

CS162
Operating Systems and
Systems Programming
Lecture 4

Thread Dispatching

January 28, 2010

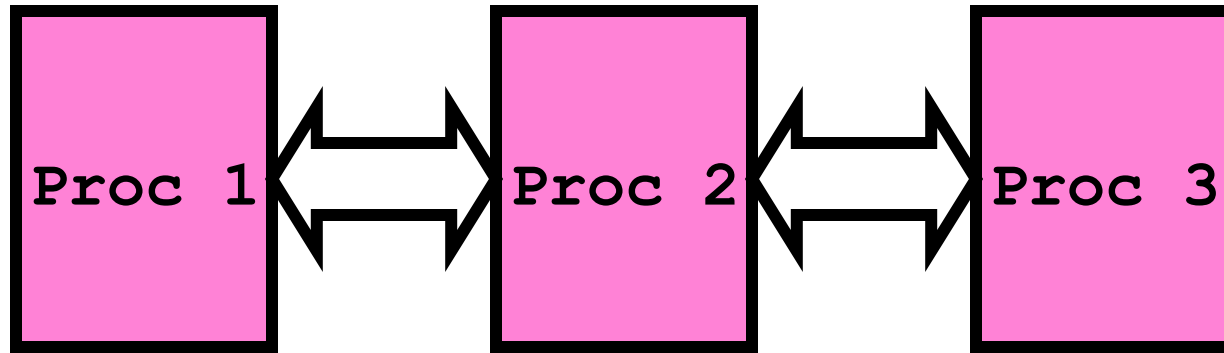
Ion Stoica

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

Programs, Processes, Threads

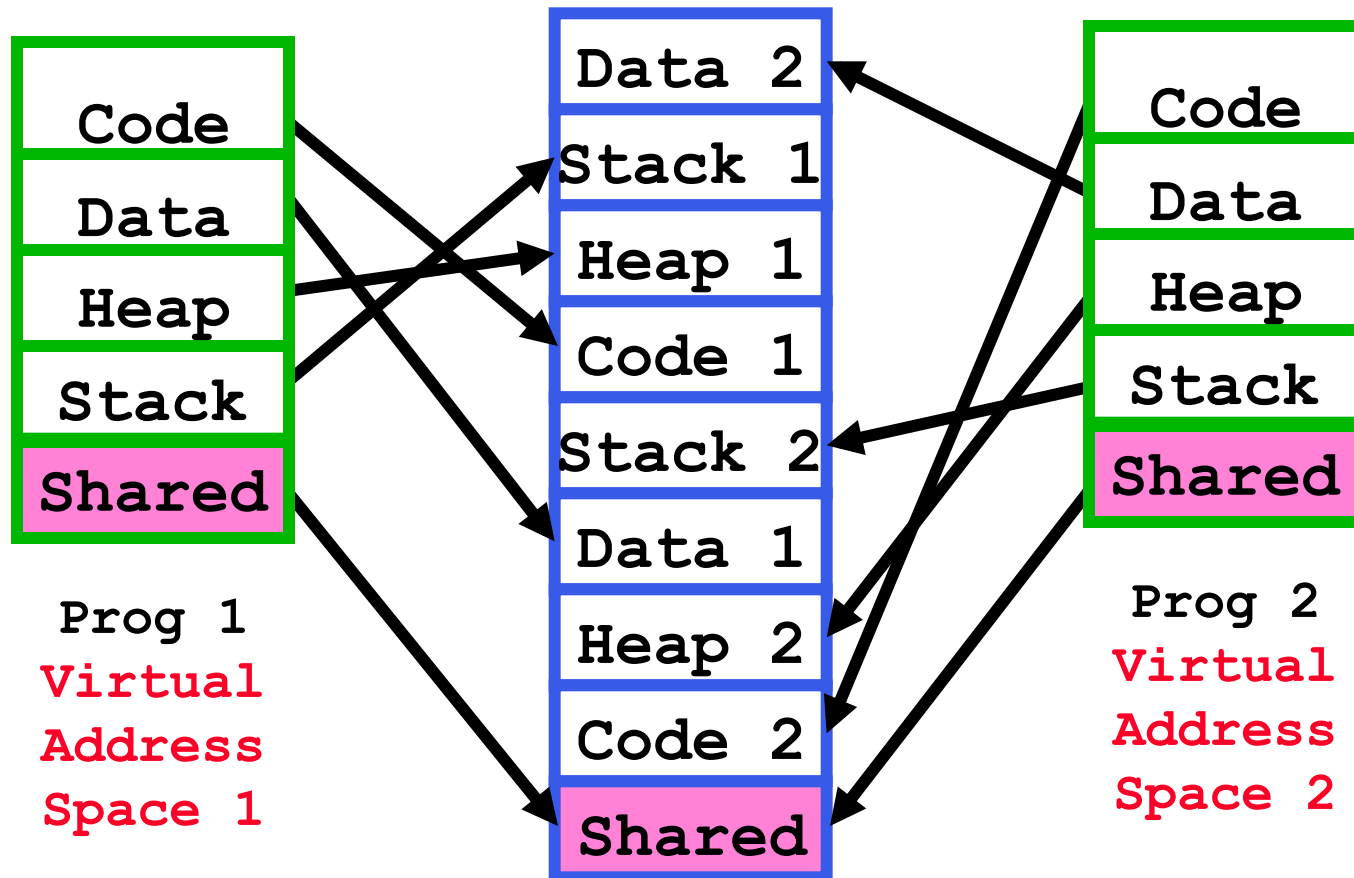
- Thread (lightweight process): unit of execution
 - A sequential execution stream of instructions
 - No protection between threads, other than CPU
- Process (heavyweight process): unit of resource allocation, management
 - Protect memory, I/O
- Why separate the concept of a thread from that of a process?
 - Discuss the “thread” part of a process (concurrency)
 - Separate from the “address space” (Protection)
 - Heavyweight Process \equiv Process with one thread

