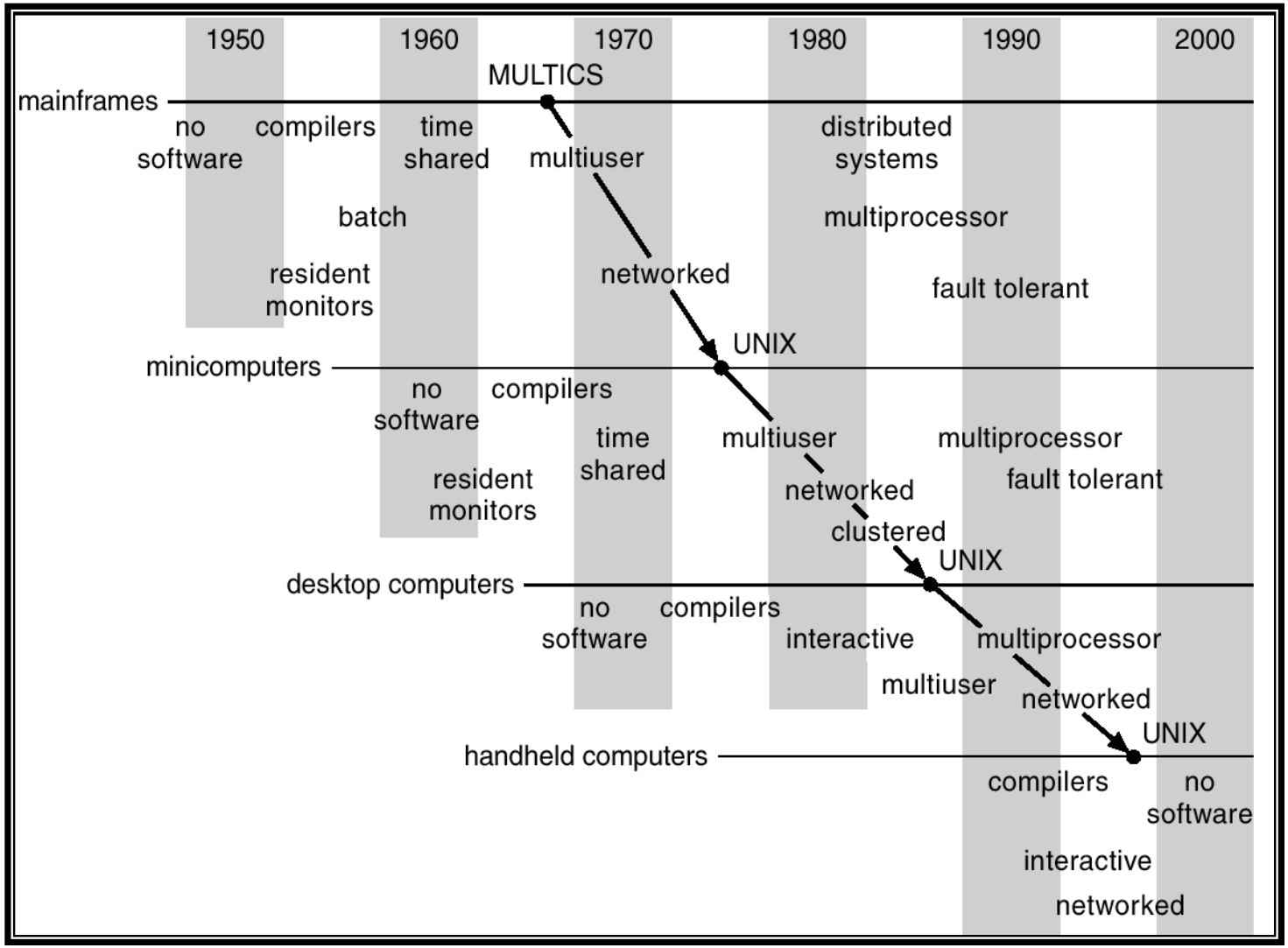
Concurrency:
Processes, Threads, and Address Spaces

Adapted by Tiziano Villa from lectures notes by
Prof. John Kubiatowicz (UC Berkeley)

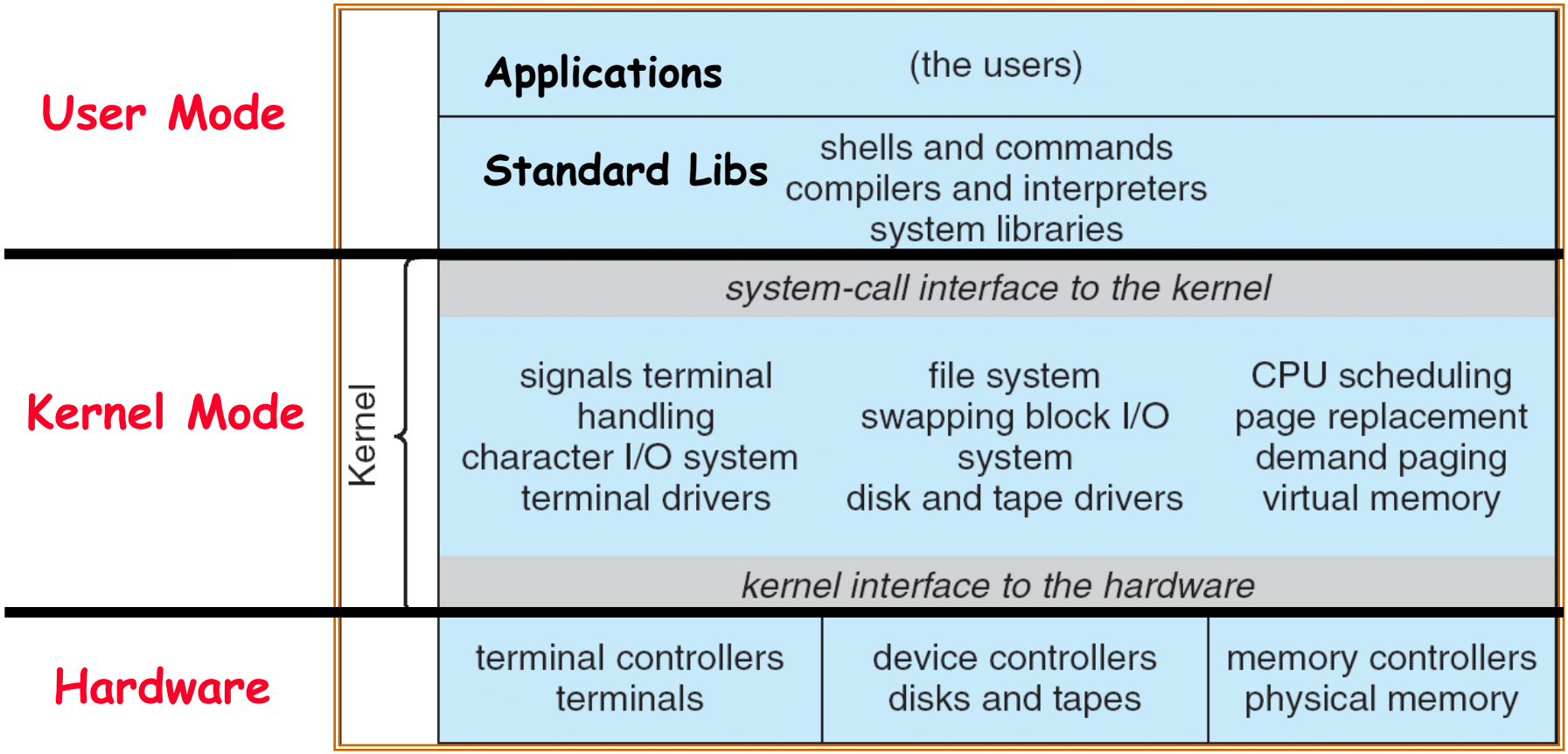
Review: History of OS

- 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: UNIX System Structure



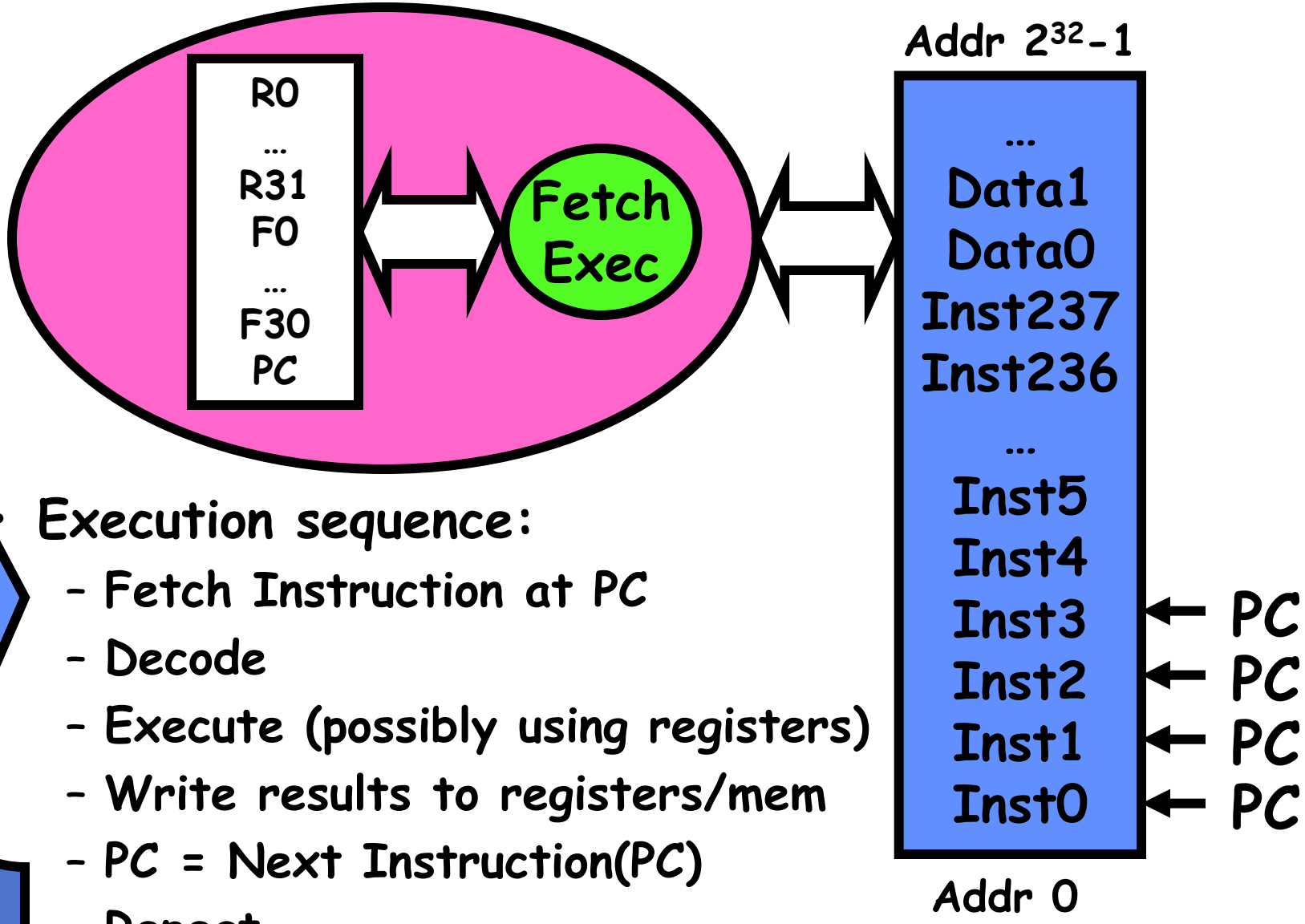
Concurrency

- "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*
  - Multics, UNIX/Linux, OS/2, Windows NT/2000/XP, Mac OS X
  - Often called "multitasking", but multitasking has other meanings (talk about this later)
- ManyCore $\Rightarrow$ Multiprogramming, right?

