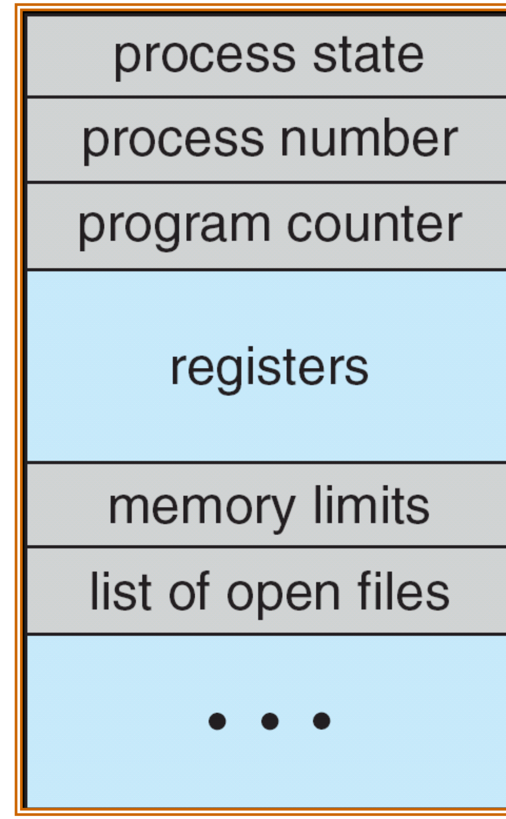
Concurrency (Processes and Threads)

Edited from the slides in

http://cs162.eecs.Berkeley.edu

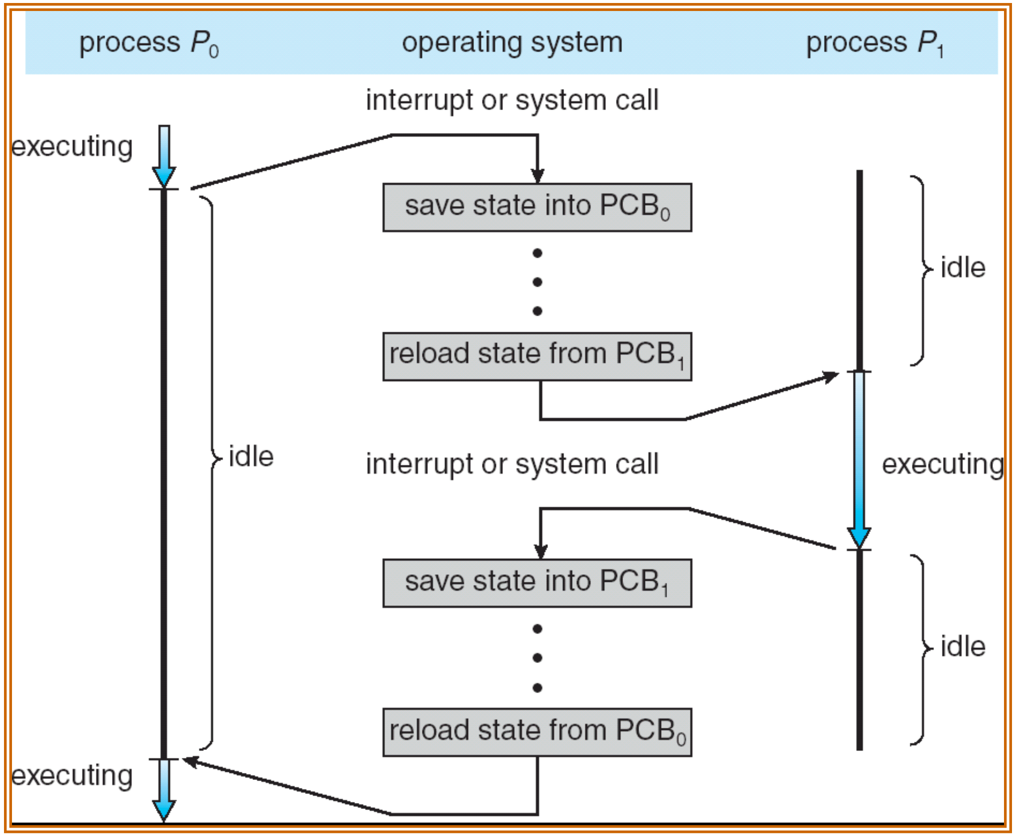
How do we Multiplex Processes?

- 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
  - Example mechanisms:
    - » Memory Translation: Give each process their own address space
    - » Kernel/User duality: Arbitrary multiplexing of I/O through system calls



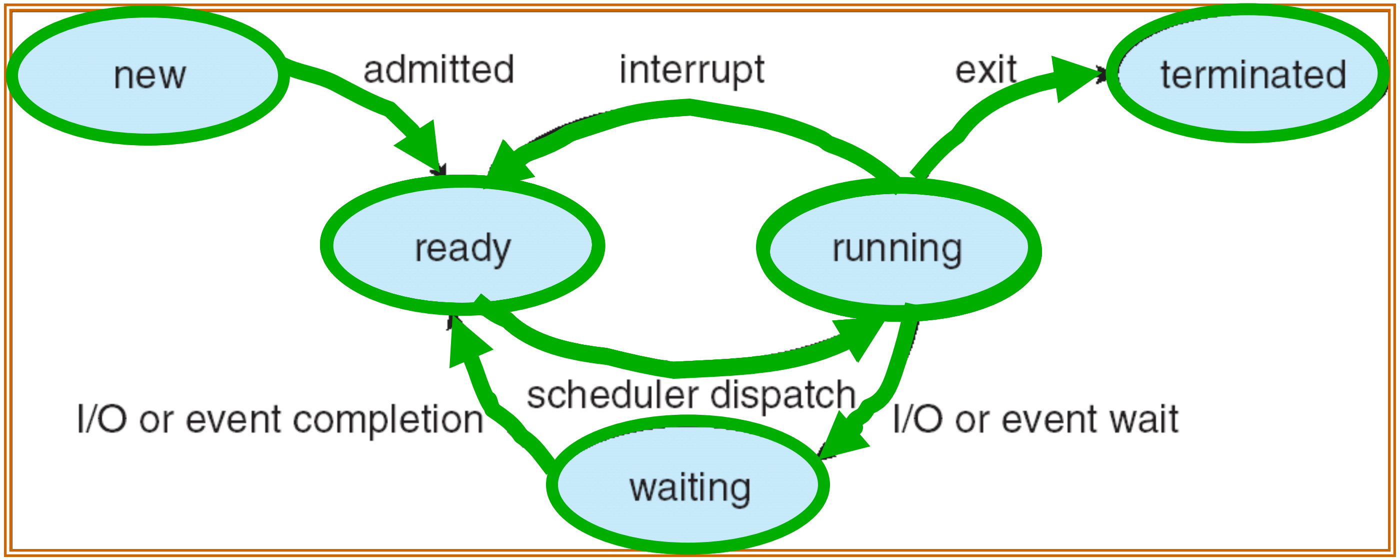
Process Control Block

CPU Switch From Process A to Process B



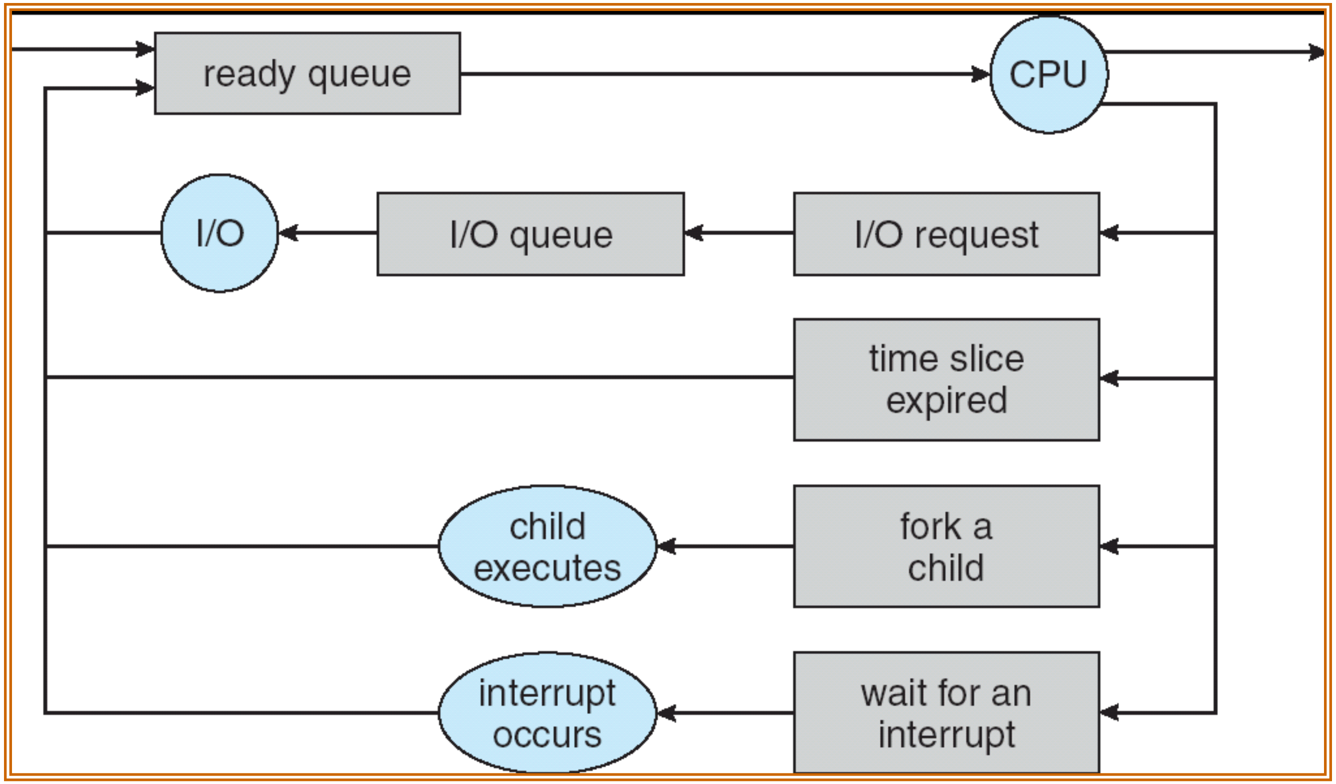
- This is also called a "context switch"
- Code executed in kernel above is ***overhead***
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/hyperthreading, but… contention for resources instead

Lifecycle of a Process



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

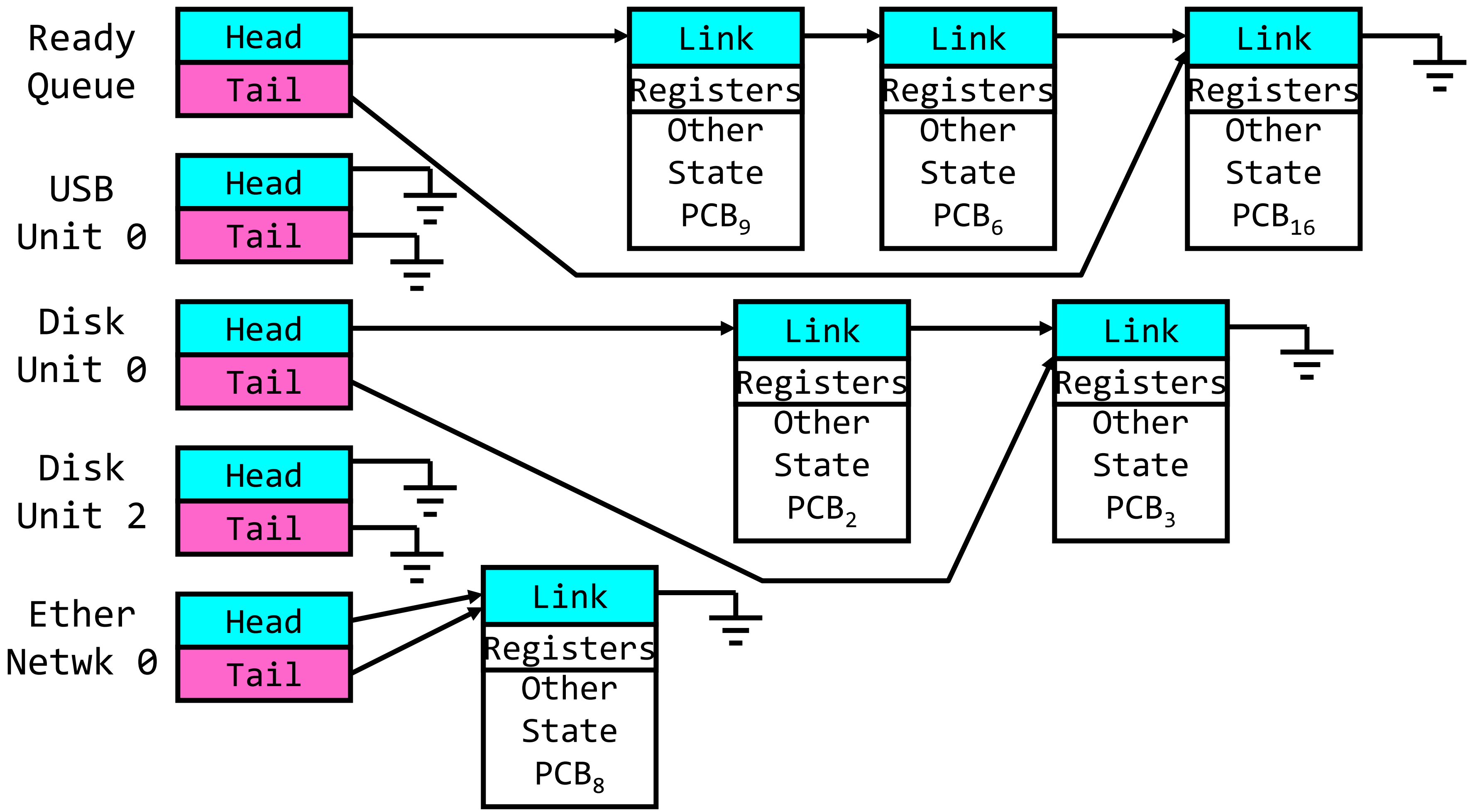
Process Scheduling



- 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)

