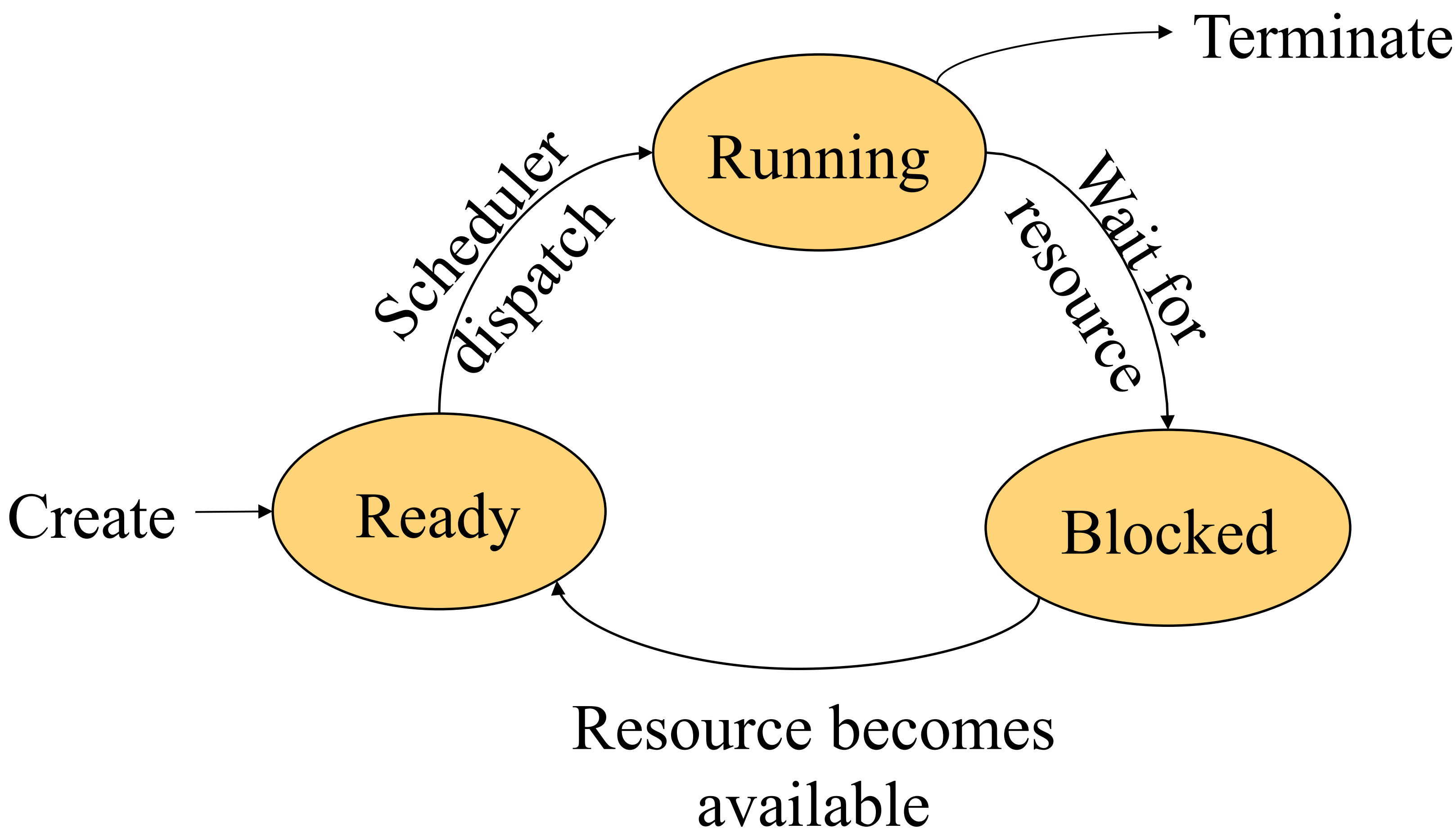
- Parent process has a PCB and an address space
- Create and initialize PCB
- Create an address space
- Copy the content of the parent address space to the new address space
- Inherit the execution context of the parent
- New process is ready



Process Context Switch

- Save a context (everything that a process may damage)
  - All registers (general purpose and floating point)
  - All co-processor state
  - Save all memory to disk?
  - What about cache and TLB?
- Start a context
  - Does the reverse
- Challenge
  - OS code must save state without changing any state
  - E.g. how should OS run without touching any registers?
    - CISC machines have a special instruction to save and restore all registers on stack
    - RISC: reserve registers for kernel or have way to carefully save one and then continue