The Basic Problem of Concurrency

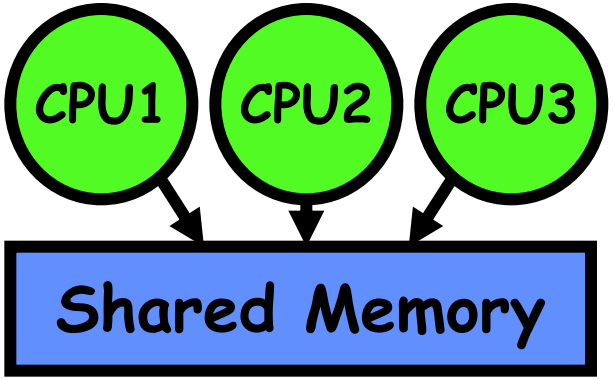
- The basic problem of concurrency involves resources:
  - Hardware: single CPU, single DRAM, single I/O devices
  - Multiprogramming API: users think they have exclusive access to shared resources
- OS Has to coordinate all activity
  - Multiple users, I/O interrupts, …
  - How can it keep all these things 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
- Dijkstra did this for the "THE system"
  - Few thousand lines vs 1 million lines in OS 360 (1K bugs)

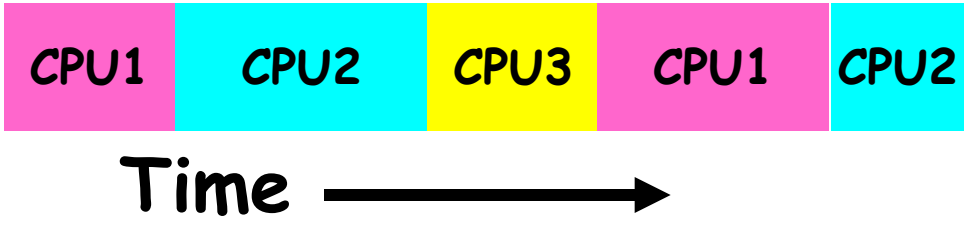
What happens during execution?



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

How can we give the illusion of multiple processors?





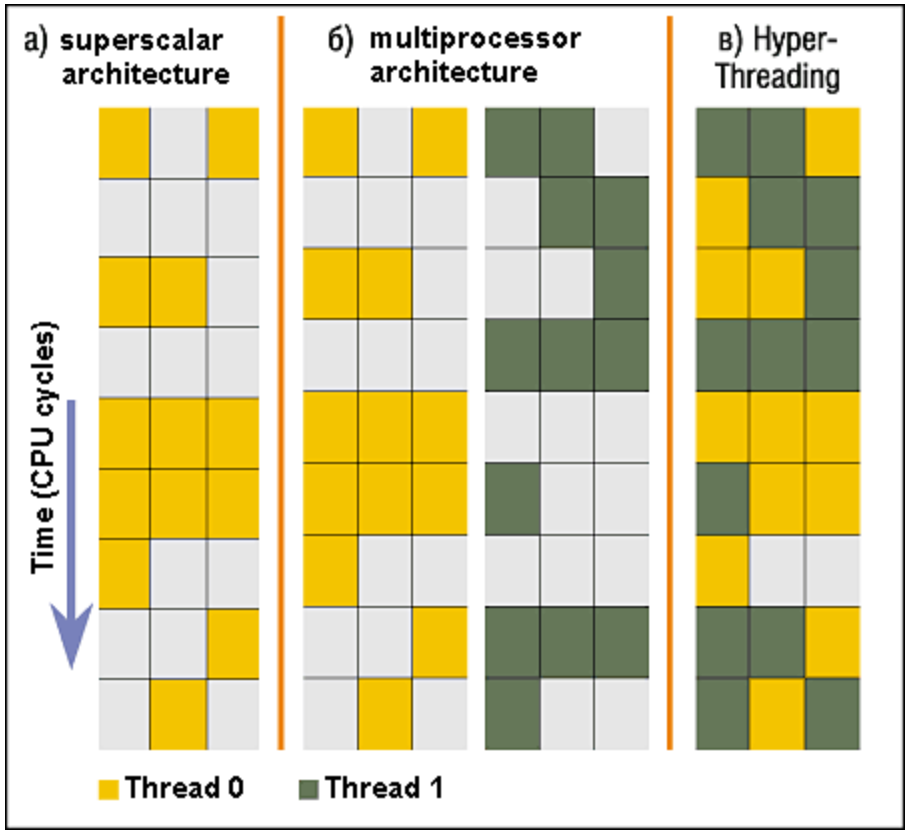
- Assume a single processor. How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Each virtual "CPU" needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How switch from one CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, 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



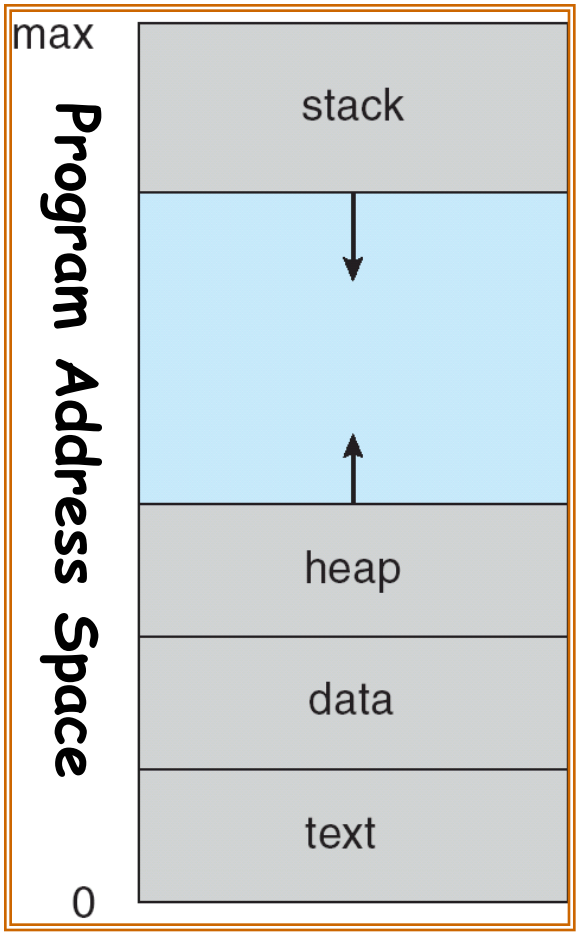
- Original technique called "Simultaneous Multithreading"
  - See http://www.cs.washington.edu/research/smt/
  - Alpha, SPARC, Pentium 4 ("Hyperthreading"), Power 5

How to protect threads from one another?

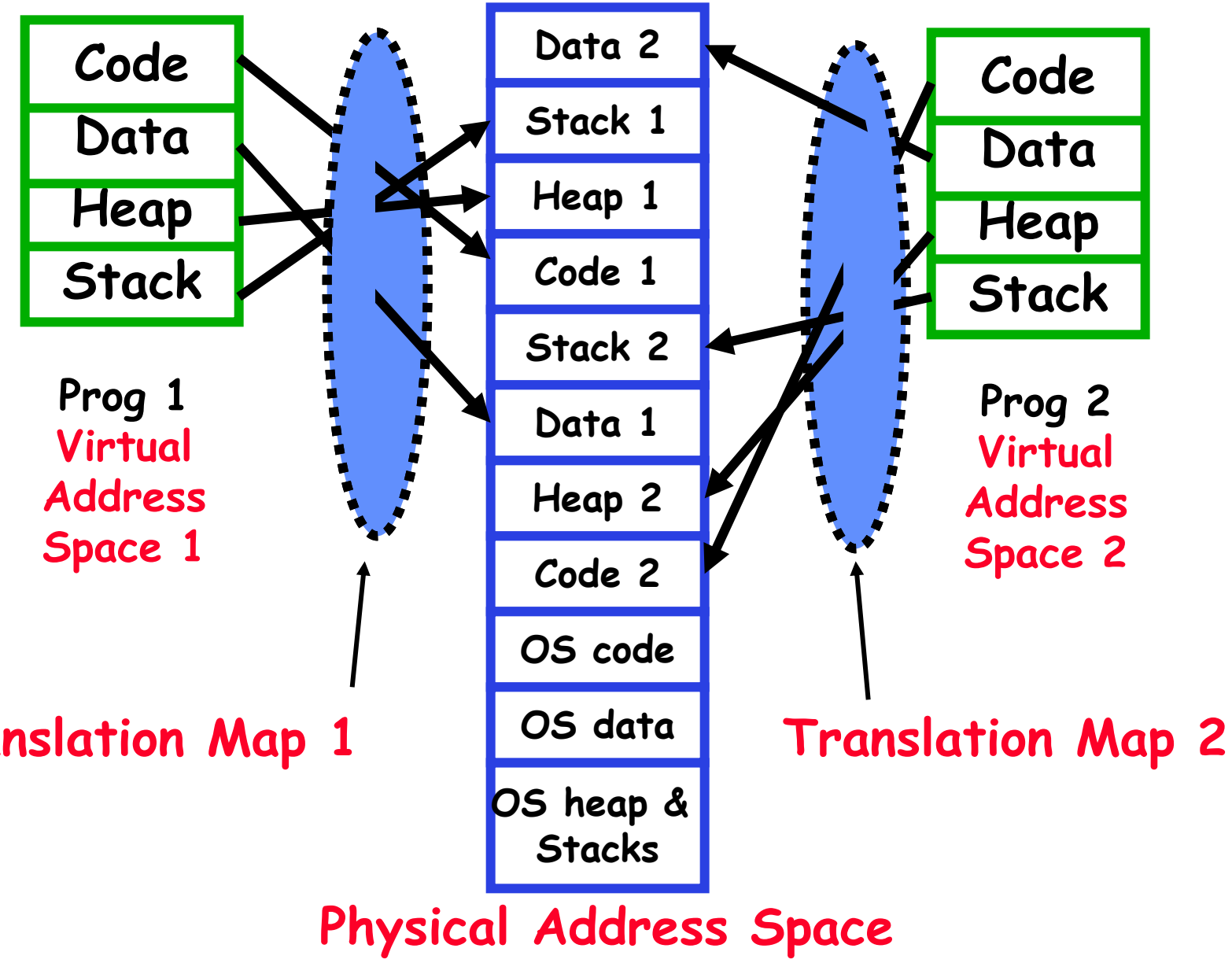
- Need three important things:
  1. Protection of memory
     - » Every task does not have access to all memory
  2. Protection of I/O devices
     - » Every task does not have access to every device
  3. Protection of Access to Processor: Preemptive switching from task to task
     - » Use of timer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

Recall: Program's Address Space

- Address space ⇒ 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

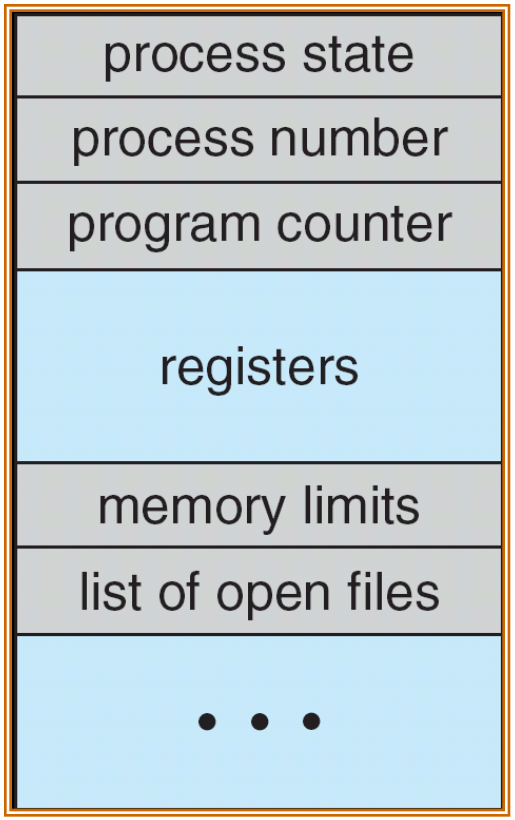


Traditional UNIX Process

- Process: *Operating system abstraction to represent what is needed to run a single program*
  - Often called a "HeavyWeight Process"
  - Formally: a single, 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