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

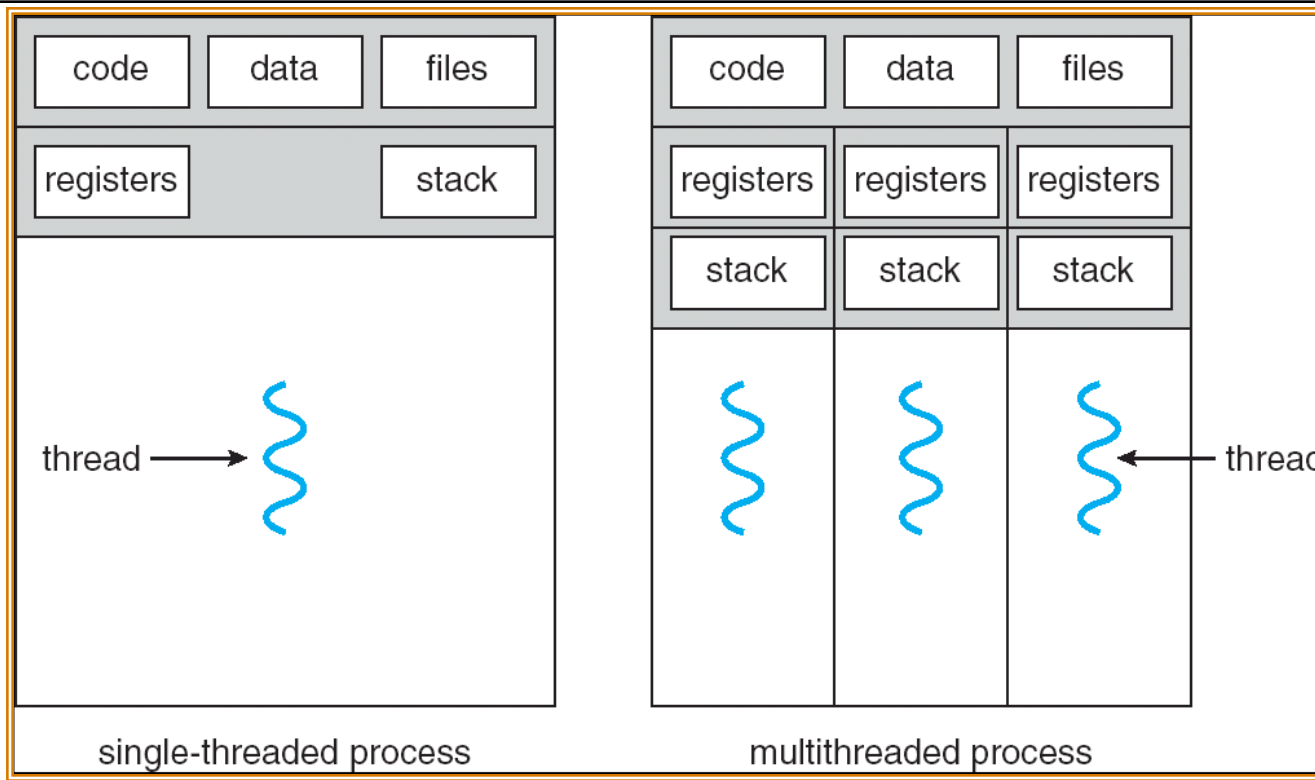


- Communication occurs by “simply” reading/writing to shared address page
 - Really low overhead communication
 - Introduces complex synchronization problems

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, syscall/trap)
 - logical (e.g., logical properties)

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 \equiv Thread Control Block
 - CPU registers (including, program counter)
 - Execution stack - what is this?
- Execution Stack
 - Parameters (function arguments), return values
 - return PCs are kept while called procedures are executing

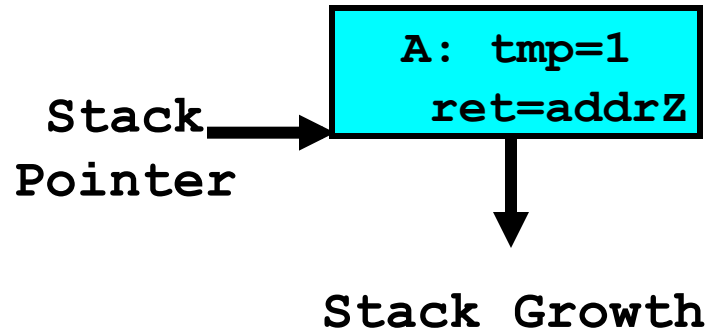