Ready Queue And Various I/O Device Queues

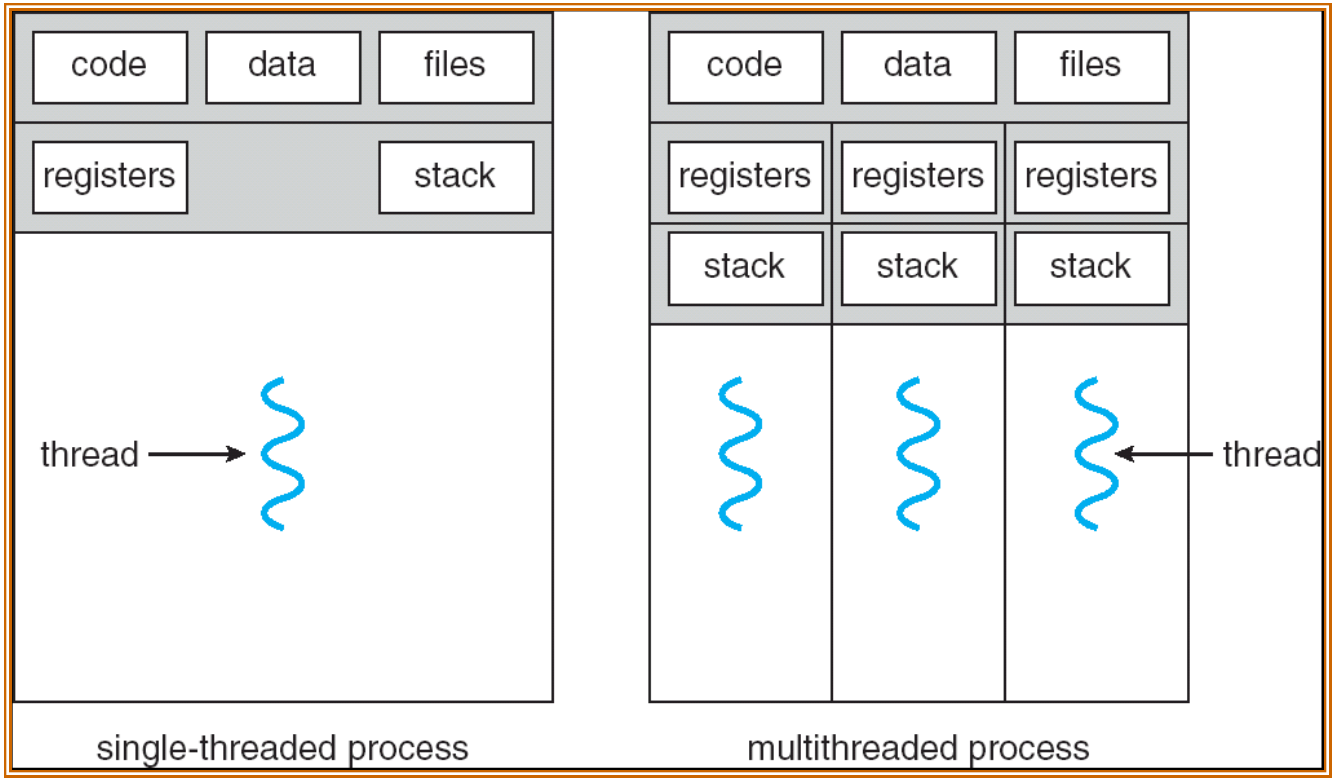
- Process not running ⇒ PCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy



Modern 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
- Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

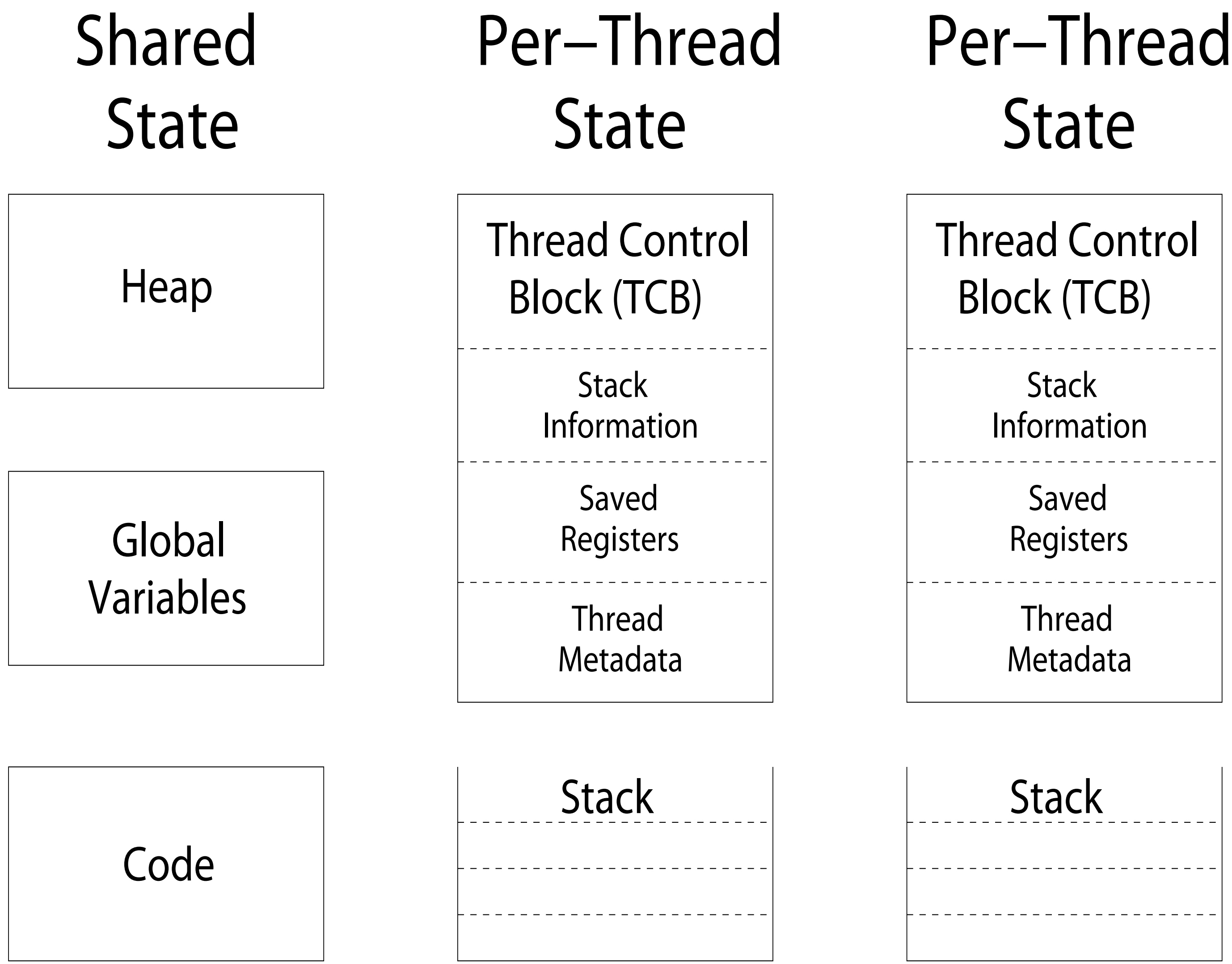
Thread State

- State shared by all threads in process/address space
  - Content of memory (global variables, heap)
  - I/O state (file descriptors, 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

Shared vs. Per-Thread State



Execution Stack Example

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

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

Execution Stack Example

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

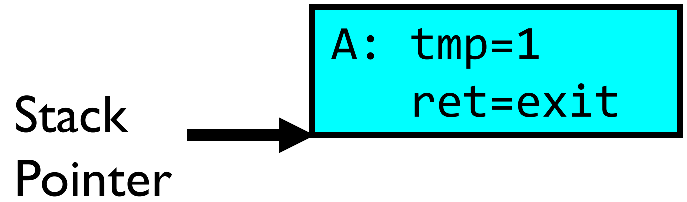
Stack Pointer →

```
A: tmp=1
   ret=exit
```

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

Execution Stack Example

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



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

Execution Stack Example

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

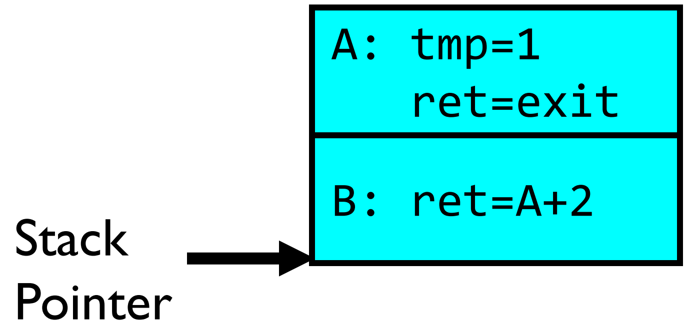
Stack Pointer →

```
A: tmp=1
   ret=exit
```

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

Execution Stack Example

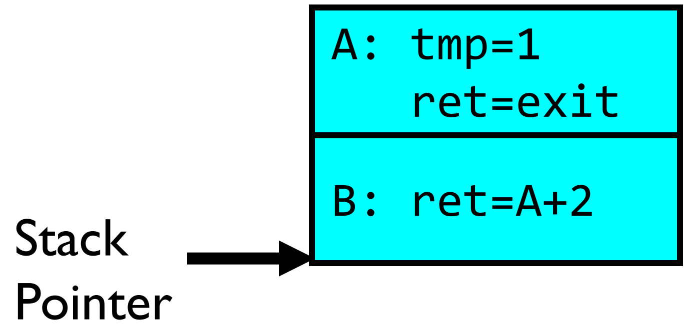
```
      A(int tmp) {
A:      if (tmp<2)
A+1:      B();
A+2:    printf(tmp);
      }
      B() {
B:      C();
B+1:  }
      C() {
C:      A(2);
C+1:  }
      A(1);
exit:
```



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

Execution Stack Example

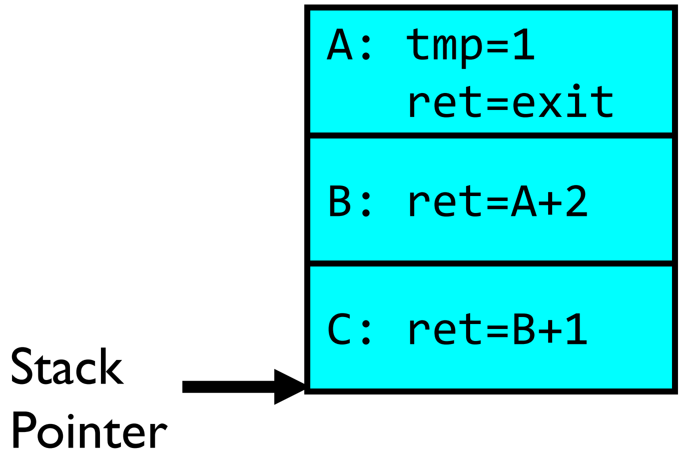
```
      A(int tmp) {
A:      if (tmp<2)
A+1:      B();
A+2:    printf(tmp);
      }
      B() {
B:      C();
B+1:  }
      C() {
C:      A(2);
C+1:  }
      A(1);
exit:
```