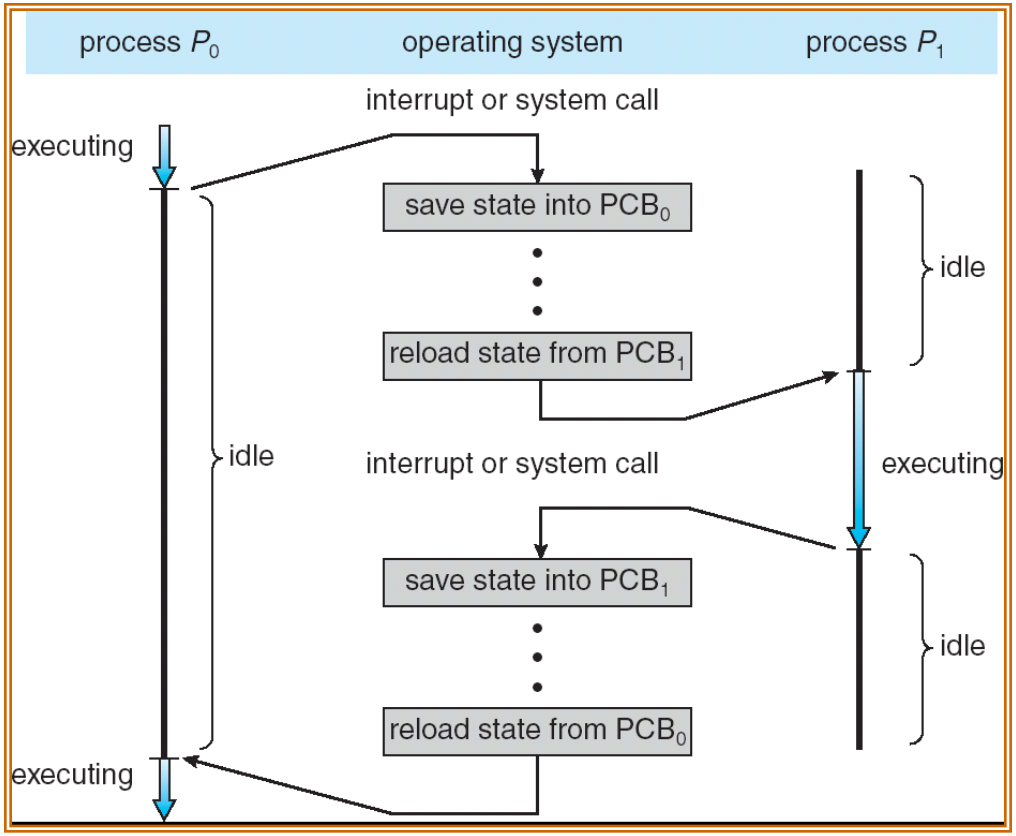
How do we multiplex processes?

- 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



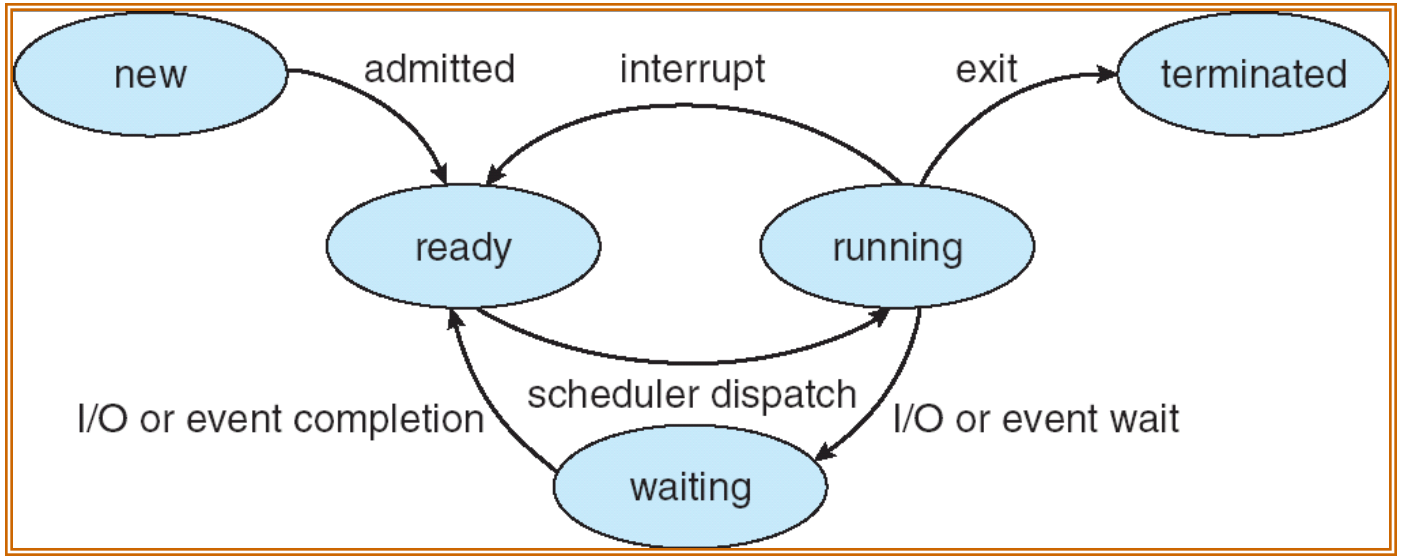
Process Control Block

CPU Switch From Process to Process



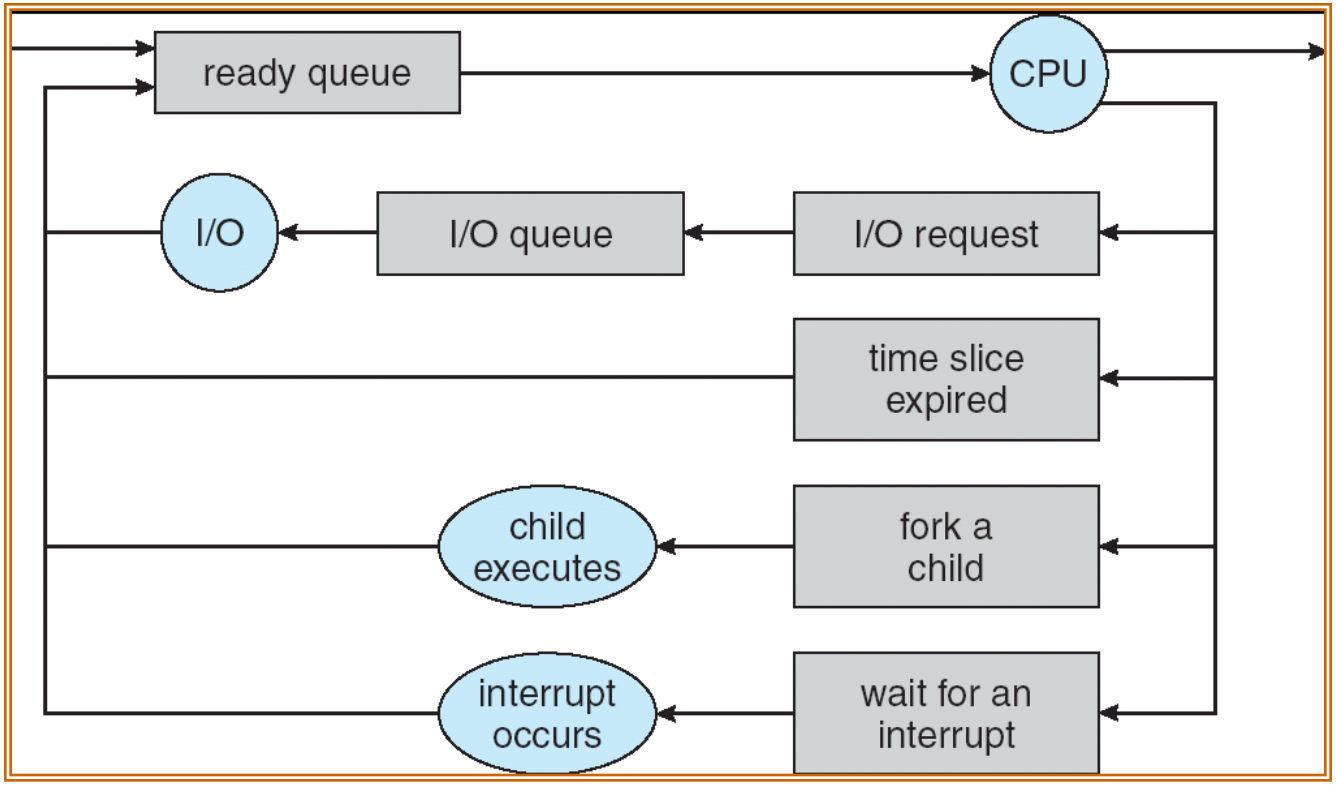
- This is also called a "context switch"
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/hyperthreading, but… contention for resources instead

Diagram of Process State



- As a process executes, it changes *state*
  - new: The process is being created
  - 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

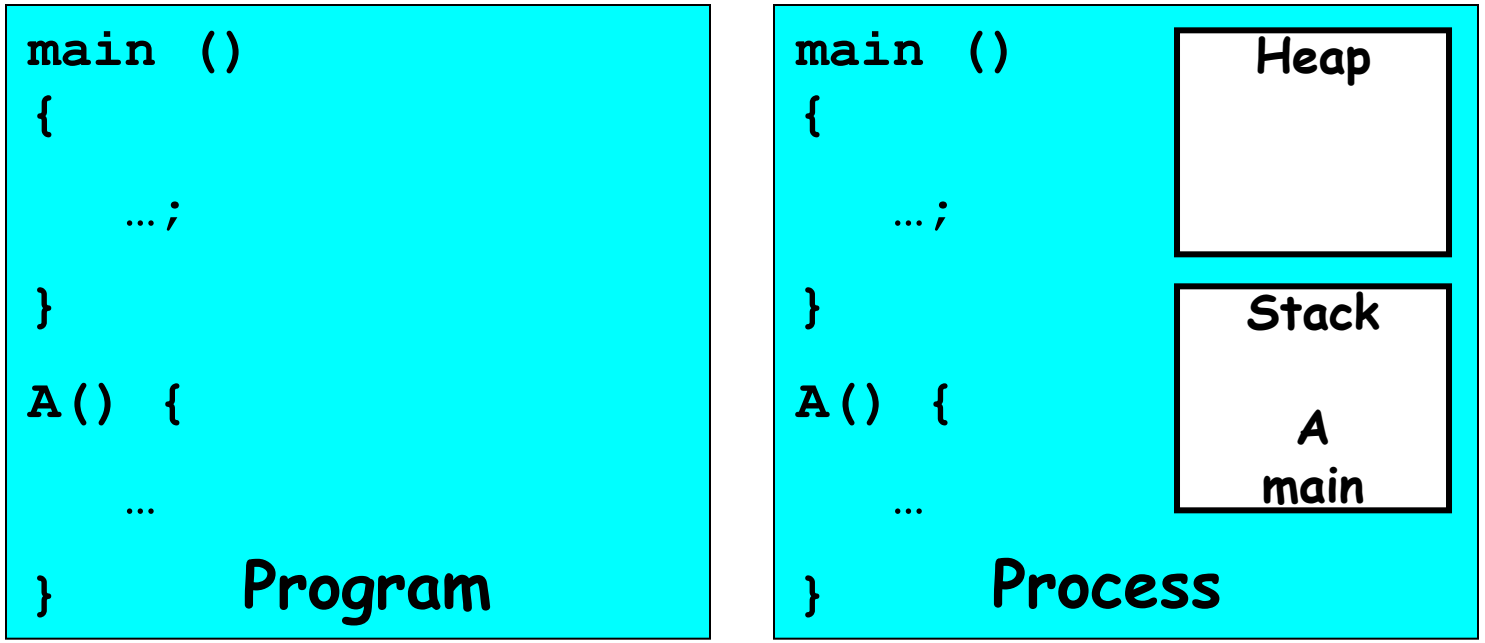


- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many algorithms possible (few weeks from now)

What does it take to create a process?