Execution Stack Example

```
addrX:  A(int tmp) {  
  .  
  .    if (tmp<2)  
  .  
  .      B();  
addrY:  printf(tmp);  
  .  
  .    }  
  .  
  .    B() {  
  .  
  .      C();  
addrU:  }  
  .  
  .    C() {  
  .  
  .      A(2);  
addrV:  }  
  .  
  .    A(1);  
addrZ:  }  
  .  
  .    exit;
```

- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

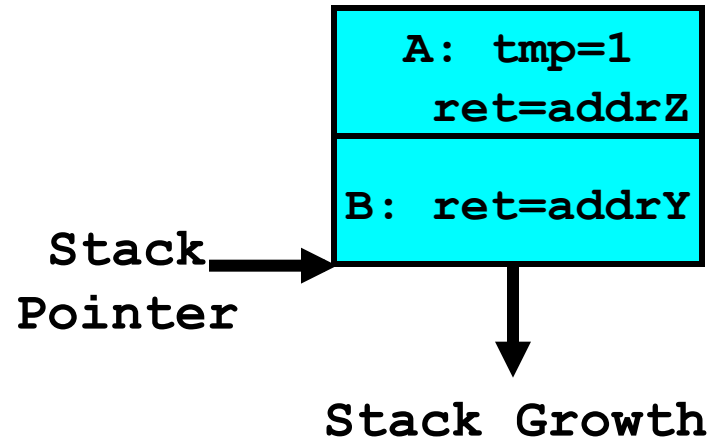
```
→ addrX: A(int tmp) {  
  .  
  .   if (tmp<2)  
  .  
  .     B();  
addrY:   printf(tmp);  
  .  
  .     }  
  .  
  .     B() {  
  .  
  .       C();  
addrU:   }  
  .  
  .     C() {  
  .  
  .       A(2);  
addrV:   }  
  .  
  .     A(1);  
  .  
addrZ:   exit;
```



- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

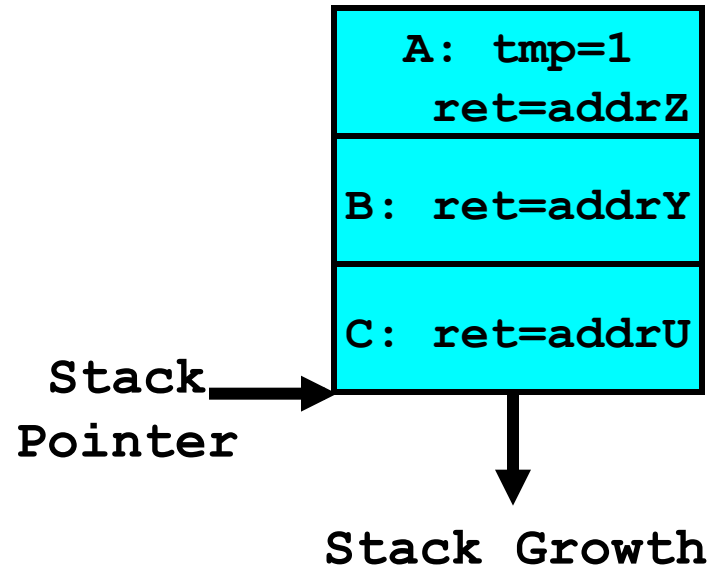
```
addrX:  A(int tmp) {  
  .  
  .     if (tmp<2)  
  .  
  .     B();  
addrY:  printf(tmp);  
  .  
  .     }  
  .  
→ .     B() {  
  .  
  .     C();  
addrU:  }  
  .  
  .     C() {  
  .  
  .     A(2);  
addrV:  }  
  .  
  .     A(1);  
  .  
addrZ:  exit;
```



- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

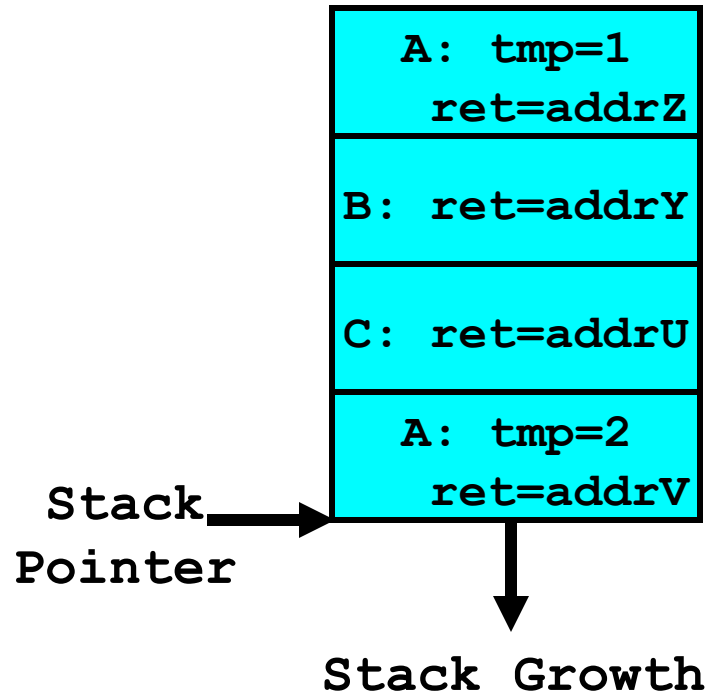
```
addrX:  A(int tmp) {  
  .  
  .     if (tmp<2)  
  .  
  .     B();  
addrY:  printf(tmp);  
  .  
  .     }  
  .  
  .     B() {  
  .  
  .     C();  
addrU:  }  
  .  
→      C() {  
  .  
  .     A(2);  
addrV:  }  
  .  
  .     A(1);  
addrZ:  }  
        exit;
```



- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
→ addrX: A(int tmp) {  
  .  
  .   if (tmp<2)  
  .  
  .   B();  
addrY:  printf(tmp);  
  .  
  .   }  
  .  
  .   B() {  
  .  
  .   C();  
addrU:  }  
  .  
  .   C() {  
  .  
  .   A(2);  
addrV:  }  
  .  
  .   A(1);  
addrZ:  exit;
```



- Stack holds function arguments, return address
- 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, Win 95?, Mac OS X, Win NT to XP, Solaris, HP-UX

- 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

Administrivia: Project Signup

- Project Signup: Use "Group/Section Signup" Link
 - 4-5 members to a group
 - » Everyone in group must be able to *actually* attend same section
 - Only submit once per group!
 - » Everyone in group must have logged into their cs162-xx accounts once before you register the group
 - » **Due Friday 1/29 by 11:59pm**
- Anyone without a group?
- Sections in this class are mandatory
 - Go to the section that you have been assigned!
 - Important information will be given in section
 - 5% of grade is participation
- Reader now available at Copy Central on Hearst
- Other things on Handouts page
 - Interesting papers
 - Synchronization examples
 - Previous finals/solutions

Goals for Today

- Further Understanding Threads
- Thread Dispatching
- Beginnings of Thread Scheduling

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatoicz.

Single-Threaded Example

- Imagine the following C program:

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

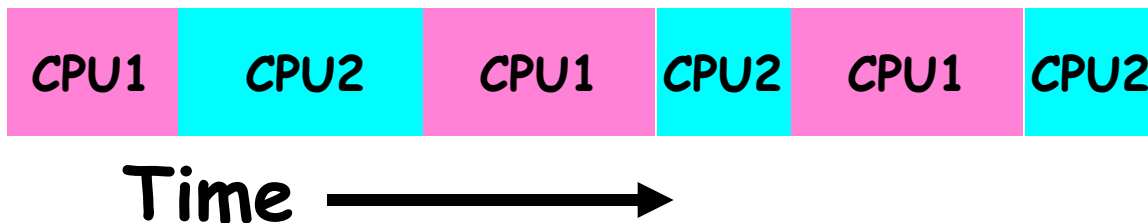
- What is the behavior here?
 - Program would never print out class list
 - Why? ComputePI would never finish

Use of Threads

- Version of program with Threads:

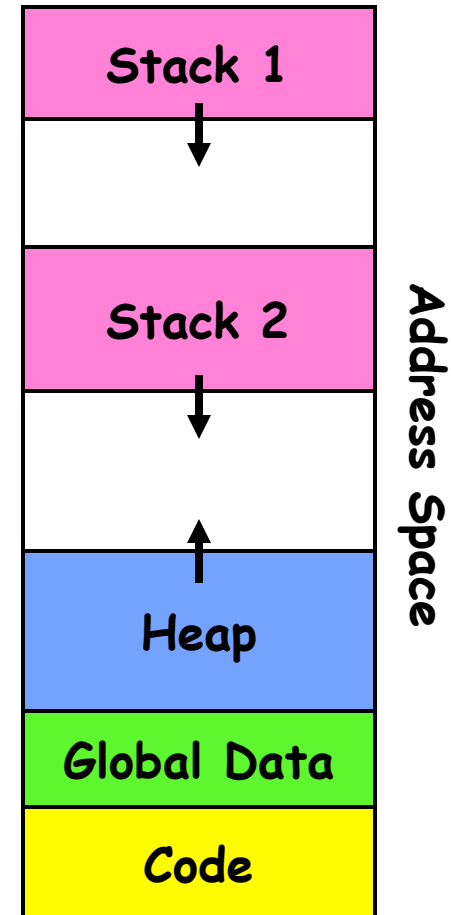
```
main() {  
    CreateThread(ComputePI("pi.txt"));  
    CreateThread(PrintClassList("clist.txt"));  
}
```

- What does "CreateThread" 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 of Two-Thread Example

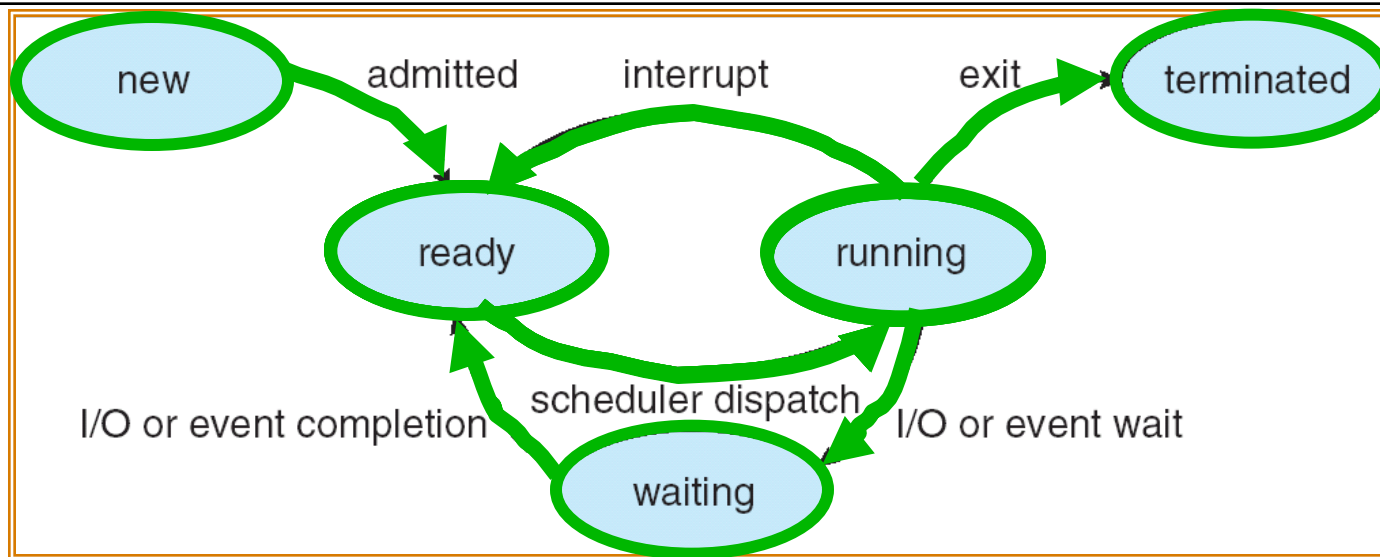
- If we stopped this program and examined it with a debugger, we would see