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

Execution Stack Example

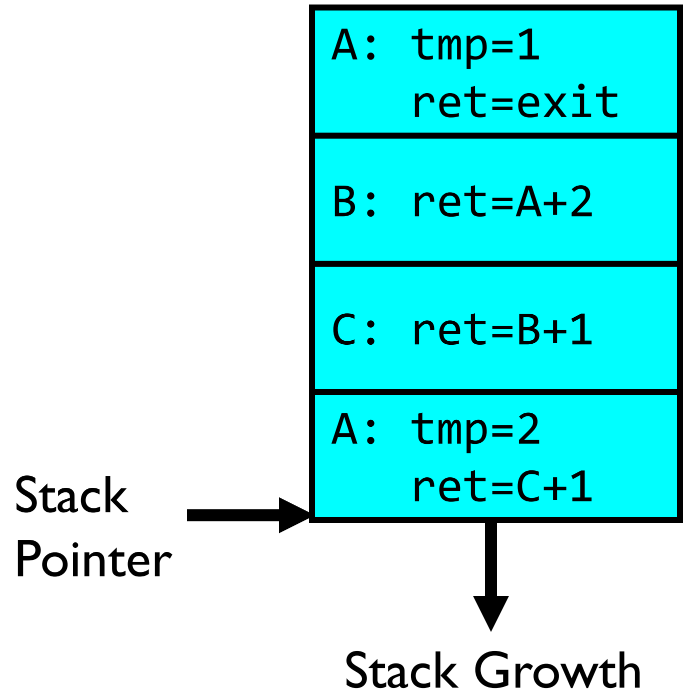
```
         A(int tmp) {
A:         if (tmp<2)
A+1:         B();
A+2:       printf(tmp);
         }
         B() {
B:         C();
B+1:     }
         C() {
C:         A(2);
C+1:     }
         A(1);
exit:
```



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

Execution Stack Example

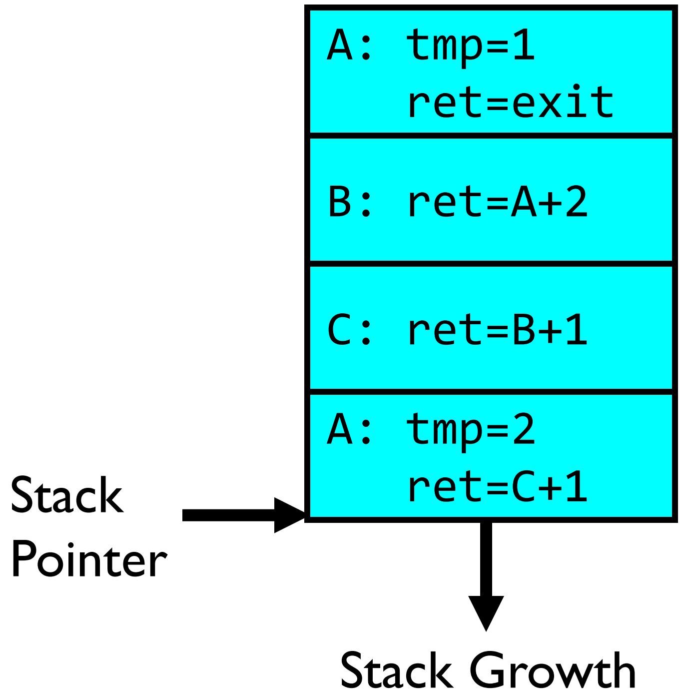
```
        A(int tmp) {
A:        if (tmp<2)
A+1:        B();
A+2:      printf(tmp);
        }
        B() {
B:        C();
B+1:    }
        C() {
C:        A(2);
C+1:    }
        A(1);
exit:
```



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

Execution Stack Example

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

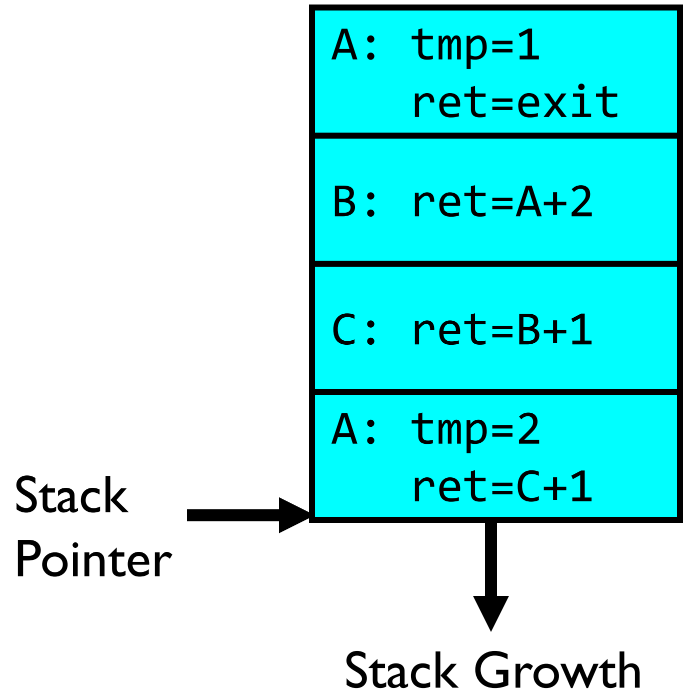


Output: >2

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

Execution Stack Example

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

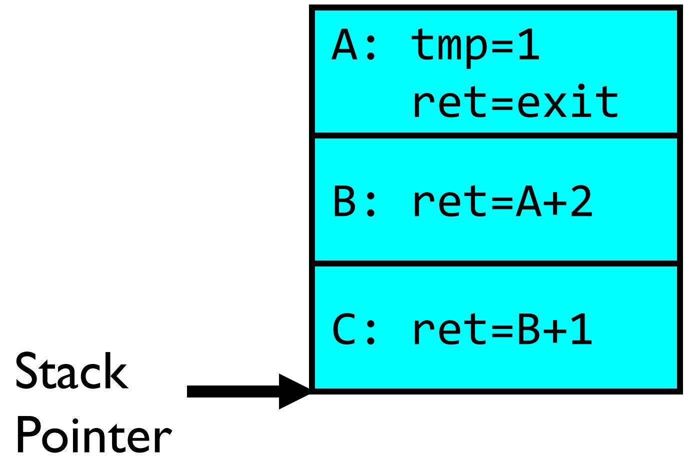


Output: >2

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

Execution Stack Example

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

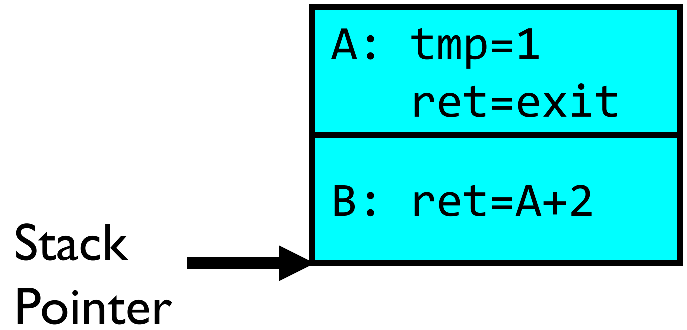


Output: >2

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

Execution Stack Example

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



Output: >2

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

Execution Stack Example

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

Stack Pointer → `A: tmp=1 ret=exit`

Output: `>2 1`

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

Execution Stack Example

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

Stack Pointer → `A: tmp=1 ret=exit`

Output: `>2 1`

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

Execution Stack Example

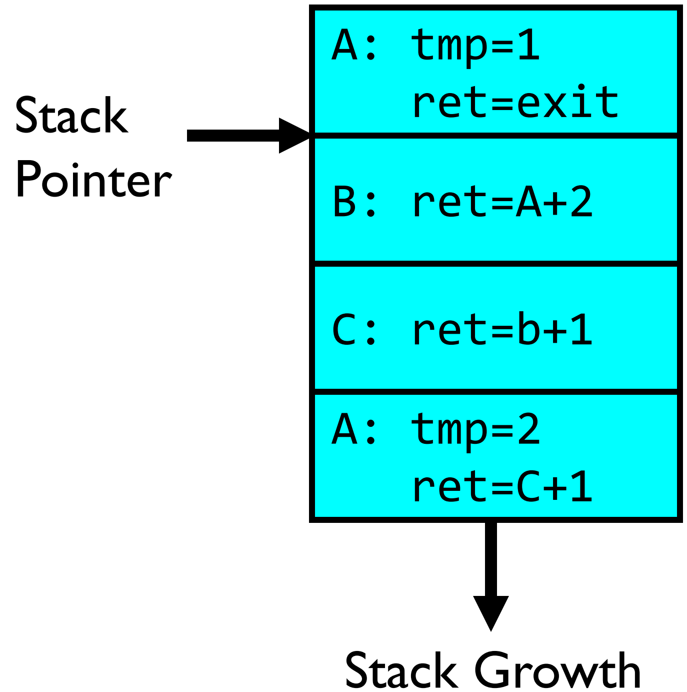
```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```

Output: `>2 1`

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

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

Motivational Example for Threads

- Imagine the following C program:

```
main() {
    ComputePI("pi.txt");
    PrintClassList("classlist.txt");
}
```

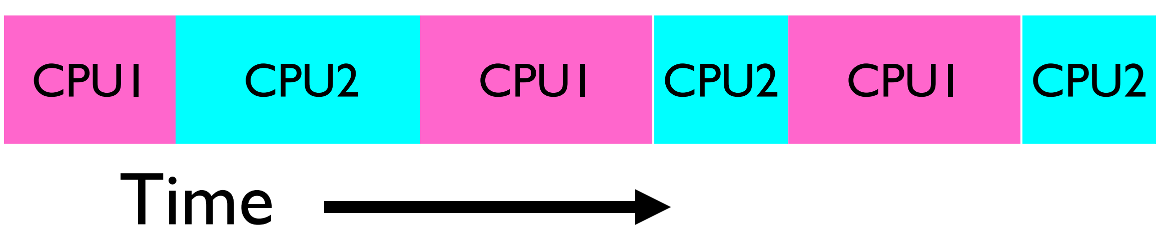
- What is the behavior here?
  - Program would never print out class list
  - Why? `ComputePI` would never finish

Use of Threads

- Version of program with Threads (loose syntax):

```
main() {
    ThreadFork(ComputePI, "pi.txt"));
    ThreadFork(PrintClassList, "classlist.txt"));
}
```

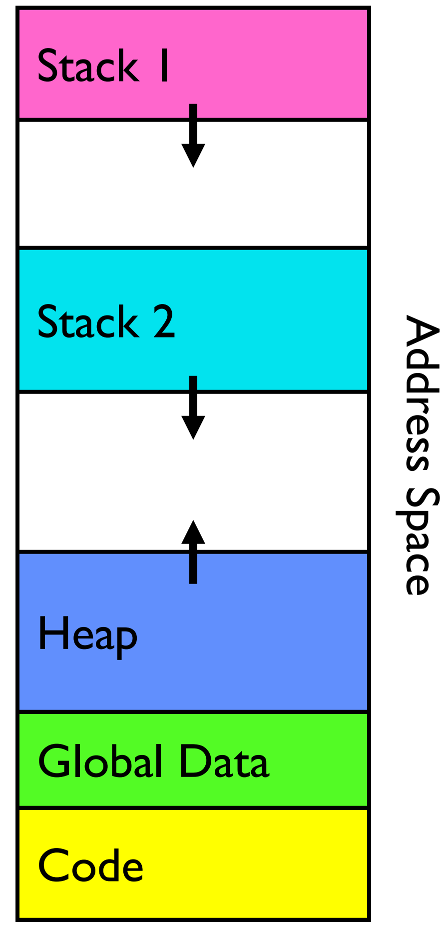
- What does `ThreadFork()` do?
  - Start independent thread running given procedure
- What is the behavior here?
  - Now, you would actually see the class list
  - This *should* behave as if there are two separate CPUs



Memory Footprint: Two-Threads

- If we stopped this program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks

- Questions:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?



Actual Thread Operations

- `thread_fork(func, args)`
  - Create a new thread to run func(args)
  - pThreads: `thread_create`
- `thread_yield()`
  - Relinquish processor voluntarily
  - pThreads : `thread_yield`
- `thread_join(thread)`
  - In parent, wait for forked thread to exit, then return
  - pThreads : `thread_join`
- `thread_exit`
  - Quit thread and clean up, wake up joiner if any
  - pThreads : `thread_exit`

- `pThreads`: POSIX standard for thread programming [POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)]

Dispatch Loop

- Conceptually, the dispatching loop of the operating system looks as follows:

```
Loop {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
}
```

- This is an *infinite* loop
  - One could argue that this is all that the OS does
- Should we ever exit this loop???
  - When would that be?

Running a thread

Consider:

```
RunThread()
...
LoadStateOfCPU(newTCB)
```

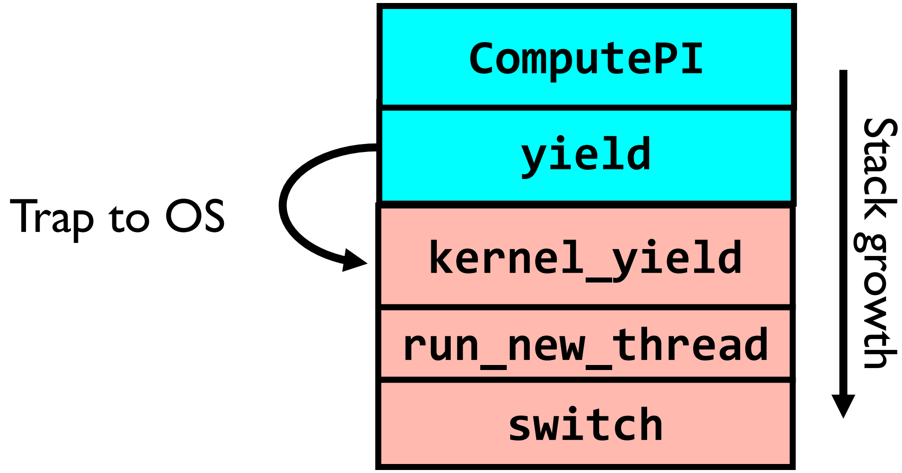
- How do I run a thread?
  - Load its state (registers, PC, stack pointer) into CPU
  - Load environment (virtual memory space, etc)
  - Jump to the PC
- How does the dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets *preempted*

Internal Events

- Blocking on I/O
  - The act of requesting I/O implicitly yields the CPU
- Waiting on a “signal” from other thread
  - Thread asks to wait and thus yields the CPU
- Thread executes a **`yield()`**
  - Thread volunteers to give up CPU

```
computePI() {
    while(TRUE) {
        ComputeNextDigit();
        yield();
    }
}
```

Stack for Yielding Thread



- How do we run a new thread?

```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* Do any cleanup */
}
```

- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack pointer
  - Maintain isolation for each thread

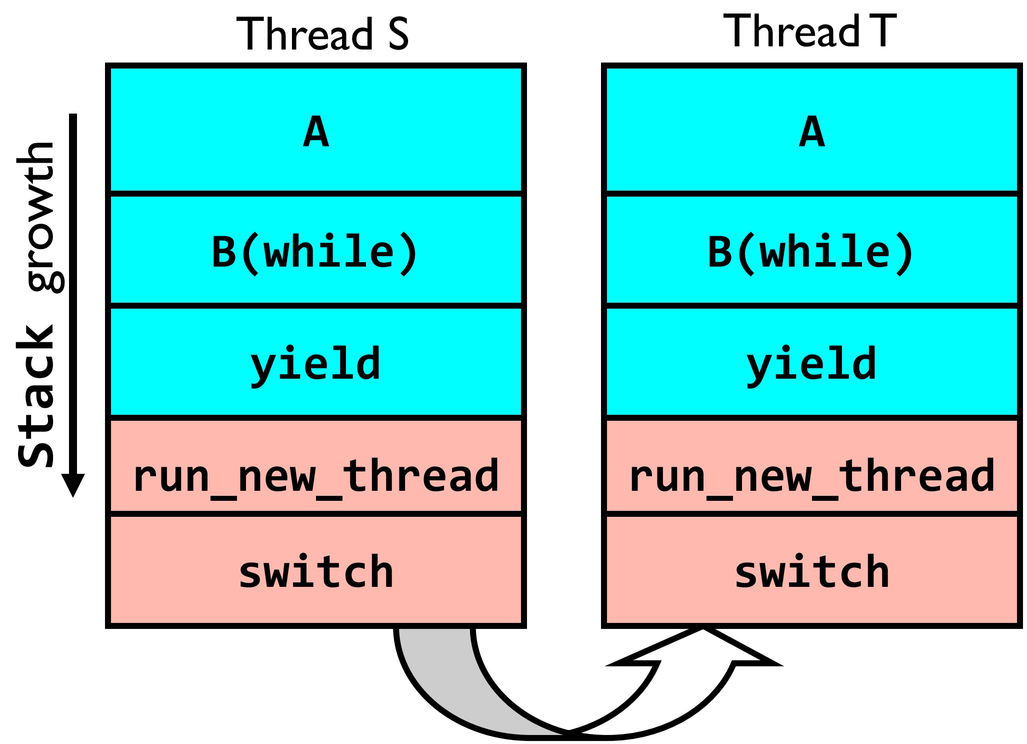
What Do the Stacks Look Like?

- Consider the following code blocks:

```
proc A() {
    B();

}
proc B() {
    while(TRUE) {
        yield();
    }
}
```

- Suppose we have 2 threads:
  – Threads S and T



Saving/Restoring state (often called "Context Switch)

```
Switch(tCur,tNew) {
   /* Unload old thread */
   TCB[tCur].regs.r7 = CPU.r7;
           …
   TCB[tCur].regs.r0 = CPU.r0;
   TCB[tCur].regs.sp = CPU.sp;
   TCB[tCur].regs.retpc = CPU.retpc; /*return addr*/