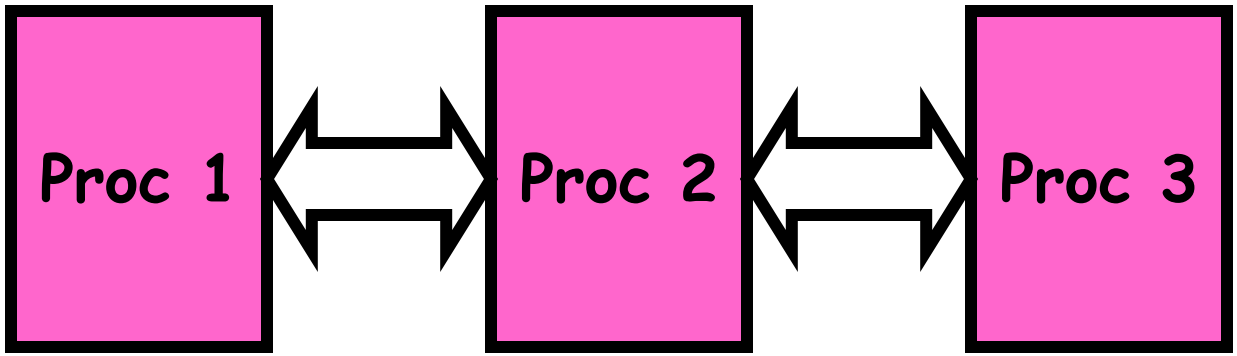
- Must construct new PCB
  - Inexpensive
- Must set up new page tables for address space
  - More expensive
- Copy data from parent process? (Unix `fork()`)
  - Semantics of Unix `fork()` are that the child process gets a complete copy of the parent memory and I/O state
  - Originally *very* expensive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

Process =? Program



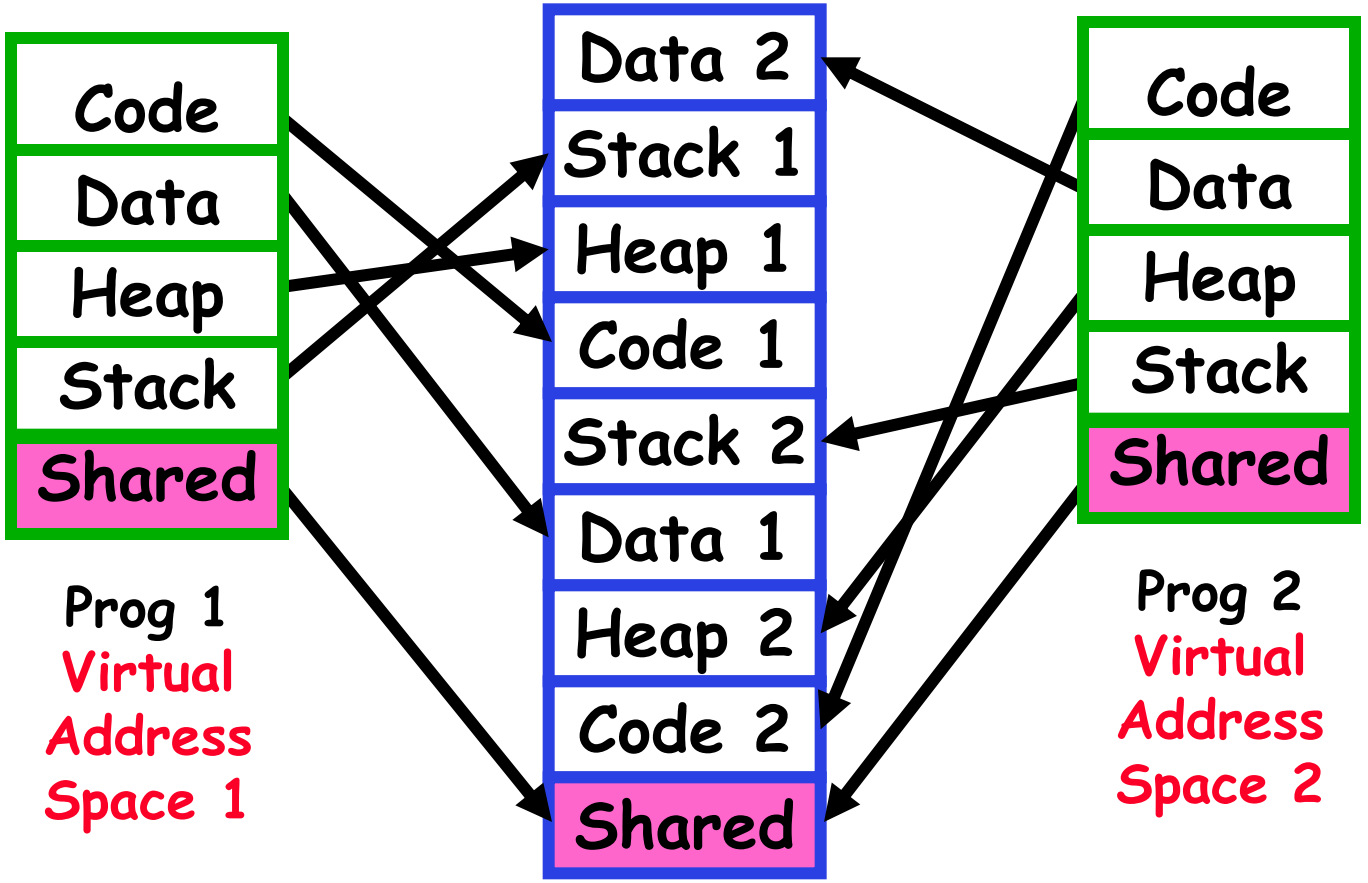
- More to a process than just a program:
  - Program is just part of the process state
  - I run emacs on lectures.txt, you run it on homework.java – Same program, different processes
- Less to a process than a program:
  - A program can invoke more than one process
  - cc starts up cpp, cc1, cc2, as, and ld

Multiple Processes Collaborate on a 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



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

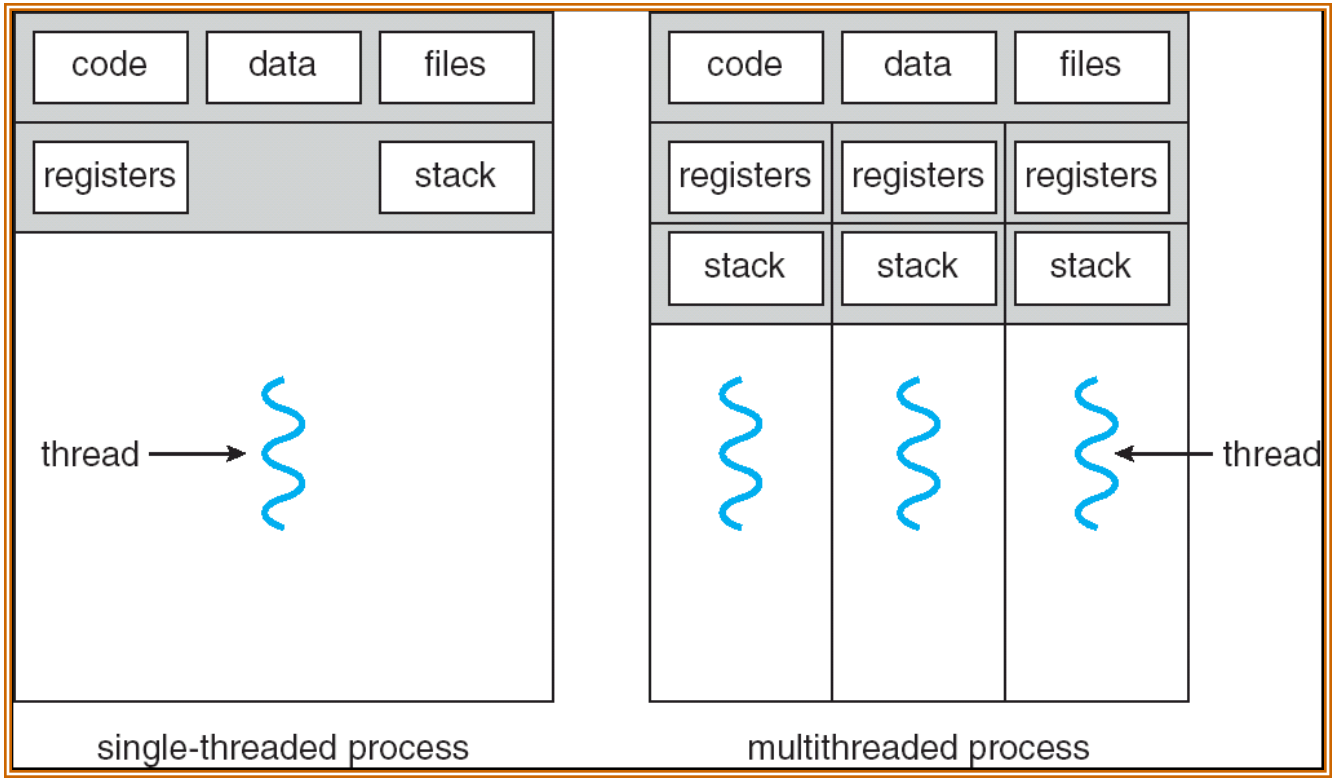
Inter-process Communication (IPC)

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

Examples of multithreaded programs

- Embedded systems
  - Elevators, Planes, Medical systems, Wristwatches
  - 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

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

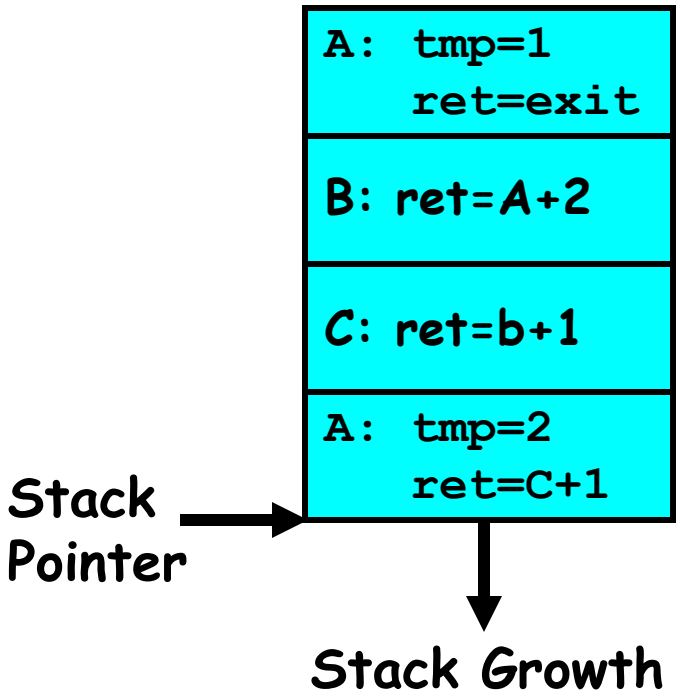
Thread State

- 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

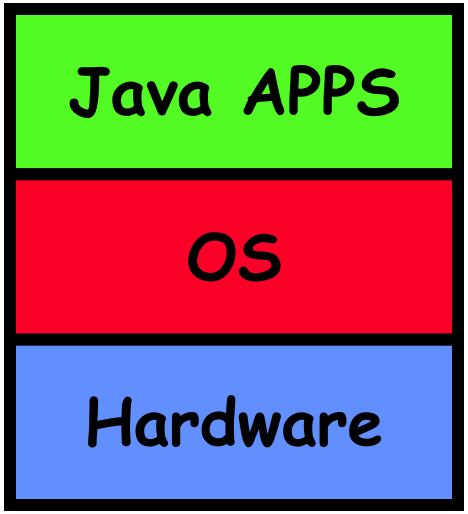
# threads Per AS: / # of addr spaces:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (Geoworks, VxWorks, JavaOS,etc) JavaOS, Pilot(PC)	Mach, OS/2, Linux Windows 9x??? Win NT to XP, Solaris, HP-UX, OS X