 - Two sets of CPU registers
 - Two sets of Stacks
- Questions:
 - How do we position stacks relative to each other?
 - What maximum size should we choose for the stacks?
 - What happens if threads violate this?
 - How might you 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)
- In Nachos: "Thread" is a class that includes the TCB
- OS Keeps track of TCBs in protected memory
 - In Array, or Linked List, or ...

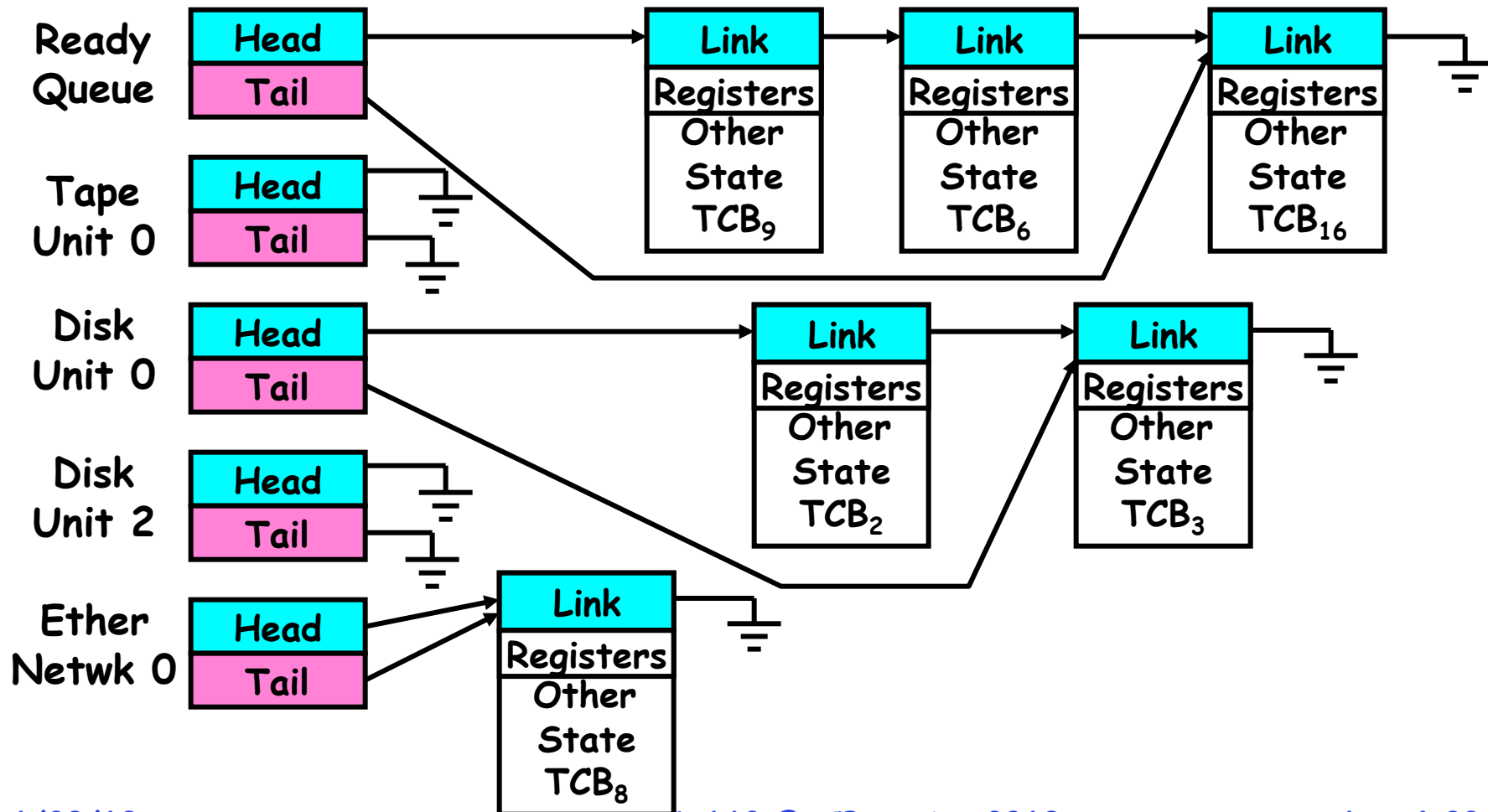
Lifecycle of a Thread (or Process)



- 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 \Rightarrow 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()`

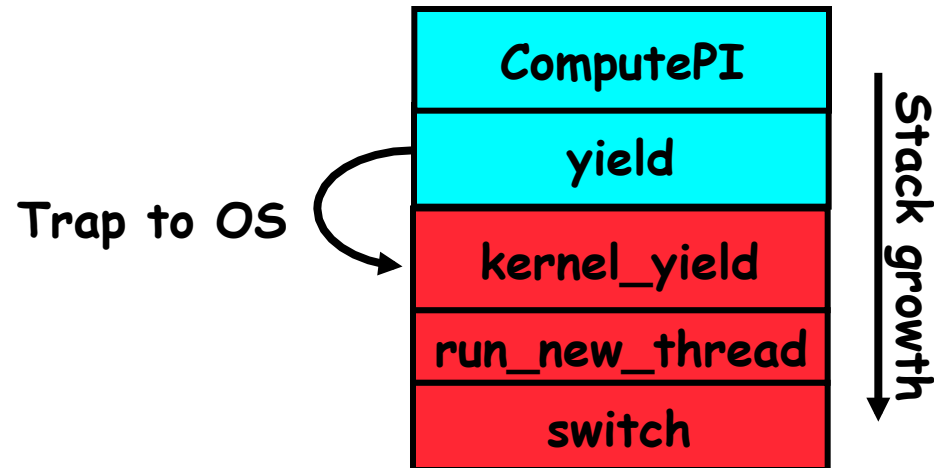
- How do I run a thread?
 - Load its state (registers, PC, stack pointer) into CPU
 - Load environment (virtual memory space, etc)
 - Jump to the PC
- How does the dispatcher get control back?
 - Internal events: thread returns control voluntarily
 - External events: thread gets *preempted*

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(); /* next Lecture */  
}
```