(Reduced) Process State Transition



Threads

- Thread
  - A sequential execution stream within a process (also called lightweight process)
  - Threads in a process share the same address space
- Thread concurrency
  - Easier to program overlapping I/O and CPU with threads than with signals
  - Human being likes to do several things at a time
  - A server (e.g. file server) serves multiple requests
  - Multiple CPUs sharing the same memory

Thread Control Block (TCB)

- State
  - Ready: ready to run
  - Running: currently running
  - Blocked: waiting for resources
- Registers
- Status (EFLAGS)
- Program counter (EIP)
- Stack
- Code

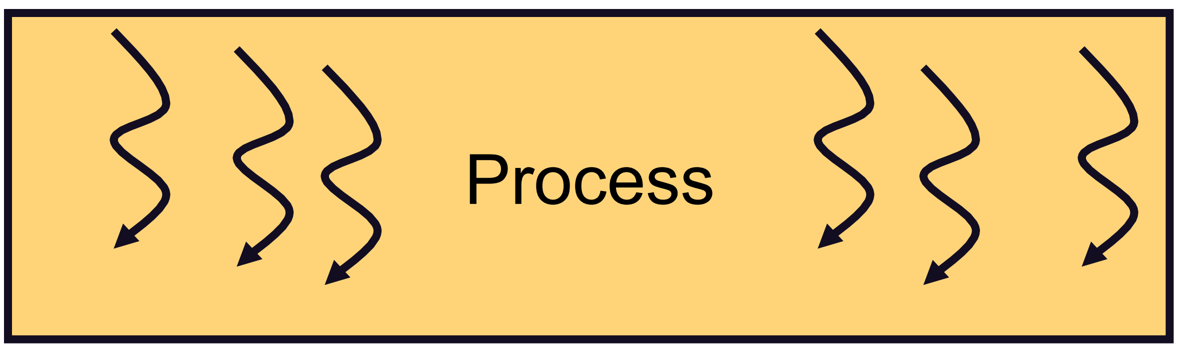
Typical Thread API

- Creation
  - Fork, Join
- Mutual exclusion
  - Acquire (lock), Release (unlock)
- Condition variables
  - Wait, Signal, Broadcast
- Alert
  - Alert, AlertWait, TestAlert

Revisit Process

- Process
  - Threads
  - Address space
  - Environment for the threads to run on OS (open files, etc)

- Simplest process has 1 thread



Thread Context Switch

- Save a context (everything that a thread may damage)
  - All registers (general purpose and floating point)
  - All co-processor state
  - Need to save stack?
  - What about cache and TLB?
- Start a context
  - Does the reverse
- May trigger a process context switch

Procedure Call

- Caller or callee save some context (same stack)
- Caller saved example:

```
save active caller registers
call foo
                    foo() {
                          do stuff
                    }

restore caller regs
```

Threads vs. Procedures

- Threads may resume out of order
  - Cannot use LIFO stack to save state
  - Each thread has its own stack
- Threads switch less often
  - Do not partition registers
  - Each thread "has" its own CPU
- Threads can be asynchronous
  - Procedure call can use compiler to save state synchronously
  - Threads can run asynchronously
- Multiple threads
  - Multiple threads can run on multiple CPUs in parallel
  - Procedure calls are sequential

Process vs. Threads

- Address space
  - Processes do not usually share memory (address space)
  - Process context switch page table and other memory mechanisms
  - Threads in a process share the entire address space
- Privileges
  - Processes have their own privileges (file accesses, e.g.)
  - Threads in a process share all privileges
- Question
  - Do you really want to share the “entire” address space?

Real Operating Systems

- One or many address spaces
- One or many threads per address space

	1 address space	Many address spaces
1 thread per address space	MSDOS Macintosh	Traditional Unix
Many threads per address spaces	Embedded OS, Pilot	VMS, Mach (OS-X), OS/2, Windows NT/XP/Vista/7, Solaris, HP-UX, Linux

Summary

- Concurrency
  - CPU and I/O
  - Among applications
  - Within an application
- Processes
  - Abstraction for application concurrency
- Threads
  - Abstraction for concurrency within an application