- 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 we 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



Recall: Modern Process with Multiple Threads

- Process: *Operating system abstraction to represent what is needed to run a single, multithreaded program*
- Two parts:
  - Multiple Threads
    - » Each thread is a *single, sequential stream of execution*
  - Protected Resources:
    - » Main Memory State (contents of Address Space)
    - » I/O state (i.e. file descriptors)
- 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

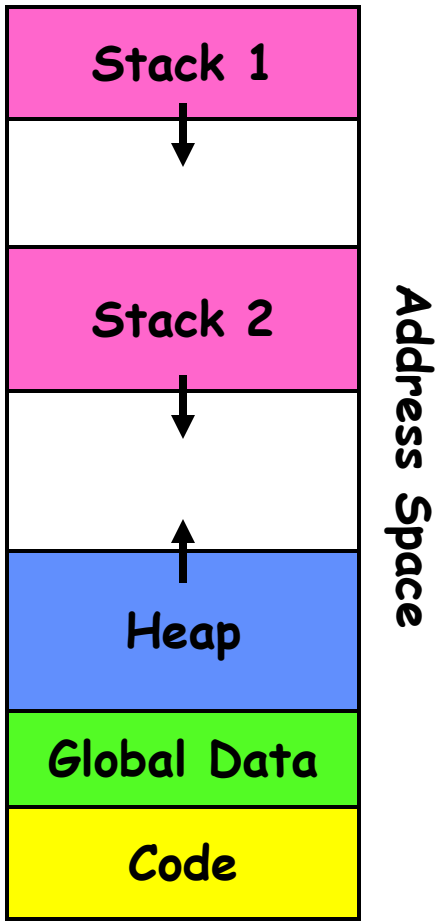
MIPS: Software conventions for Registers

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	
16	s0	callee saves
. . .		(callee must save)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	frame pointer
31	ra	Return Address (HW)

- Before calling procedure:
  - Save caller-saves regs
  - Save v0, v1
  - Save ra
- After return, assume
  - Callee-saves reg OK
  - gp,sp,fp OK (restored!)
  - Other things trashed

Memory Footprint of Two-Thread Example

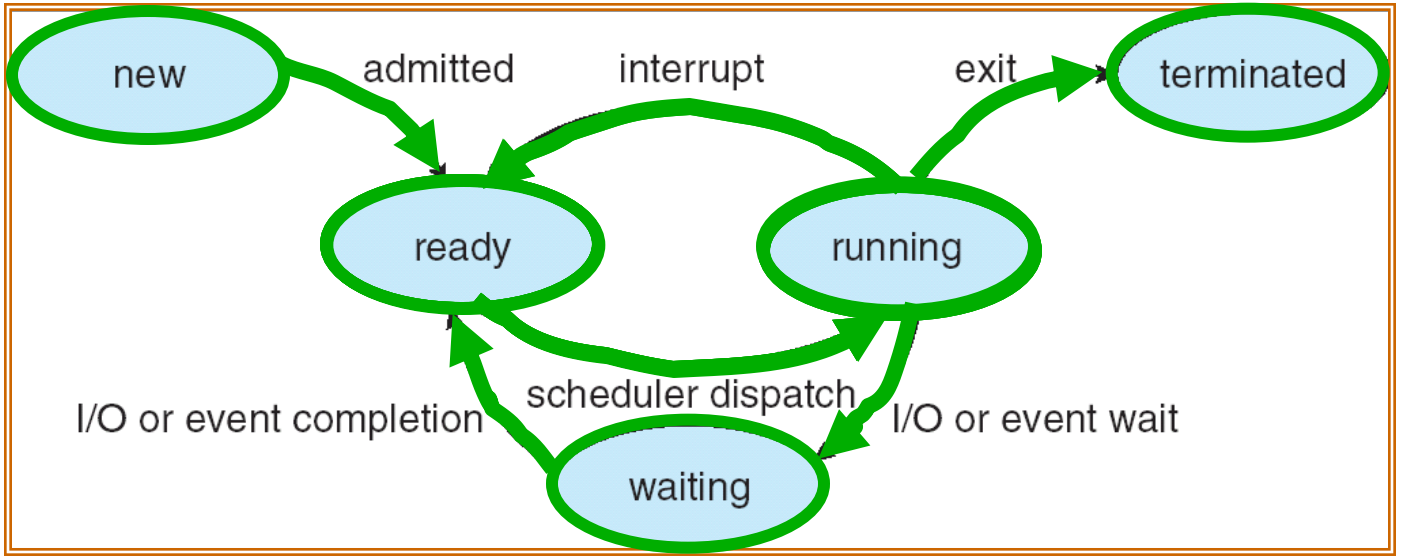
- If we stopped this program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks
- Questions:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?



Per Thread State

- Each Thread has a *Thread Control Block* (TCB)
  - Execution State: CPU registers, program counter, pointer to stack
  - Scheduling info: State (more later), priority, CPU time
  - Accounting Info
  - Various Pointers (for implementing scheduling queues)
  - Pointer to enclosing process? (PCB)?
  - Etc (add stuff as you find a need)
- OS Keeps track of TCBs in protected memory
  - In Array, or Linked List, or …

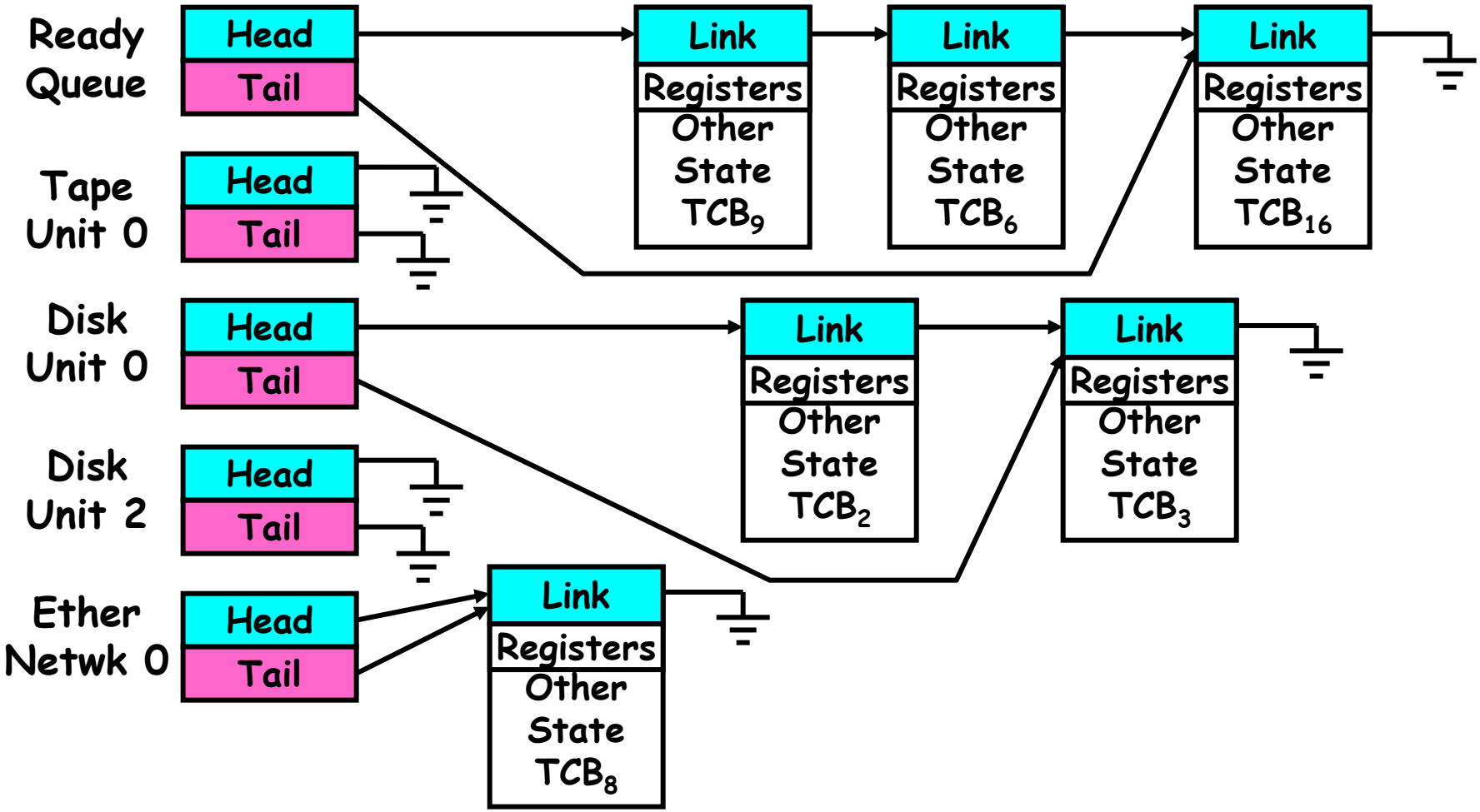
Lifecycle of a Thread (or Process)



- As a thread executes, it changes state:
  - **new**: The thread is being created
  - **ready**: The thread is waiting to run
  - **running**: Instructions are being executed
  - **waiting**: Thread waiting for some event to occur
  - **terminated**: The thread has finished execution
- "Active" threads are represented by their TCBs
  - TCBs organized into queues based on their state

Ready Queue And Various I/O Device Queues

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



Dispatch Loop

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

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

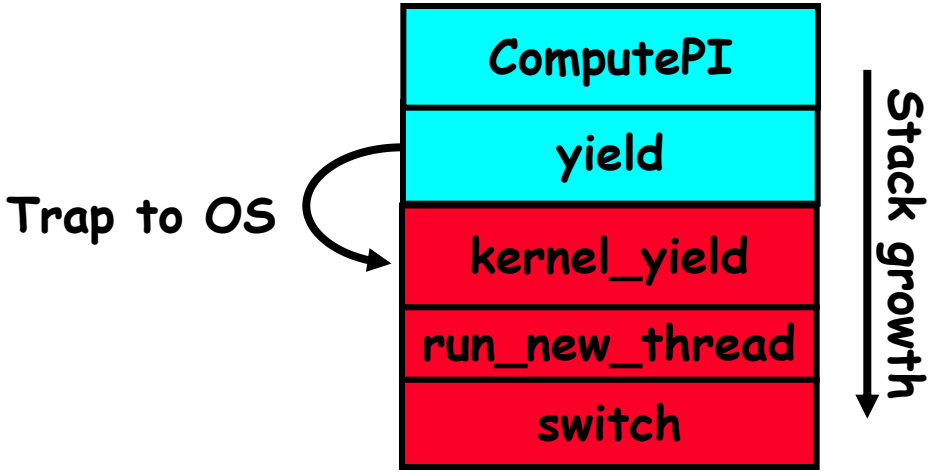
- This is an *infinite* loop
  - One could argue that this is all that the OS does
- Should we ever exit this loop???
  - When would that be?