   /* Load and execute new thread */
   CPU.r7 = TCB[tNew].regs.r7;
           …
   CPU.r0 = TCB[tNew].regs.r0;
   CPU.sp = TCB[tNew].regs.sp;
   CPU.retpc = TCB[tNew].regs.retpc;
   return; /* Return to CPU.retpc */
}
```

Switch Details (continued)

- What if you make a mistake in implementing switch?
  - Suppose you forget to save/restore register 32
  - Get intermittent failures depending on when context switch occurred and whether new thread uses register 32
  - System will give wrong result without warning
- Can you devise an exhaustive test to test switch code?
  - No! Too many combinations and inter-leavings
- Cautionary tale:
  - For speed, Topaz kernel saved one instruction in switch()
  - Carefully documented! Only works as long as kernel size < 1MB
  - What happened?
    - » Time passed, People forgot
    - » Later, they added features to kernel (no one removes features!)
    - » Very weird behavior started happening
  - Moral of story: Design for simplicity

Some Numbers

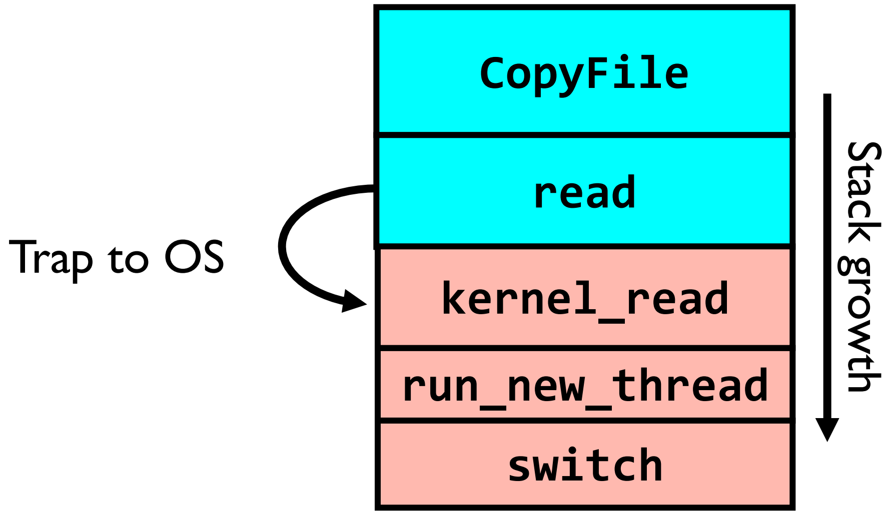
- Frequency of performing context switches: 10-100ms
- Context switch time in Linux: 3-4 μsecs(0.003ms) (Intel i7 & E5)
  - Thread switching faster than process switching (100 ns(0.0001ms))
  - But switching across cores ~2x more expensive than within-core

- Context switch time increases sharply with size of working set*
  - Can increase 100x or more

  *The working set is subset of memory used by process in a time window

- Moral: context switching depends mostly on cache limits and the process or thread's hunger for memory

What happens when thread blocks on I/O?



- What happens when a thread requests a block of data from the file system?
  - User code invokes a system call
  - Read operation is initiated
  - Run new thread/switch
- Thread communication similar
  - Wait for Signal/Join
  - Networking

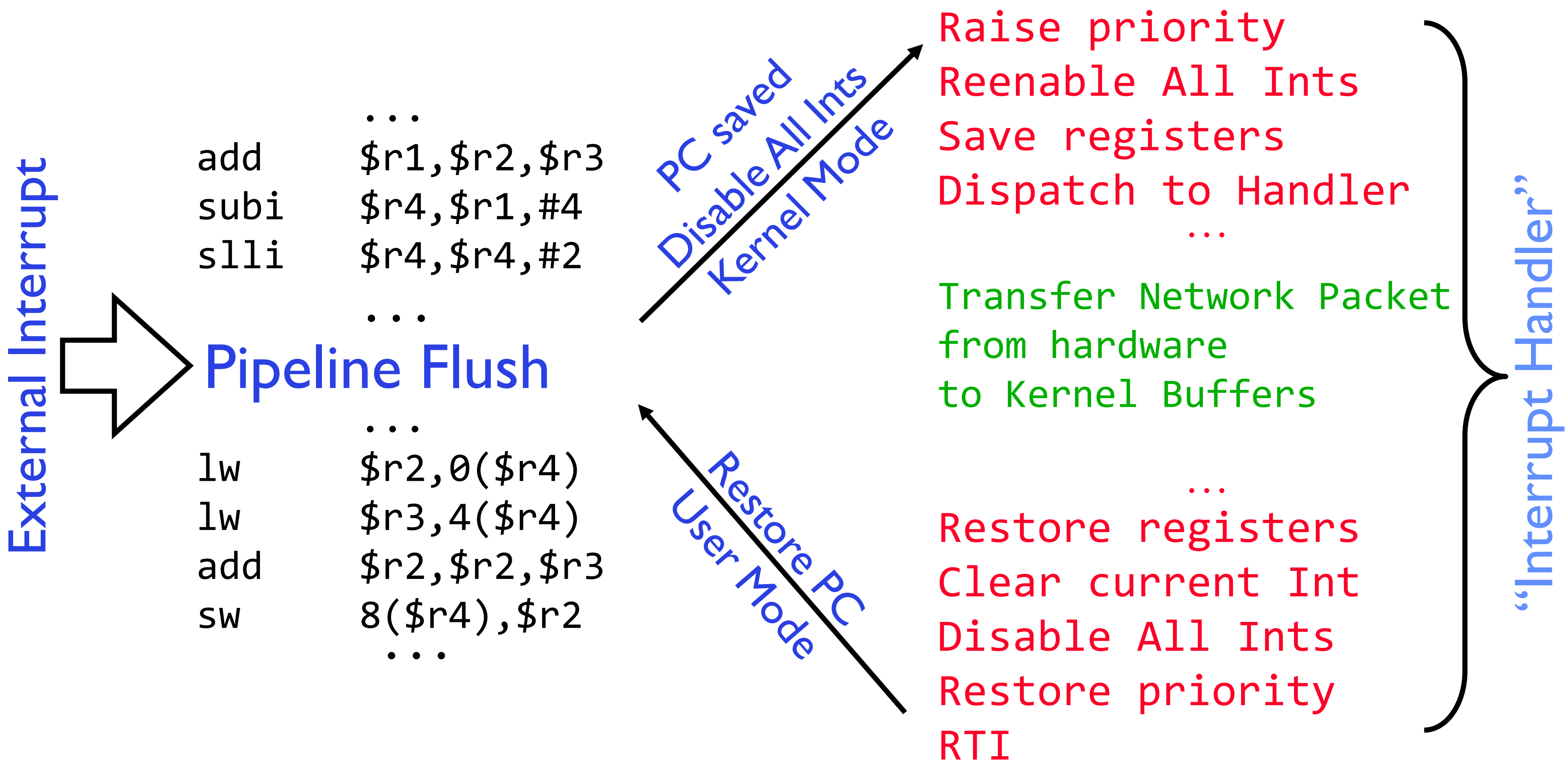
External Events

- What happens if thread never does any I/O, never waits, and never yields control?
  - Could the `ComputePI` program grab all resources and never release the processor?
    - » What if it didn't print to console?
  - Must find way that dispatcher can regain control!

- Answer: utilize external events
  - Interrupts: signals from hardware or software that stop the running code and jump to kernel
  - Timer: like an alarm clock that goes off every some milliseconds

- If we make sure that external events occur frequently enough, can ensure dispatcher runs

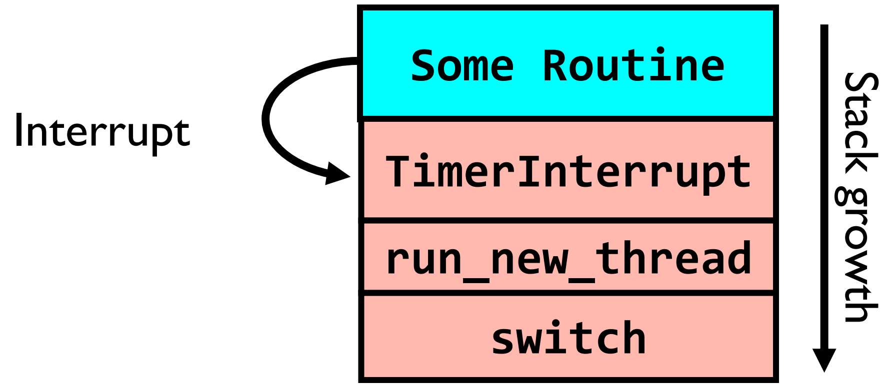
Example: Network Interrupt



- An interrupt is a hardware-invoked context switch
  - No separate step to choose what to run next
  - Always run the interrupt handler immediately

Use of Timer Interrupt to Return Control

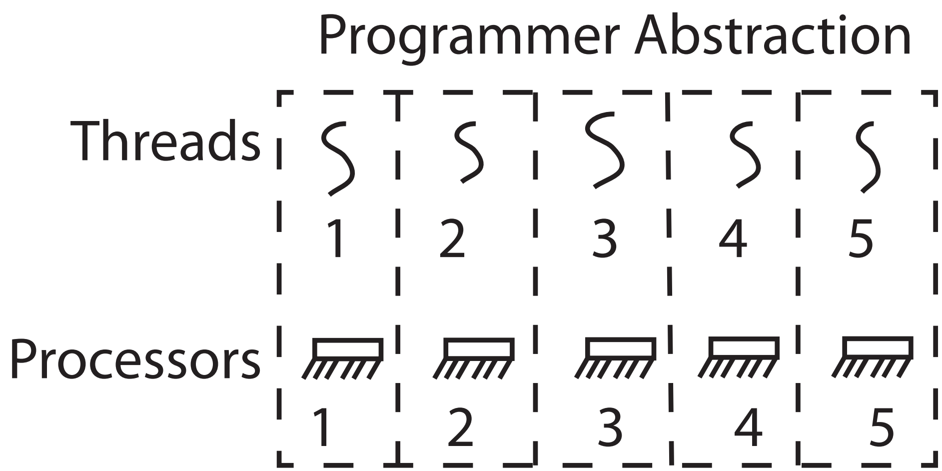
- Solution to our dispatcher problem
  – Use the timer interrupt to force scheduling decisions