Running a thread

Consider first portion: `RunThread()`

- 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*

Stack for Yielding Thread



- How do we run a new thread?

```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* next Lecture */
}
```

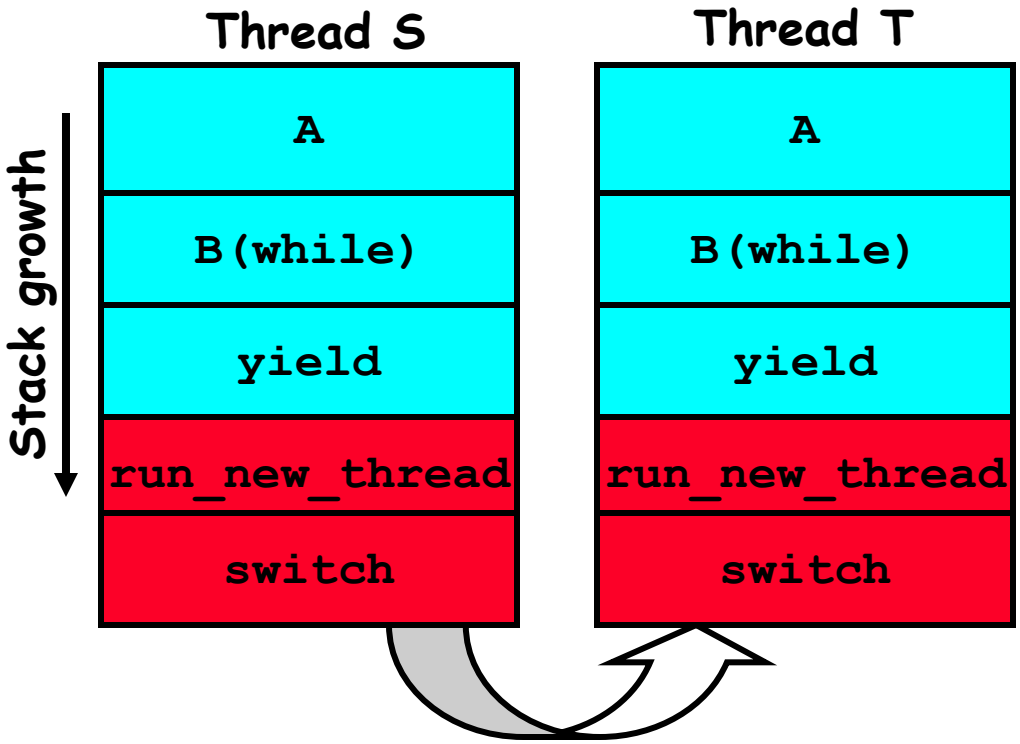
- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack
  - Maintain isolation for each thread

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 */
}
```

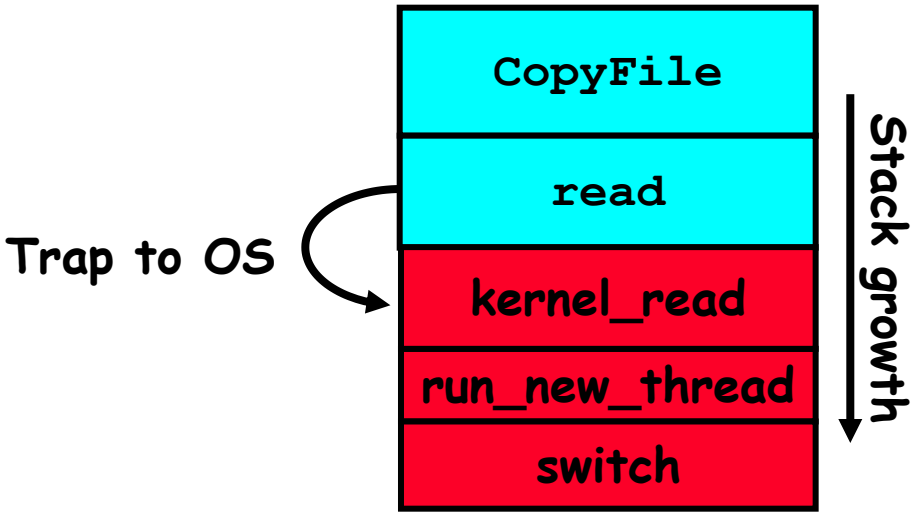
Switch Details

- How many registers need to be saved/restored?
  - MIPS 4k: 32 Int(32b), 32 Float(32b)
  - Pentium: 14 Int(32b), 8 Float(80b), 8 SSE(128b),…
  - Sparc(v7): 8 Regs(32b), 16 Int regs (32b) * 8 windows = 136 (32b)+32 Float (32b)
  - Itanium: 128 Int (64b), 128 Float (82b), 19 Other(64b)
- `retpc` is where the return should jump to.
  - In reality, this is implemented as a jump
- There is a real implementation of switch in Nachos.
  - See switch.s
    - » Normally, switch is implemented as assembly!
  - Of course, it's magical!
  - But you should be able to follow it!

Switch Details (continued)

- What if you make a mistake in implementing switch?
  - Suppose you forget to save/restore register 4
  - Get intermittent failures depending on when context switch occurred and whether new thread uses register 4
  - 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 tail:
  - 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

What happens when thread blocks on I/O?



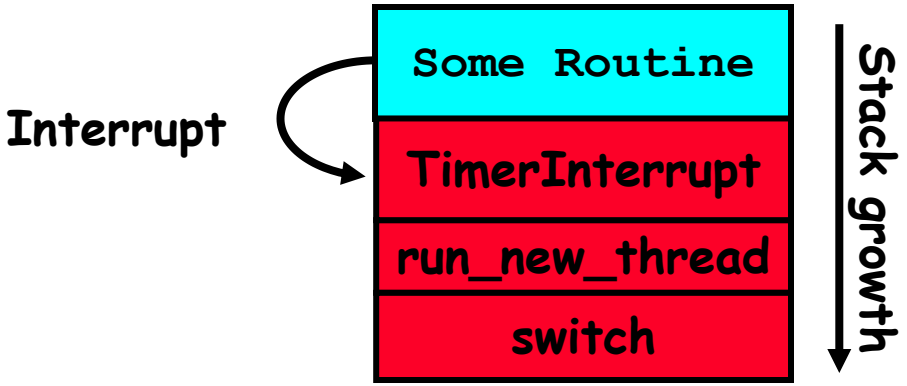
- What happens when a thread requests a block of data from the file system?
  - User code invokes a system call
  - Read operation is initiated
  - Run new thread/switch
- Thread communication similar
  - Wait for Signal/Join
  - Networking

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

Use of Timer Interrupt to Return Control

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



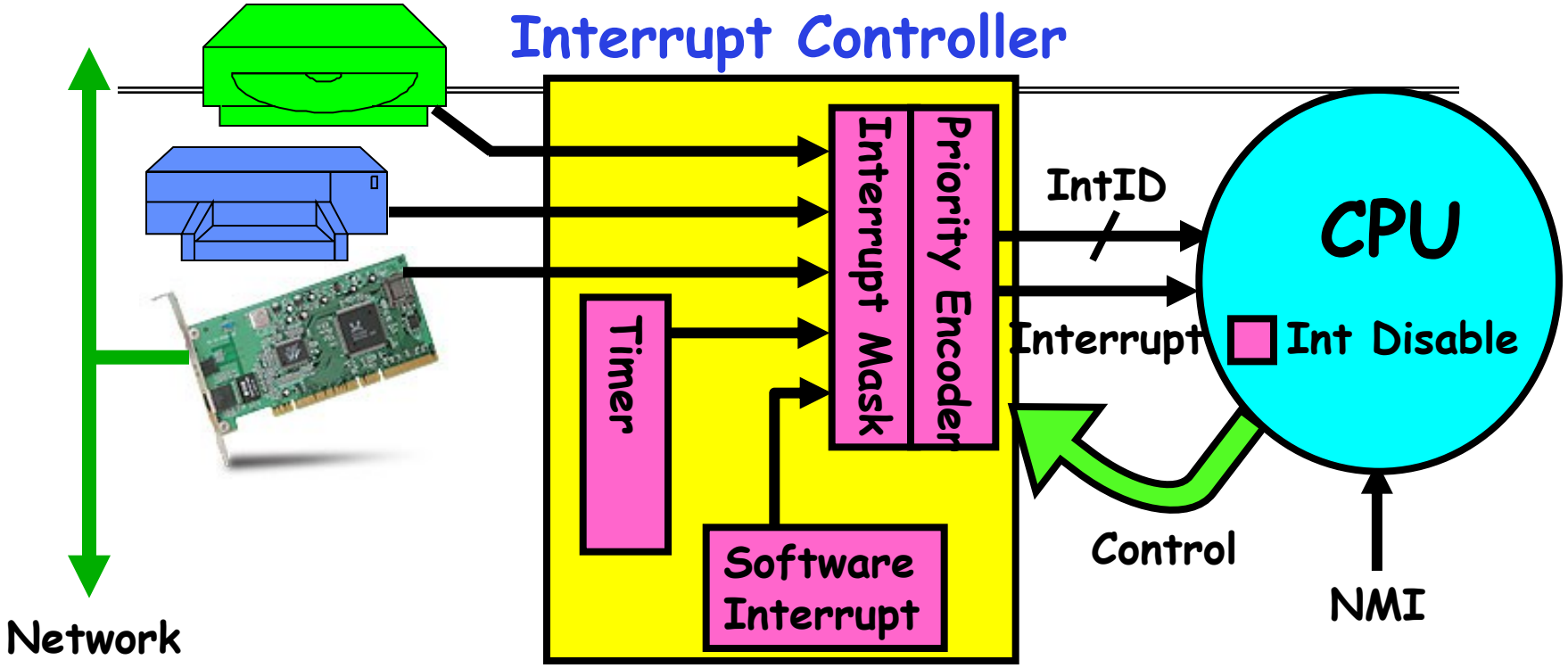
- Timer Interrupt routine:

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

- I/O interrupt: same as timer interrupt except that `DoHousekeeping()` replaced by `ServiceIO()`.

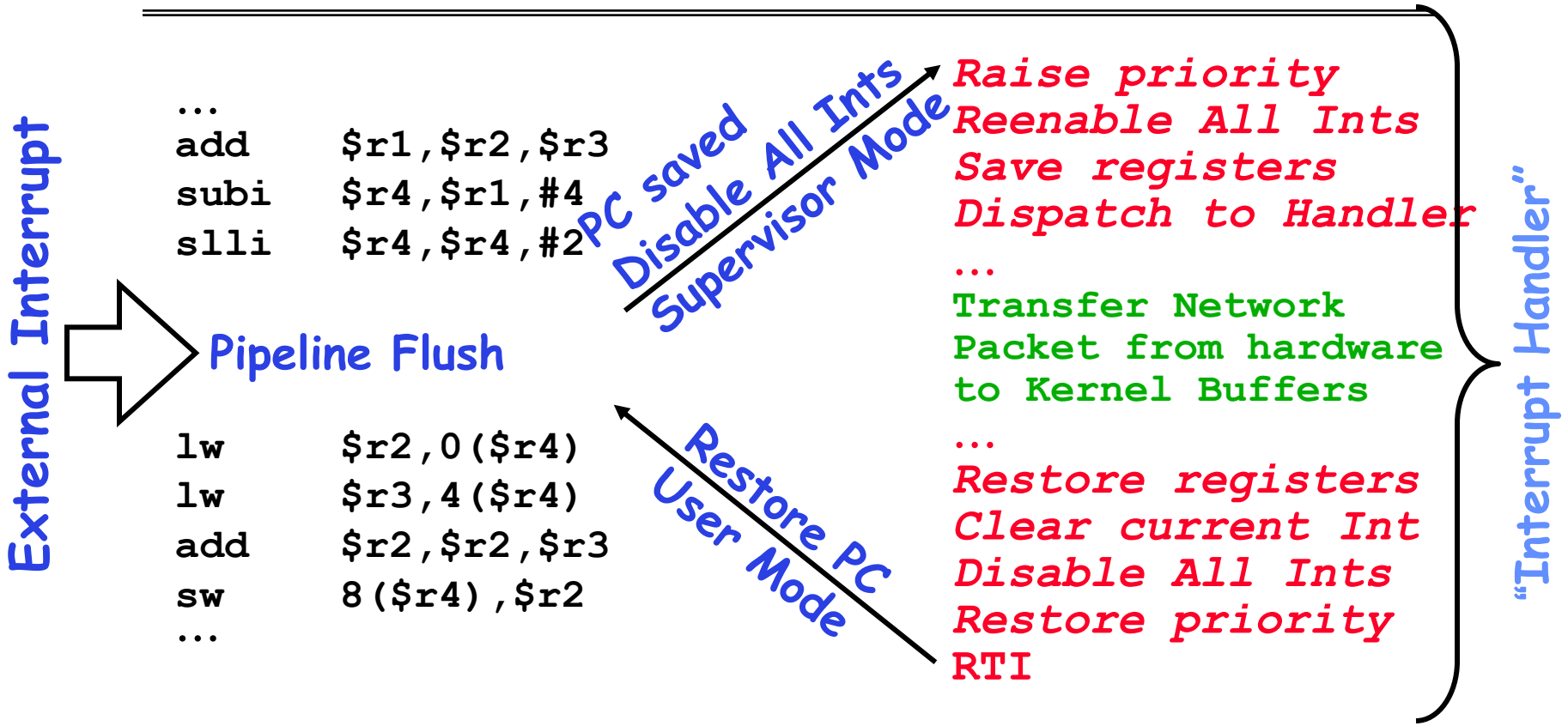
Choosing a Thread to Run

- How does Dispatcher decide what to run?
  - Zero ready threads – dispatcher loops
    - » Alternative is to create an "idle thread"
    - » Can put machine into low-power mode
  - Exactly one ready thread – easy
  - More than one ready thread: use scheduling priorities
- Possible priorities:
  - LIFO (last in, first out):
    - » put ready threads on front of list, remove from front
  - Pick one at random
  - FIFO (first in, first out):
    - » Put ready threads on back of list, pull them from front
    - » This is fair and is what Nachos does
  - Priority queue:
    - » keep ready list sorted by TCB priority field



- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
  - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-maskable interrupt line (NMI) can't be disabled

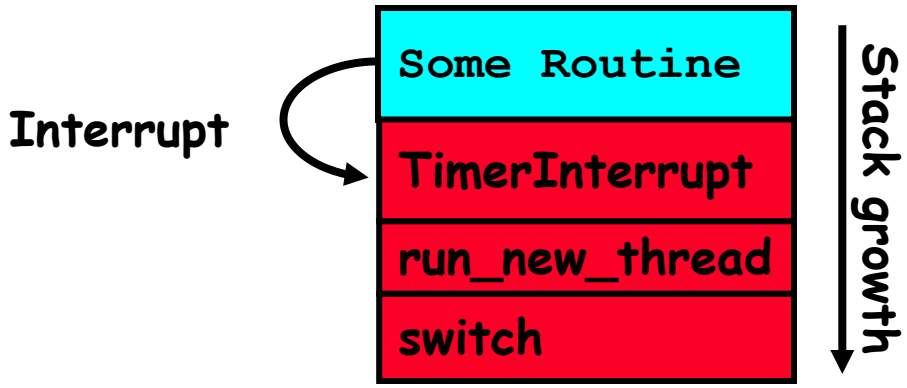
Example: Network Interrupt



- Disable/Enable All Ints ⇒ Internal CPU disable bit
  - RTI reenables interrupts, returns to user mode
- Raise/lower priority: change interrupt mask
- Software interrupts can be provided entirely in software at priority switching boundaries

Review: Preemptive Multithreading

- Use the timer interrupt to force scheduling decisions

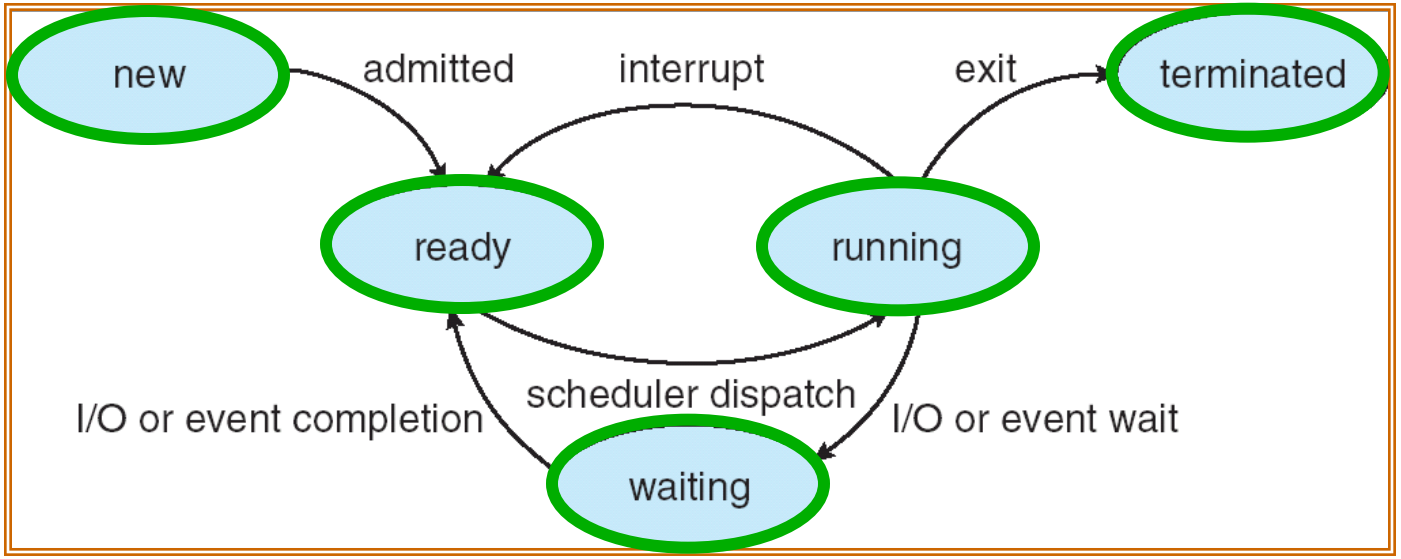


- Timer Interrupt routine:

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

- This is often called preemptive multithreading, since threads are preempted for better scheduling
  - Solves problem of user who doesn't insert yield();

Review: Lifecycle of a Thread (or Process)



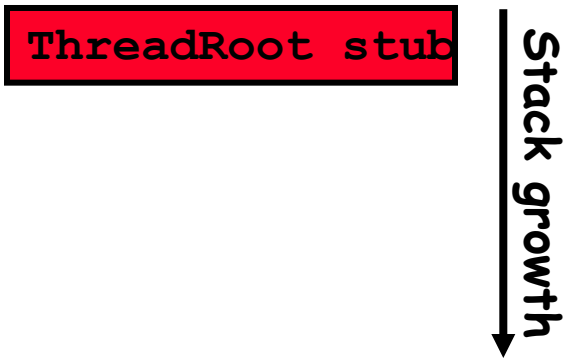
- As a thread executes, it changes state:
  - new: The thread is being created
  - ready: The thread is waiting to run
  - running: Instructions are being executed
  - waiting: Thread waiting for some event to occur
  - terminated: The thread has finished execution
- "Active" threads are represented by their TCBs
  - TCBs organized into queues based on their state

`ThreadFork()`: Create a New Thread

- `ThreadFork()` is a user-level procedure that creates a new thread and places it on ready queue
  - We called this `CreateThread()` earlier
- 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).

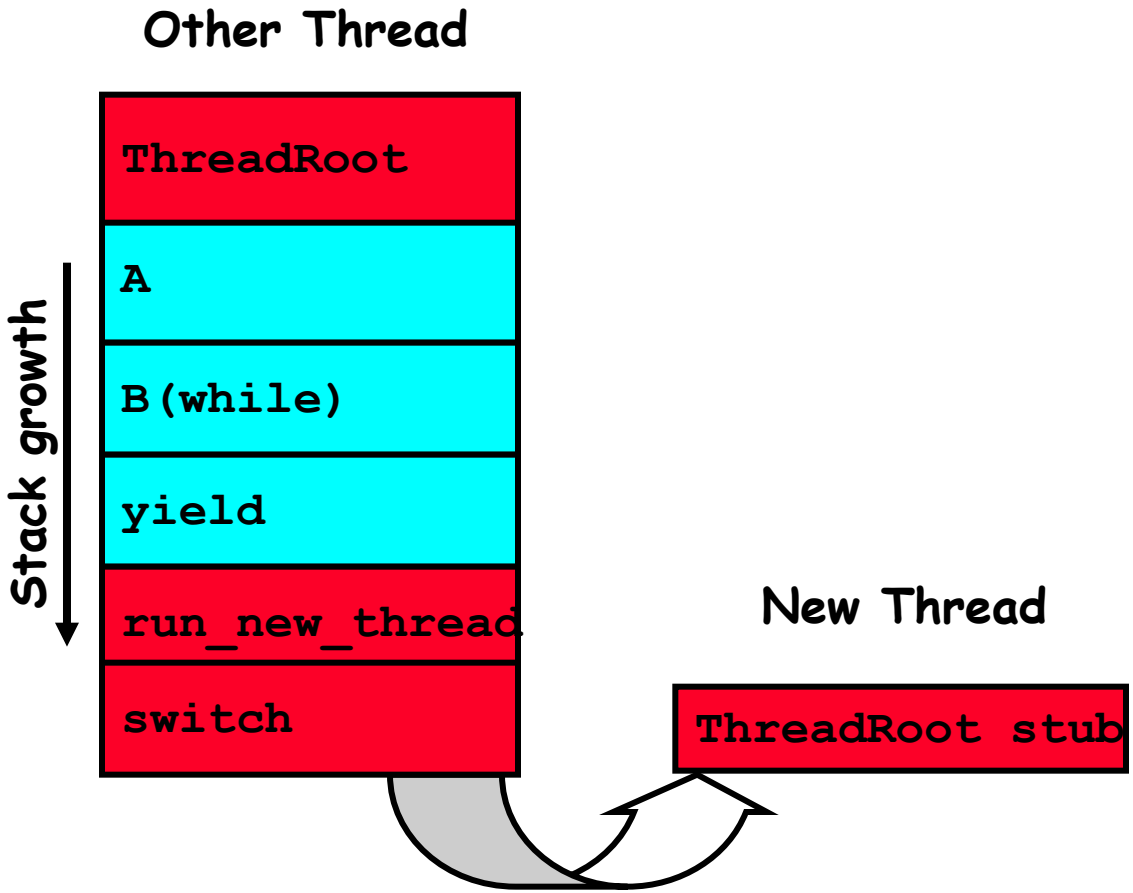
How do we initialize TCB and Stack?

- 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?



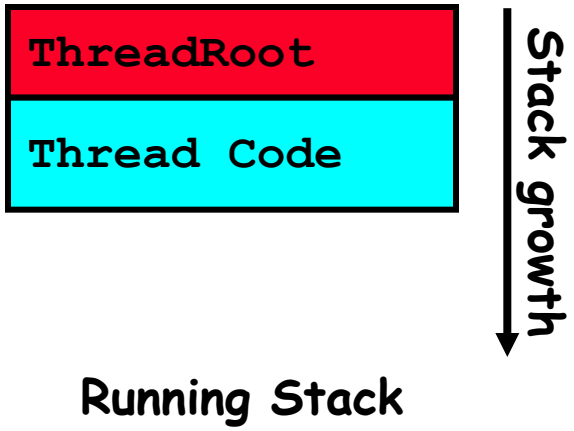
- Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`
  - This really starts the new thread

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() will start at user-level

What does `ThreadFinish()` do?

- Needs to re-enter kernel mode (system call)
- "Wake up" (place on ready queue) threads waiting for this thread
  - Threads (like the parent) may be on a wait queue waiting for this thread to finish
- Can't deallocate thread yet
  - We are still running on its stack!
  - Instead, record thread as "waitingToBeDestroyed"
- Call `run_new_thread()` to run another thread:

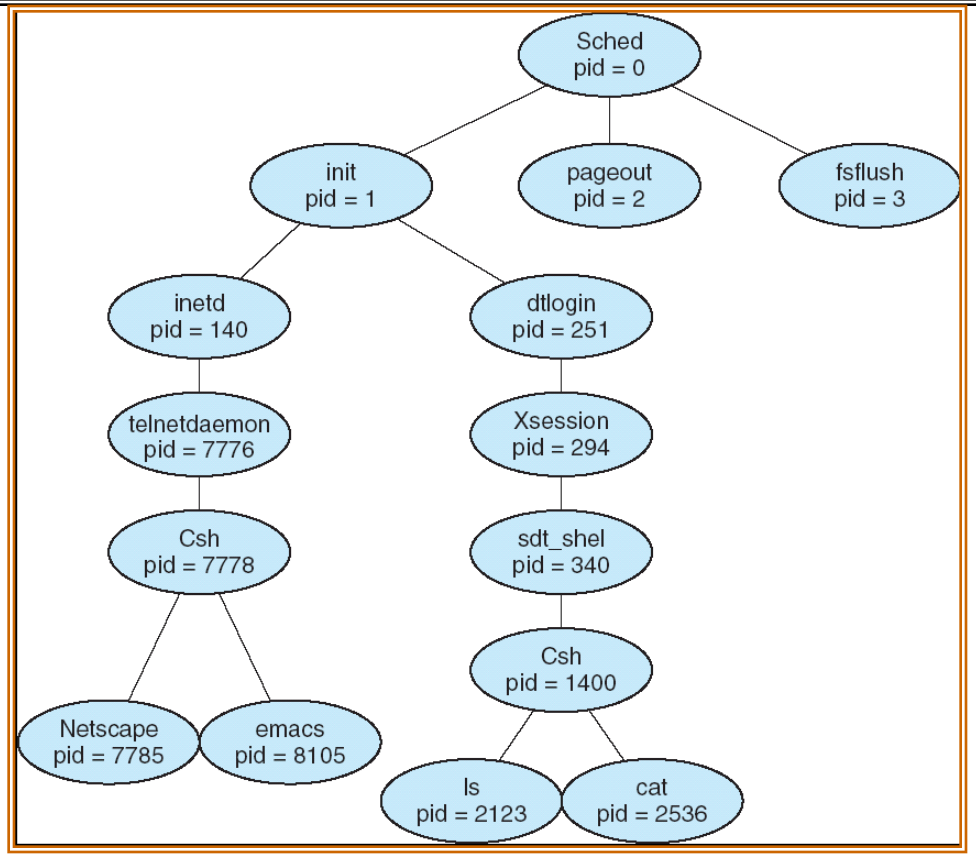
```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping();
}
```

  - `ThreadHouseKeeping()` notices waitingToBeDestroyed and deallocates the finished thread's TCB and stack

Additional Detail

- Thread Fork is not the same thing as UNIX fork
  - UNIX fork creates a new *process* so it has to create a new address space
  - For now, don't worry about how to create and switch between address spaces
- Thread fork is very much like an asynchronous procedure call
  - Runs procedure in separate thread
  - Calling thread doesn't wait for finish
- What if thread wants to exit early?
  - `ThreadFinish()` and `exit()` are essentially the same procedure entered at user level

Parent-Child relationship

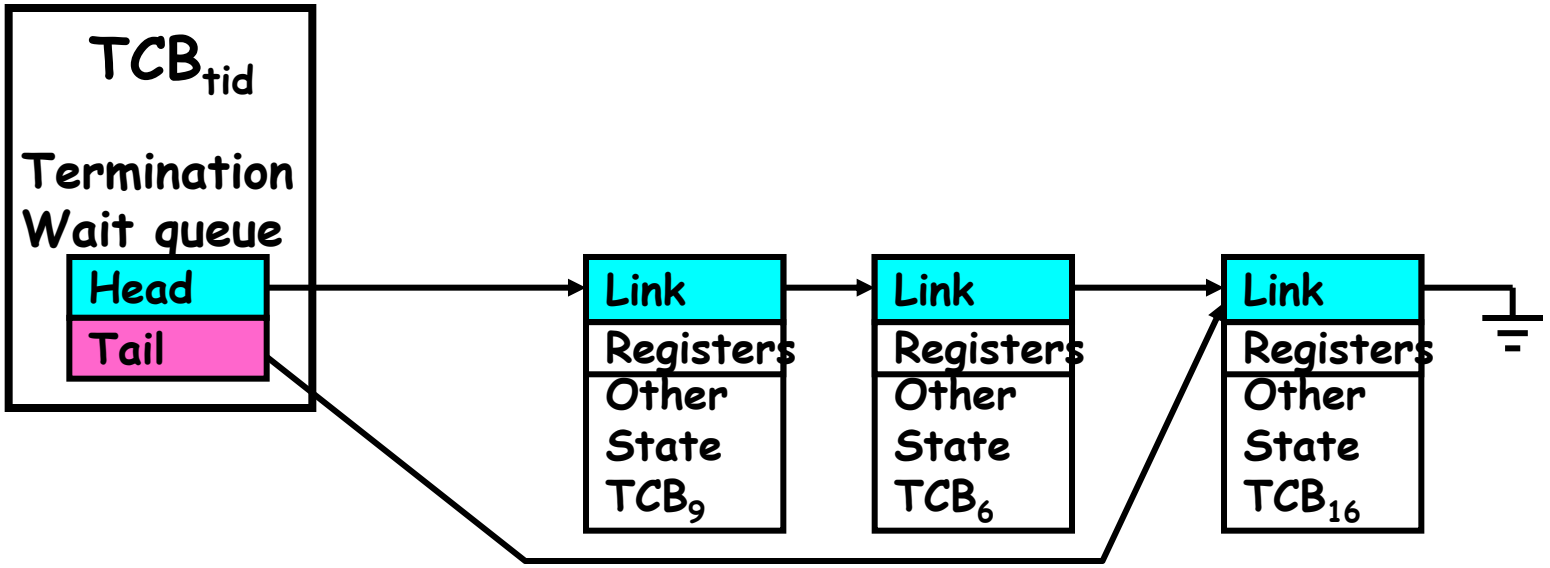


Typical process tree for Solaris system

- Every thread (and/or Process) has a parentage
  - A "parent" is a thread that creates another thread
  - A child of a parent was created by that parent

ThreadJoin() system call

- One thread can wait for another to finish with the `ThreadJoin(tid)` call
  - Calling thread will be taken off run queue and placed on waiting queue for thread `tid`
- Where is a logical place to store this wait queue?
  - On queue inside the TCB



- Similar to `wait()` system call in UNIX
  - Lets parents wait for child processes

Use of Join for Traditional Procedure Call

- A traditional procedure call is logically equivalent to doing a ThreadFork followed by ThreadJoin
- Consider the following normal procedure call of B() by A():