- Timer Interrupt routine:

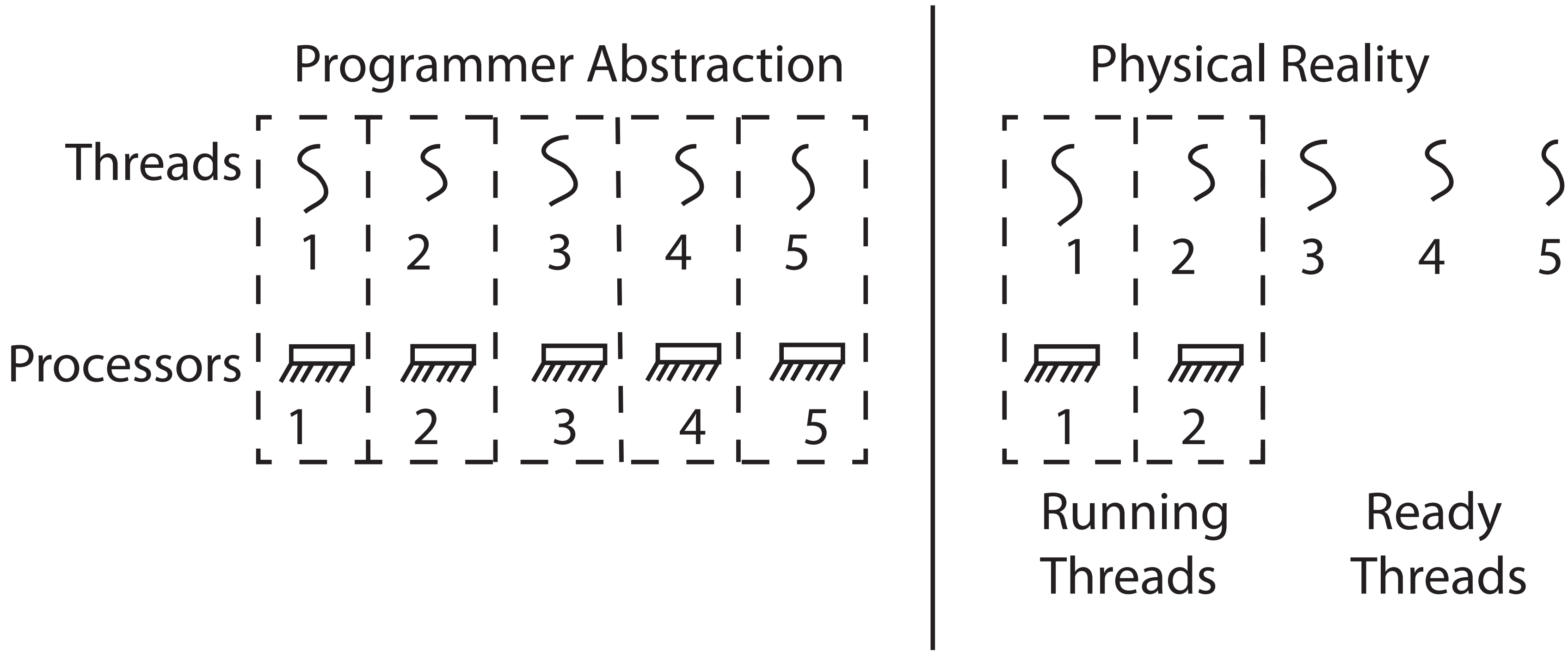
```
TimerInterrupt() {
   DoPeriodicHouseKeeping();
   run_new_thread();
}
```

Thread Abstraction



- Illusion: Infinite number of processors

Thread Abstraction



- Illusion: Infinite number of processors
- Reality: Threads execute with variable speed
  - Programs must be designed to work with any schedule

Programmer vs. Processor View

Programmer's View	Possible Execution #1
.	.
.	.
.	.
x = x + 1;	x = x + 1;
y = y + x;	y = y + x;
z = x +5y;	z = x + 5y;
.	.
.	.
.	.

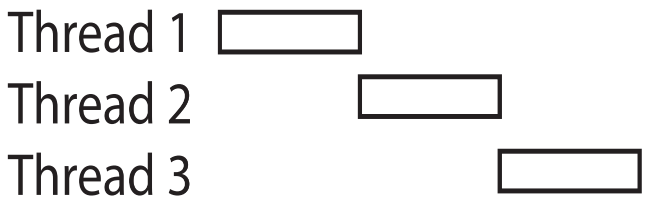
Programmer vs. Processor View

Programmer's View	Possible Execution #1	Possible Execution #2
.	.	.
.	.	.
.	.	.
x = x + 1;	x = x + 1;	x = x + 1
y = y + x;	y = y + x;	....
z = x +5y;	z = x + 5y;	thread is suspended
.	.	other thread(s) run
.	.	thread is resumed
.	.	....
		y = y + x
		z = x + 5y

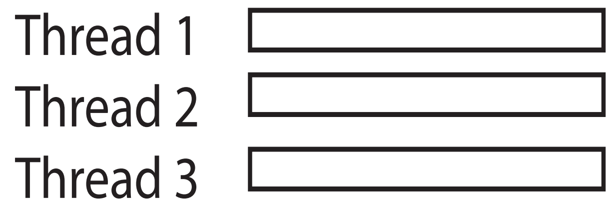
Programmer vs. Processor View

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1	x = x + 1
y = y + x;	y = y + x;	....	y = y + x
z = x +5y;	z = x + 5y;	thread is suspended	....
.	.	other thread(s) run	thread is suspended
.	.	thread is resumed	other thread(s) run
.	.	....	thread is resumed
		y = y + x	....
		z = x + 5y	z = x + 5y

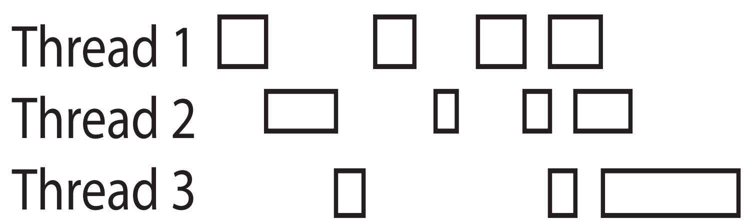
Possible Executions



a) One execution

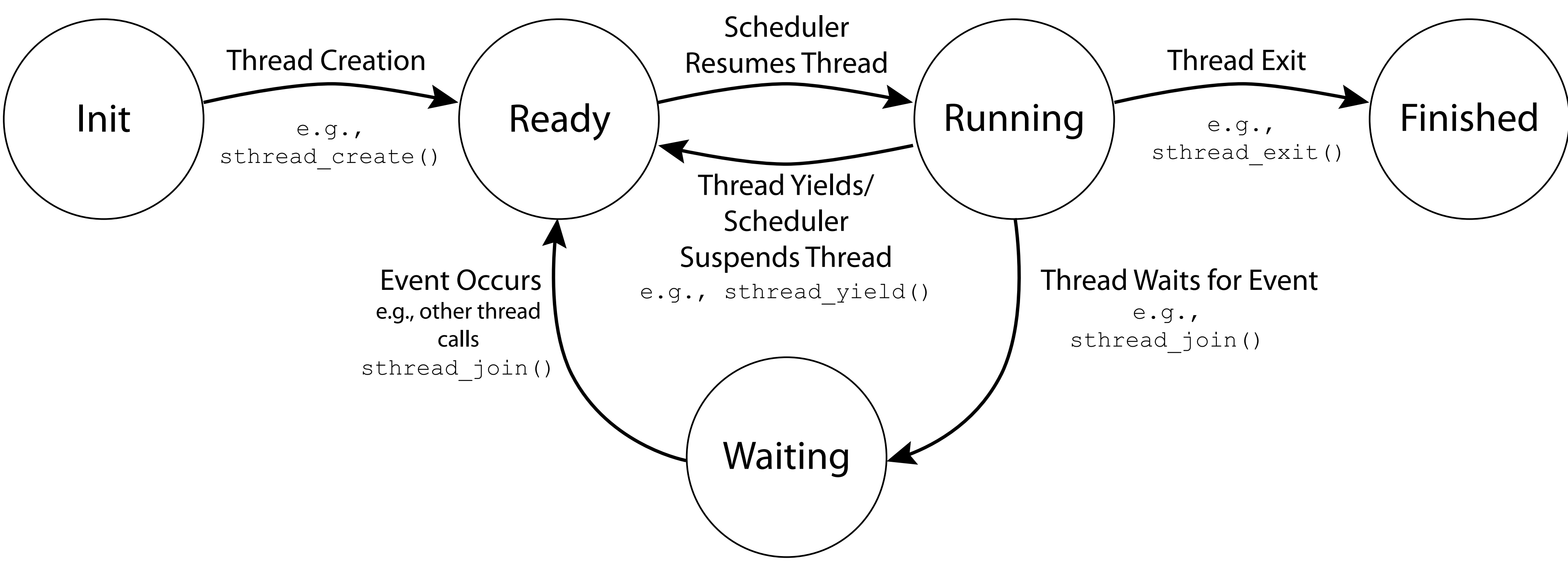


b) Another execution



c) Another execution

Thread Lifecycle



Per Thread Descriptor (Kernel Supported Threads)

- Each Thread has a *Thread Control Block* (TCB)
  - Execution State: CPU registers, program counter (PC), pointer to stack (SP)
  - Scheduling info: state, priority, CPU time
  - Various Pointers (for implementing scheduling queues)
  - Pointer to enclosing process (PCB) – user threads
  - Etc (add stuff as you find a need)

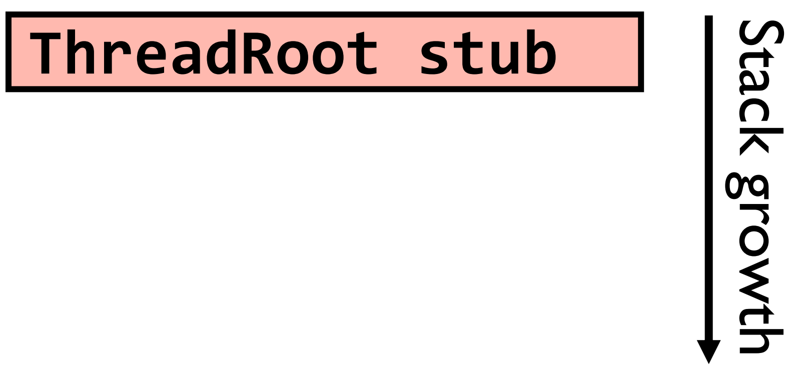
- OS Keeps track of TCBs in "kernel memory"
  - In Array, or Linked List, or …
  - I/O state (file descriptors, network connections, etc)

ThreadFork(): Create a New Thread

- `ThreadFork()` is a user-level procedure that creates a new thread and places it on ready queue
- Arguments to `ThreadFork()`
  - Pointer to application routine (`fcnPtr`)
  - Pointer to array of arguments (`fcnArgPtr`)
  - Size of stack to allocate
- Implementation
  - Sanity check arguments
  - Enter Kernel-mode and Sanity Check arguments again
  - Allocate new Stack and TCB
  - Initialize TCB and place on ready list (Runnable)

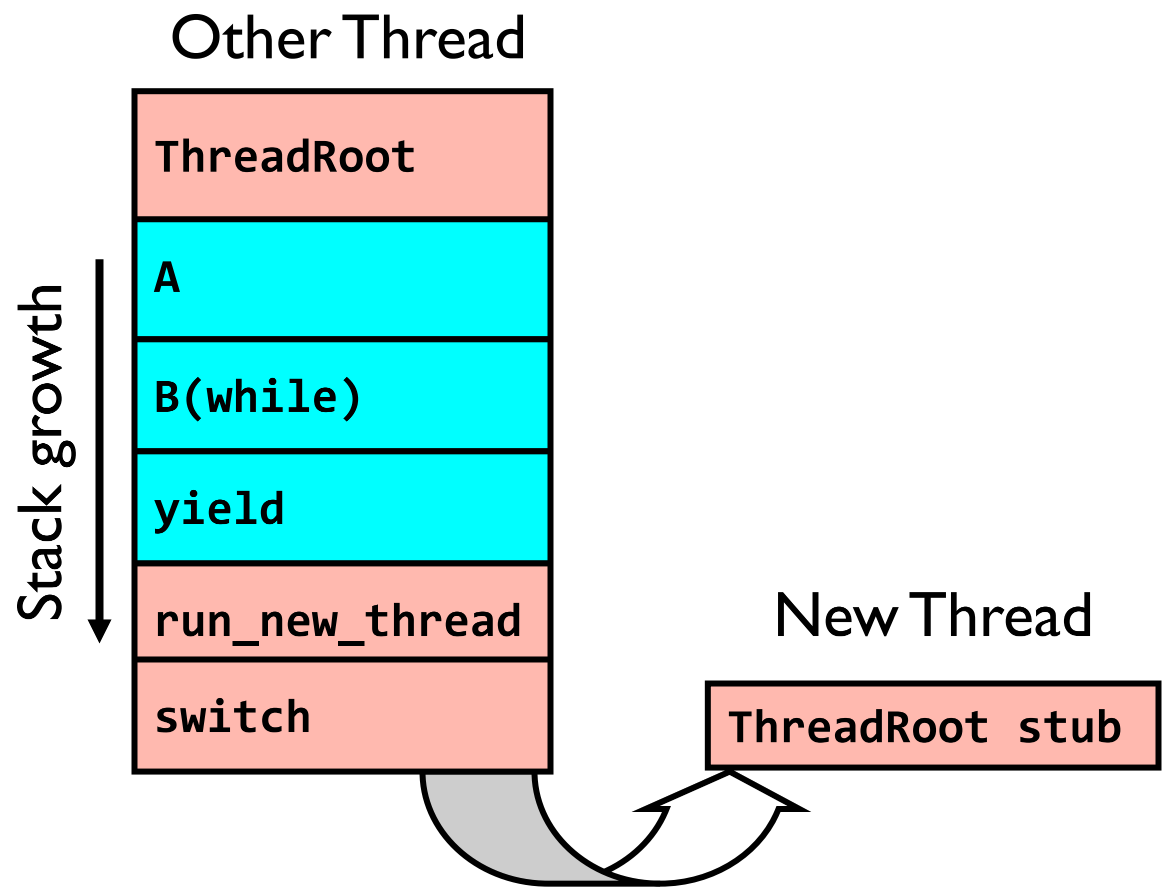
How do we initialize TCB and Stack?

- Initialize Register fields of TCB
  - Stack pointer made to point at stack
  - PC return address ⇒ OS (asm) routine `ThreadRoot()`
  - Two arg registers (a0 and a1) initialized to `fcnPtr` and `fcnArgPtr`, respectively
- Initialize stack data?
  - No. Important part of stack frame is in registers (ra)