```
A() { B(); }
B() { Do interesting, complex stuff }
```

- The procedure A() is equivalent to A'():

```
A'() {
    tid = ThreadFork(B,null);
    ThreadJoin(tid);
}
```

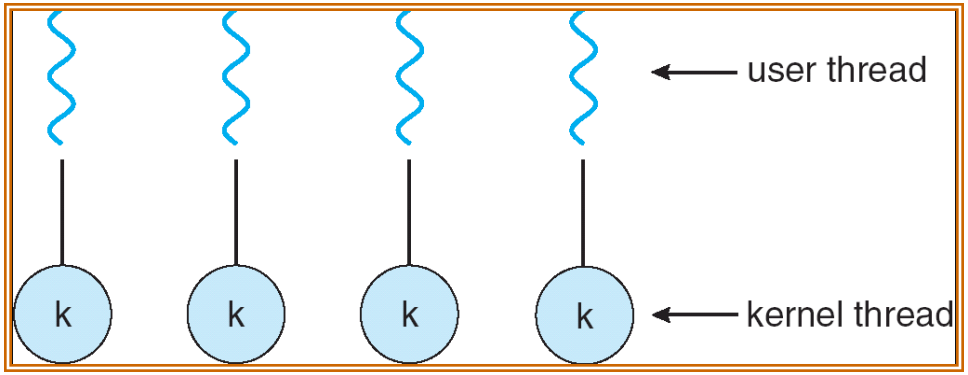
- Why not do this for every procedure?
  - Context Switch Overhead
  - Memory Overhead for Stacks

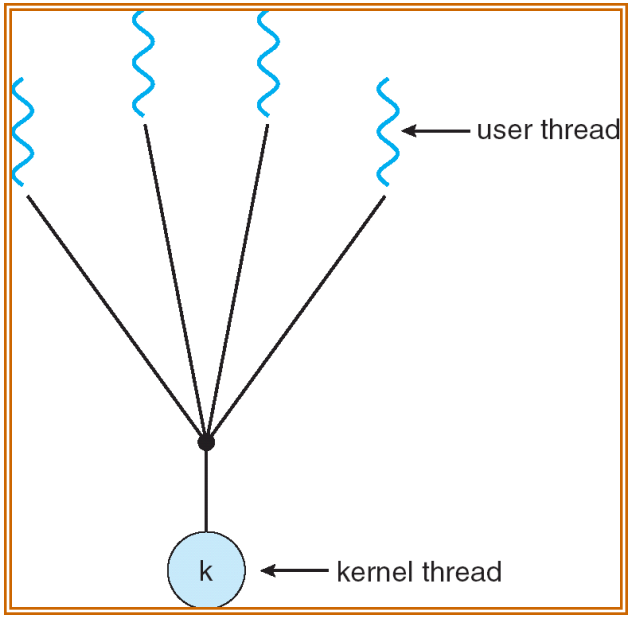
Kernel versus User-Mode threads

- 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
- Even lighter weight option: User Threads
  - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-premptively 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…

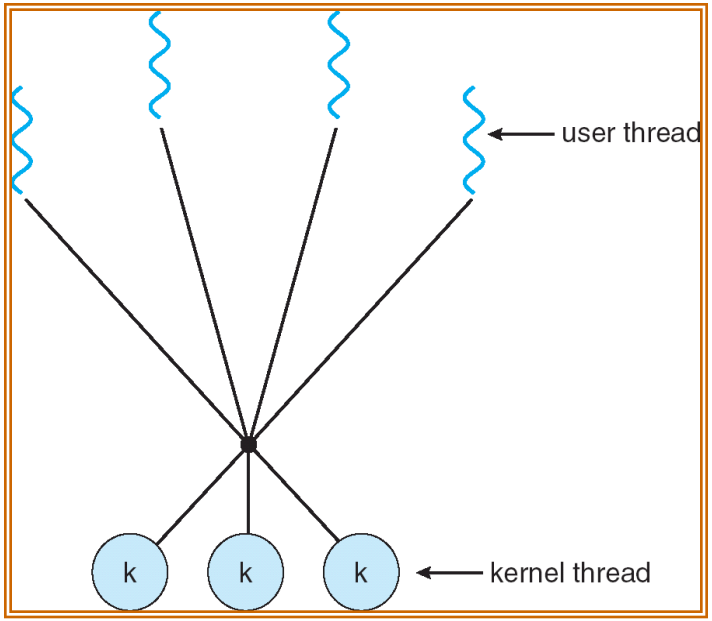
Threading models mentioned by book

Simple One-to-One Threading Model





Many-to-One



Many-to-Many

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

Summary

- The state of a thread is contained in the TCB
  - Registers, PC, stack pointer
  - States: New, Ready, Running, Waiting, or Terminated
- Multithreading provides simple illusion of multiple CPUs
  - Switch registers and stack to dispatch new thread
  - Provide mechanism to ensure dispatcher regains control
- Switch routine
  - Can be very expensive if many registers
  - Must be very carefully constructed!
- Many scheduling options
  - Decision of which thread to run complex enough for complete lecture

Summary

- Interrupts: hardware mechanism for returning control to operating system
  - Used for important/high-priority events
  - Can force dispatcher to schedule a different thread (premptive multithreading)
- New Threads Created with `ThreadFork()`
  - Create initial TCB and stack to point at `ThreadRoot()`
  - `ThreadRoot()` calls thread code, then `ThreadFinish()`
  - `ThreadFinish()` wakes up waiting threads then prepares TCB/stack for distruction
- Threads can wait for other threads using `ThreadJoin()`
- Threads may be at user-level or kernel level
- Cooperating threads have many potential advantages
  - But: introduces non-reproducibility and non-determinism
  - Need to have Atomic operations