  - Think of stack frame as just before body of `ThreadRoot()` really gets started



Initial Stack

How does Thread get started?

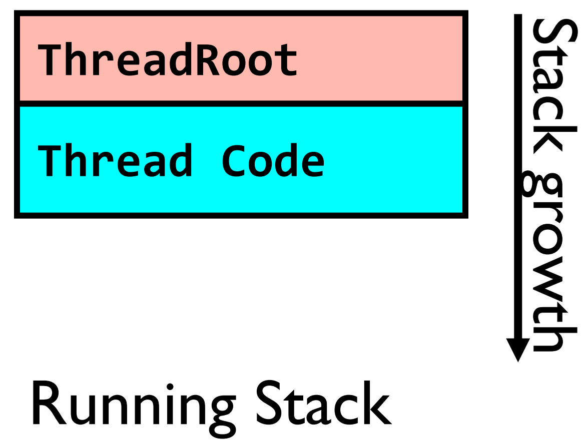


- Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`
  - This really starts the new thread

What does ThreadRoot() look like?

- `ThreadRoot()` is the root for the thread routine:

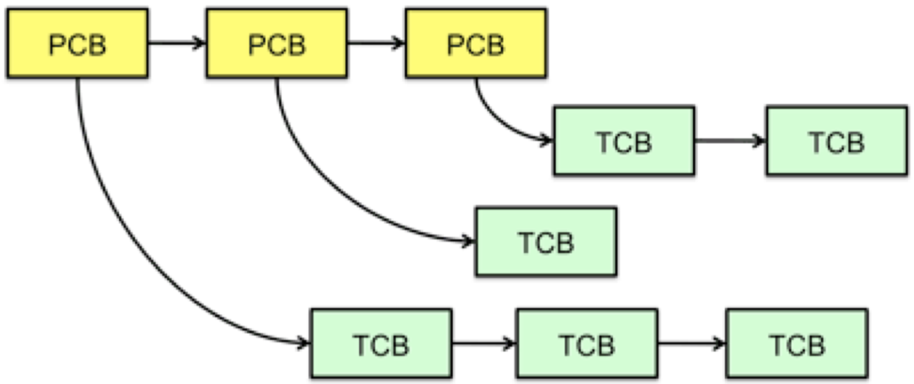
```
ThreadRoot() {
    DoStartupHousekeeping();
    UserModeSwitch(); /* enter user mode */
    Call fcnPtr(fcnArgPtr);
    ThreadFinish();
}
```



- Startup Housekeeping
  - Includes things like recording start time of thread
  - Other statistics
- Stack will grow and shrink with execution of thread
- Final return from thread returns into `ThreadRoot()` which calls `ThreadFinish()`
  - `ThreadFinish()` wake up sleeping threads

Multithreaded Processes

- Process Control Block (PCBs) points to multiple Thread Control Blocks (TCBs):



- Switching threads within a block is a simple thread switch
- Switching threads across blocks requires changes to memory and I/O address tables

Examples multithreaded programs

- Embedded systems
  - Elevators, planes, medical systems, smart watches
  - Single program, concurrent operations
- Most modern OS kernels
  - 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

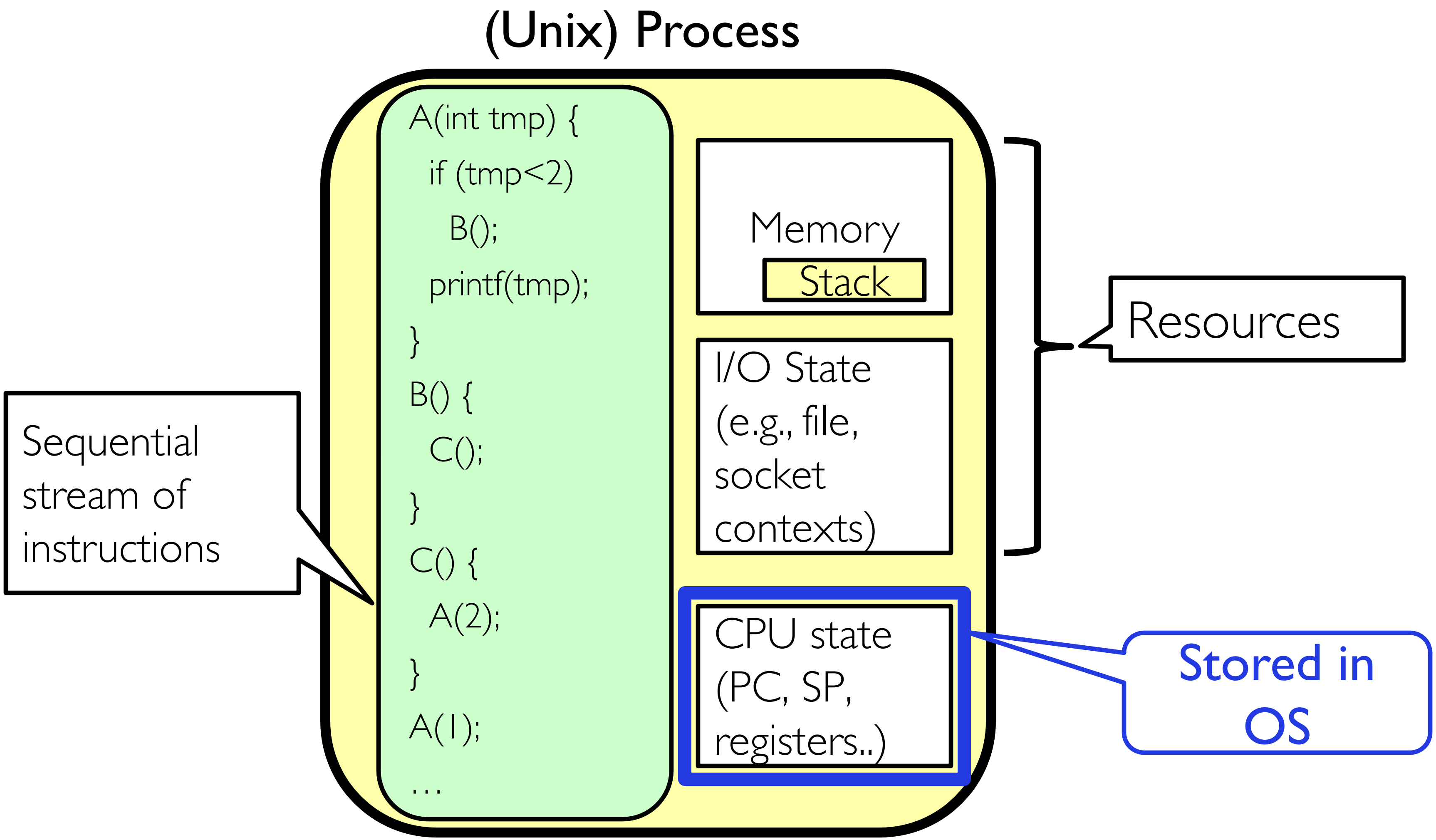
Example multithreaded programs (con't)

- Network servers
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations
  - File server, Web server, and airline reservation systems

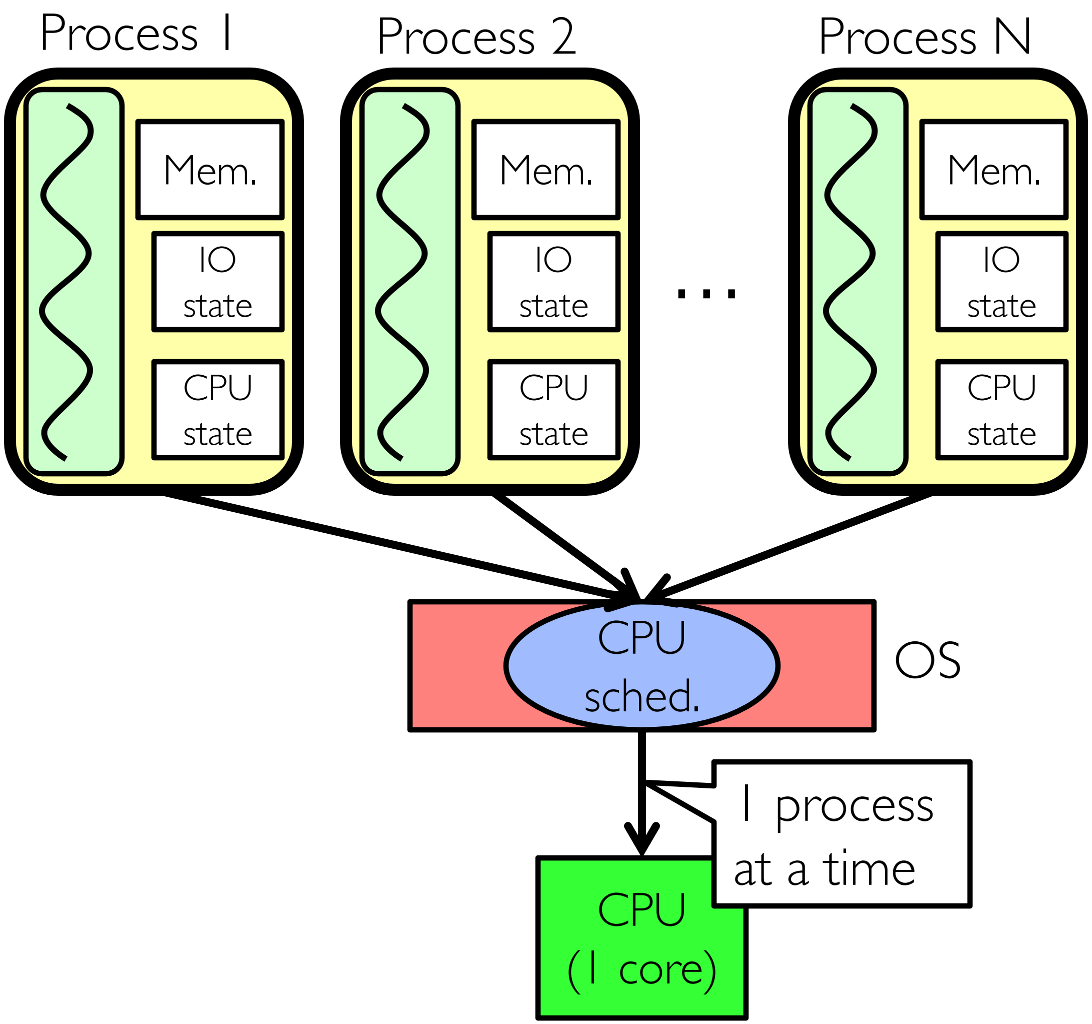
- Parallel programming (more than one physical CPU)
  - Split program into multiple threads for parallelism
  - This is called Multiprocessing

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

Putting it Together: Process

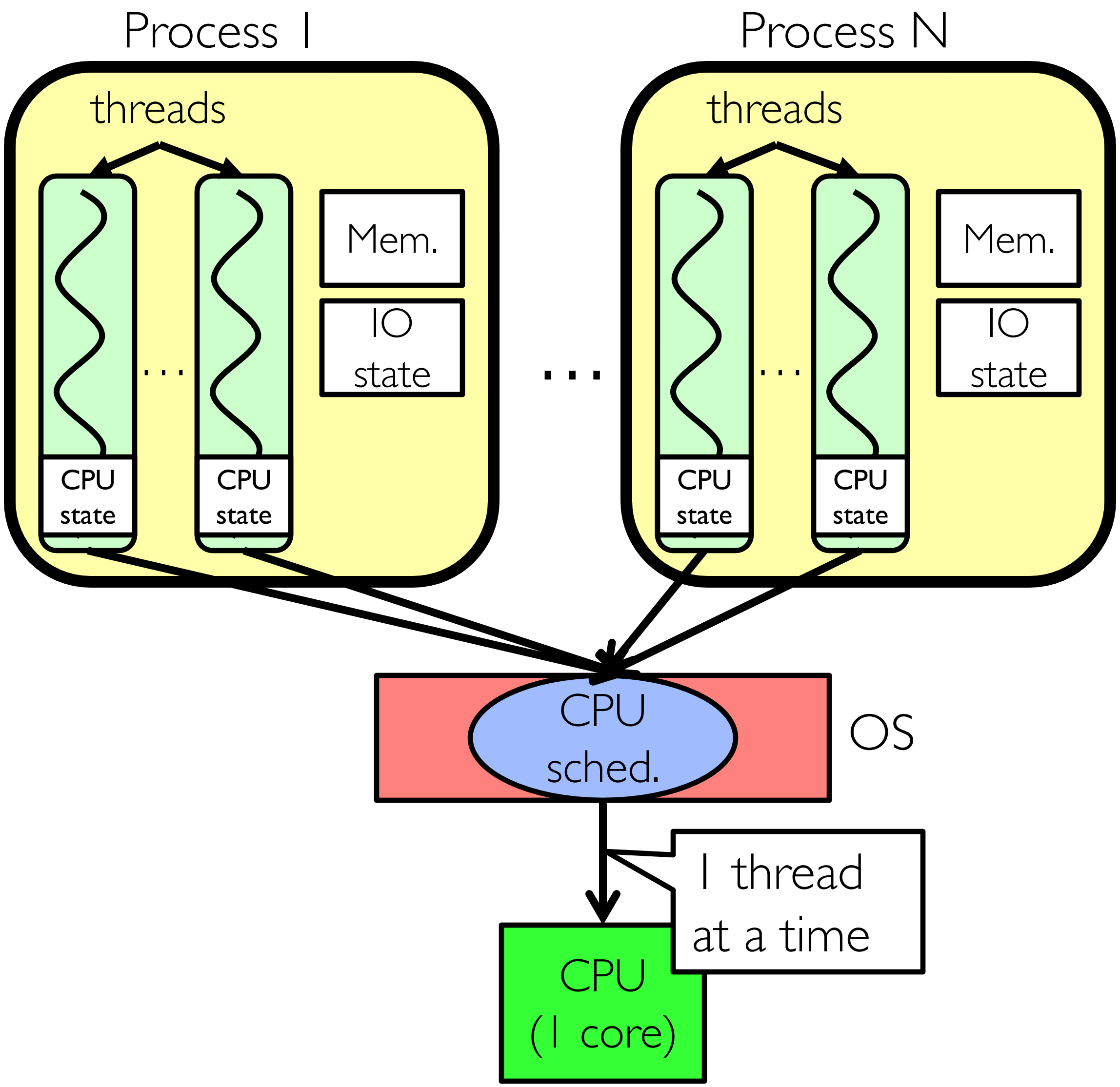


Putting it Together: Processes



- Switch overhead: high
  - CPU state: *low*
  - Memory/IO state: high
- Process creation: high
- Protection
  - CPU: *yes*
  - Memory/IO: *yes*
- Sharing overhead: high (involves at least a context switch)

Putting it Together: Threads



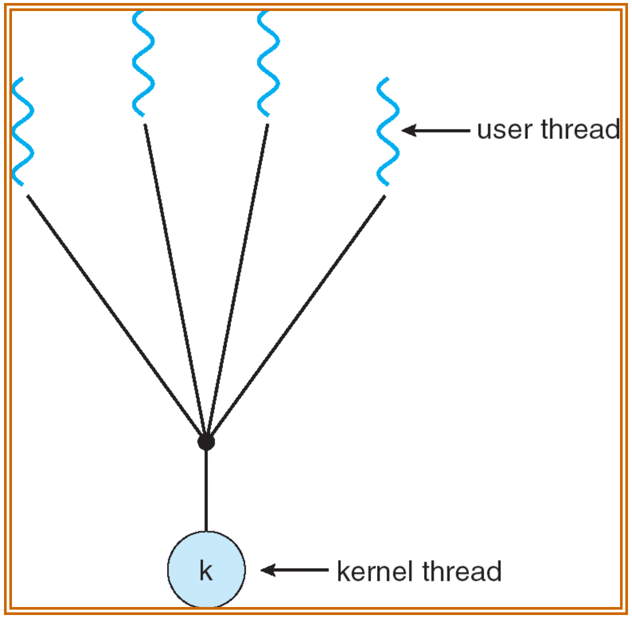
- Switch overhead: medium
  - CPU state: *low*
- Thread creation: medium
- Protection
  - CPU: *yes*
  - Memory/IO: No
- Sharing overhead: *low(ish)* (thread switch overhead low)

Kernel versus User-Mode Threads

- We have been talking about kernel threads
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things
- Downside of kernel threads: a bit expensive
  - Need to make a crossing into kernel mode to schedule
- Lighter weight option: User Threads

User-Mode Threads

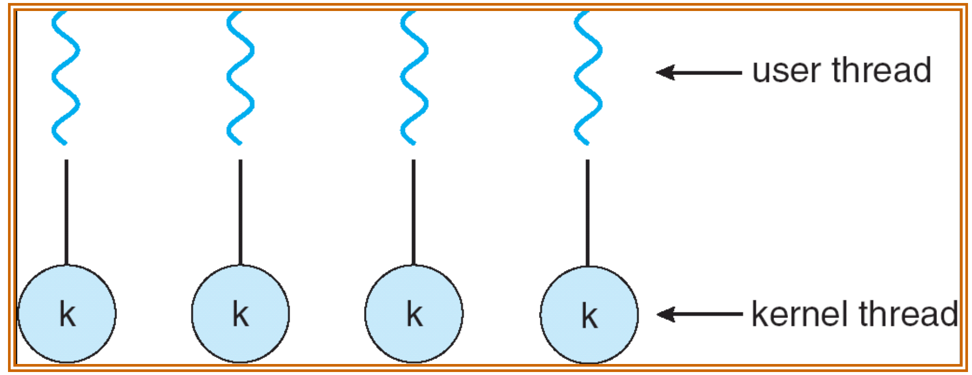
- Lighter weight option:
  - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-preemptively relative to each other (only switch on yield())
  - Cheap

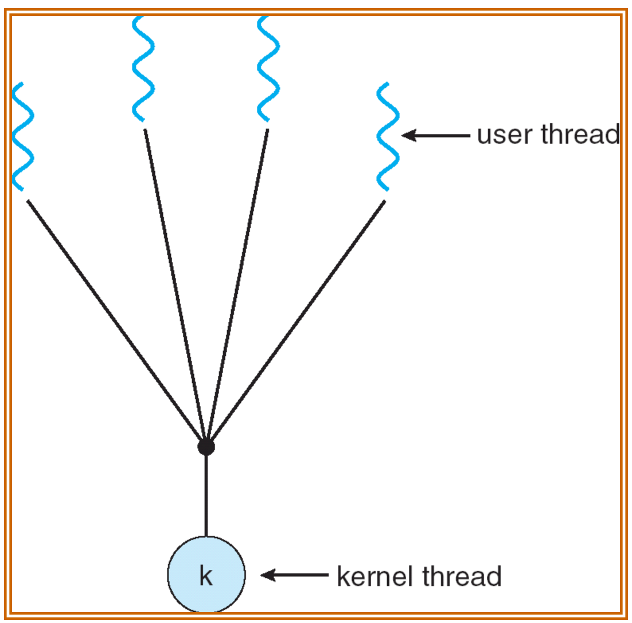


- Downside of user threads:
  - When one thread blocks on I/O, all threads block
  - Kernel cannot adjust scheduling among all threads
  - Option: *Scheduler Activations*
    - » Have kernel inform user level when thread blocks…

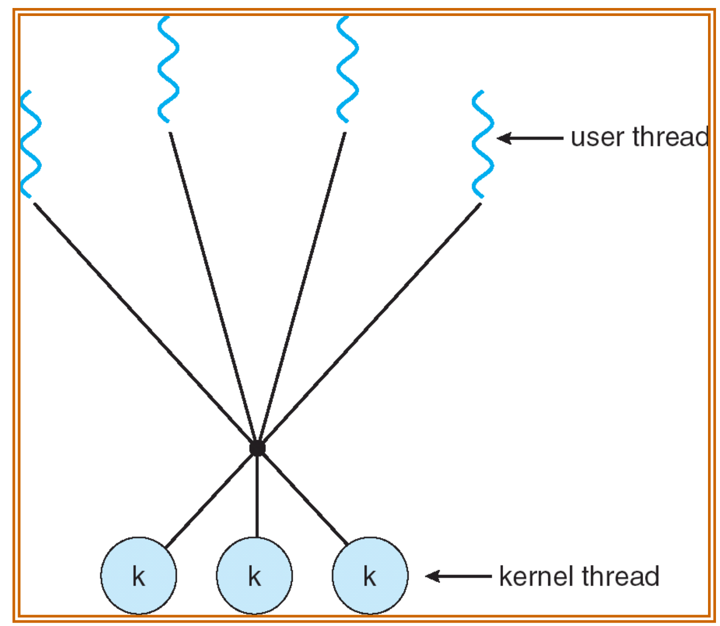
Some Threading Models

Simple One-to-One Threading Model





Many-to-One

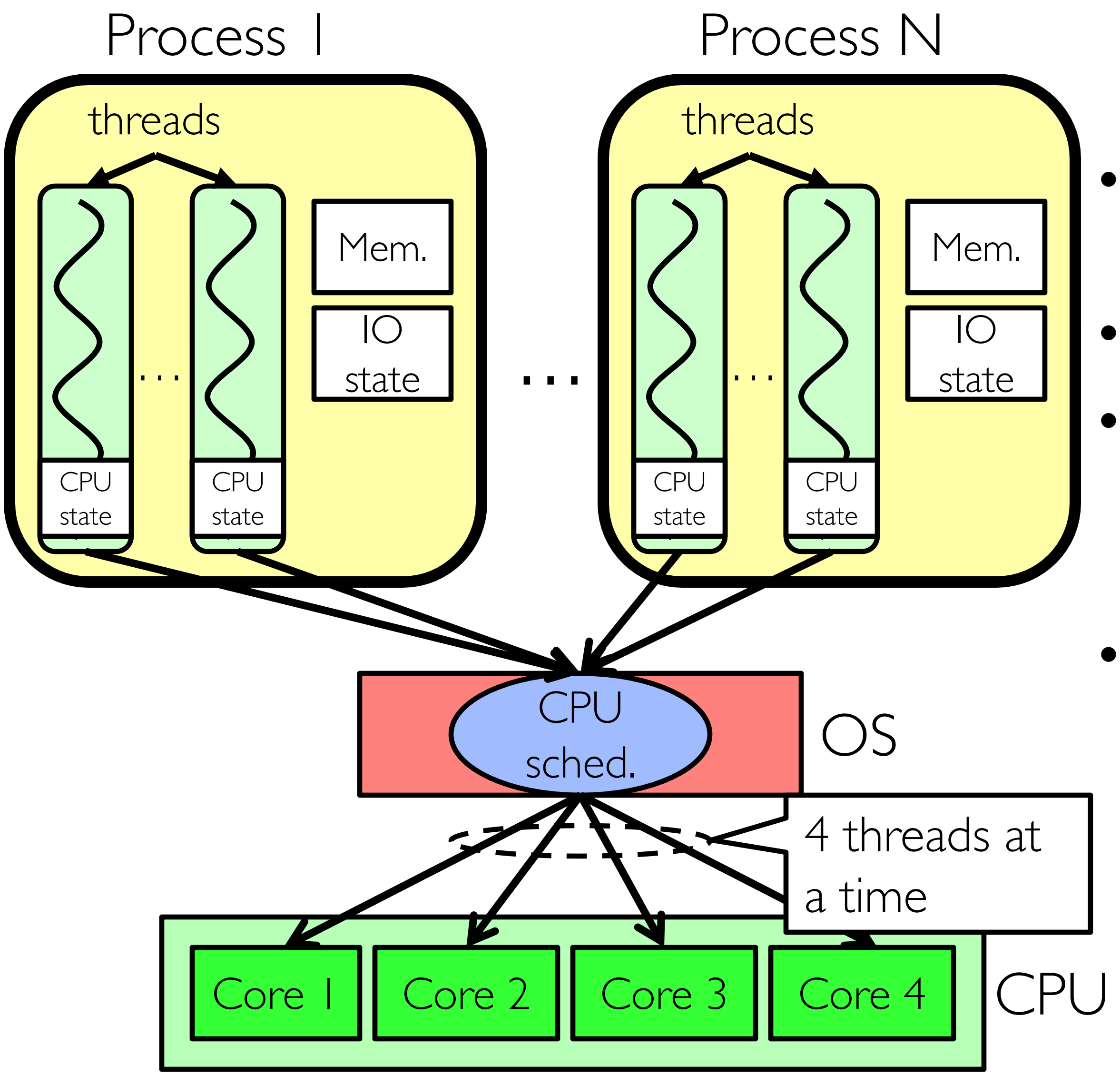


Many-to-Many

Threads in a Process

- Threads are useful at user-level: parallelism, hide I/O latency, interactivity
- Option A (early Java): user-level library, within a single-threaded process
  - Library does thread context switch
  - Kernel time slices between processes, e.g., on system call I/O
- Option B (SunOS, Linux/Unix variants): green threads
  - User-level library does thread multiplexing
- Option C (Windows): scheduler activations
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - System call I/O that blocks triggers upcall
- Option D (Linux, MacOS, Windows): use kernel threads
  - System calls for thread fork, join, exit (and lock, unlock,…)
  - Kernel does context switching
  - Simple, but a lot of transitions between user and kernel mode

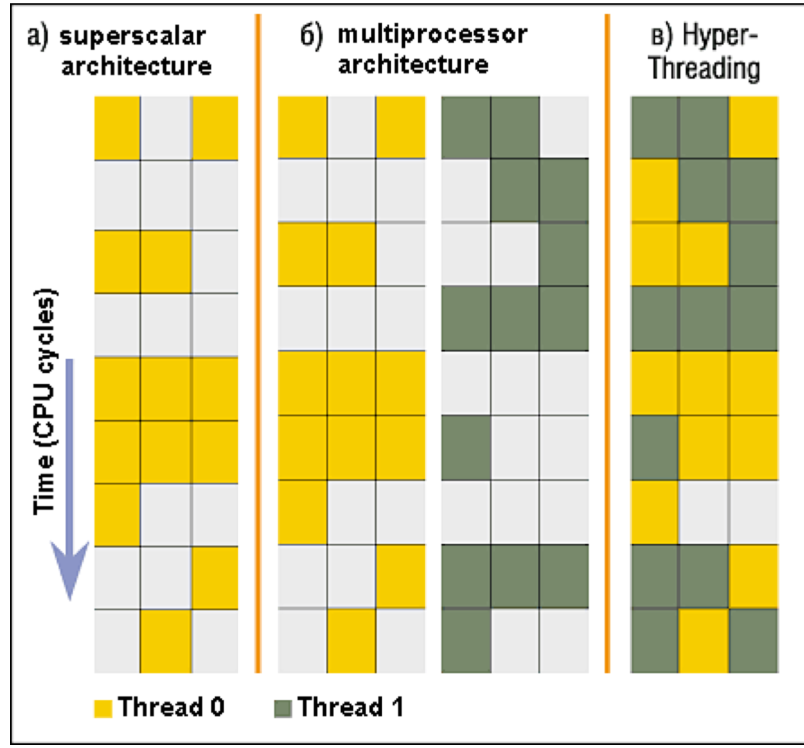
Putting it Together: Multi-Cores



- Switch overhead: *low* (only CPU state)
- Thread creation: *low*
- Protection
  - CPU: *yes*
  - Memory/IO: No
- Sharing overhead: *low* (thread switch overhead low, may not need to switch at all!)

Simultaneous MultiThreading/Hyperthreading

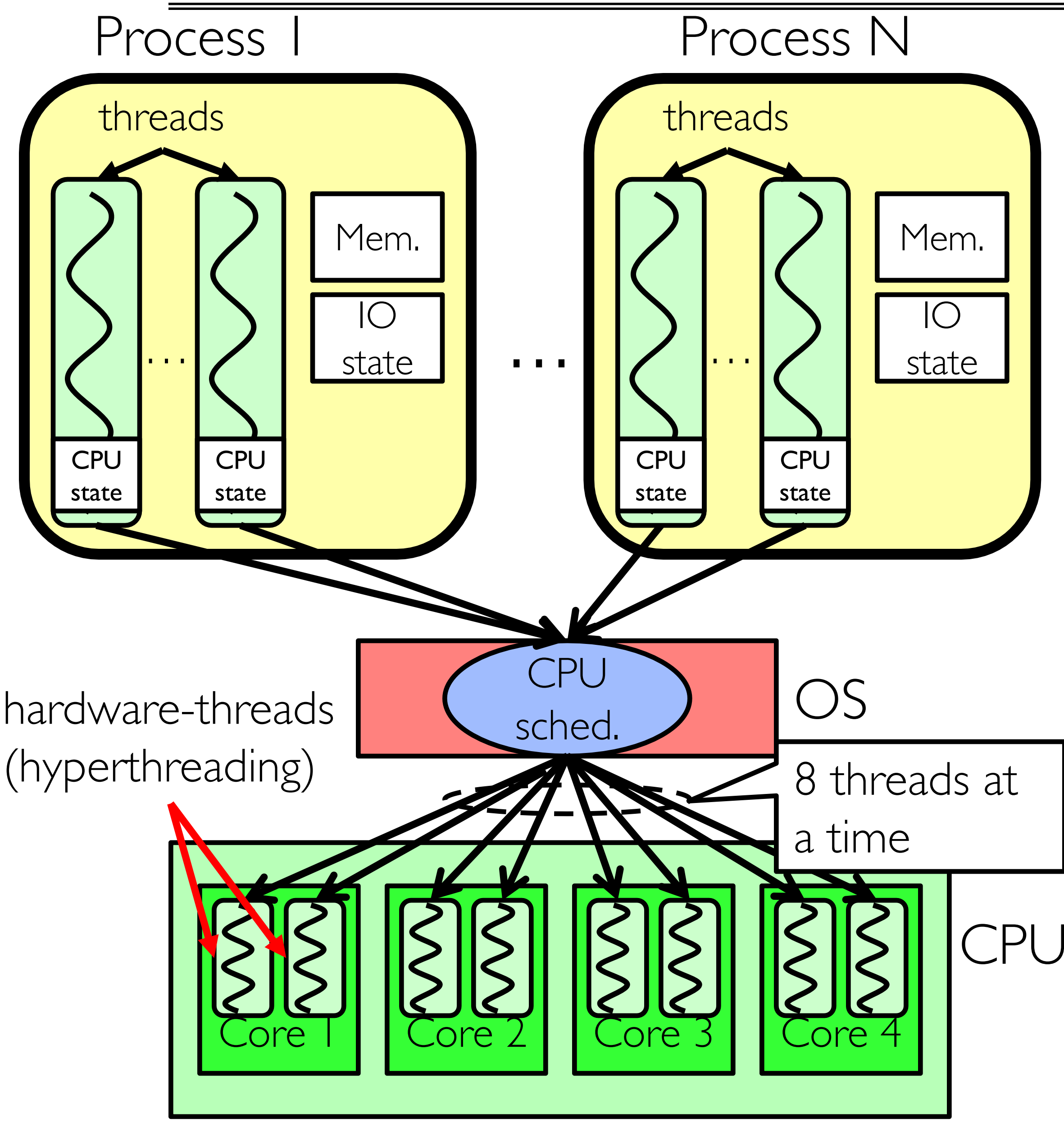
- Hardware technique
  - Superscalar processors can execute multiple instructions that are independent
  - Hyperthreading duplicates register state to make a second "thread," allowing more instructions to run
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!



Colored blocks show instructions executed

- Original technique called "Simultaneous Multithreading"
  - http://www.cs.washington.edu/research/smt/index.html
  - SPARC, Pentium 4/Xeon ("Hyperthreading"), Power 5

Putting it Together: Hyper-Threading



- Switch overhead between hardware-threads: *very-low* (done in hardware)
- Contention for ALUs/FPUs may hurt performance

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 10 Win NT to XP, Solaris, HP-UX, OS X

- Most operating systems have either
  - One or many address spaces
  - One or many threads per address space

Summary

- Processes have two parts
  - Threads (Concurrency)
  - Address Spaces (Protection)
- Various textbooks talk 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
- Concurrent threads are a very useful abstraction
  - Allow transparent overlapping of computation and I/O
  - Allow use of parallel processing when available
- Concurrent threads introduce problems when accessing shared data
  - Programs must be insensitive to arbitrary interleavings
  - Without careful design, shared variables can become completely inconsistent