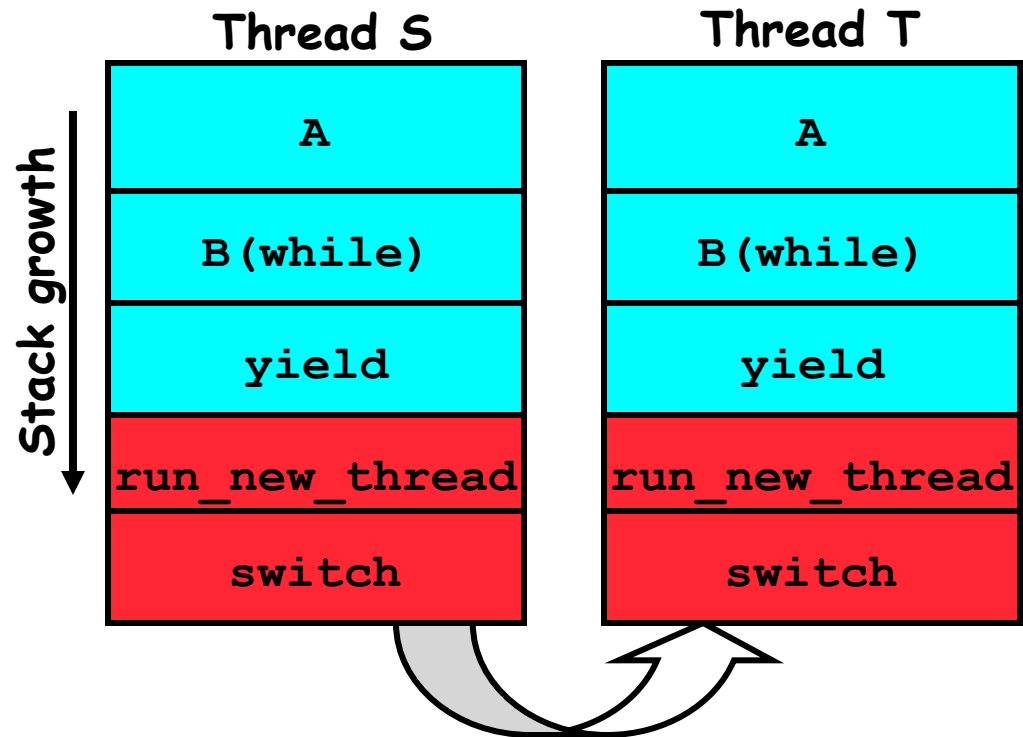
- How does dispatcher switch to a new thread?
 - Save anything next thread may trash: PC, regs, stack
 - Maintain isolation for each thread

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 */  
}
```

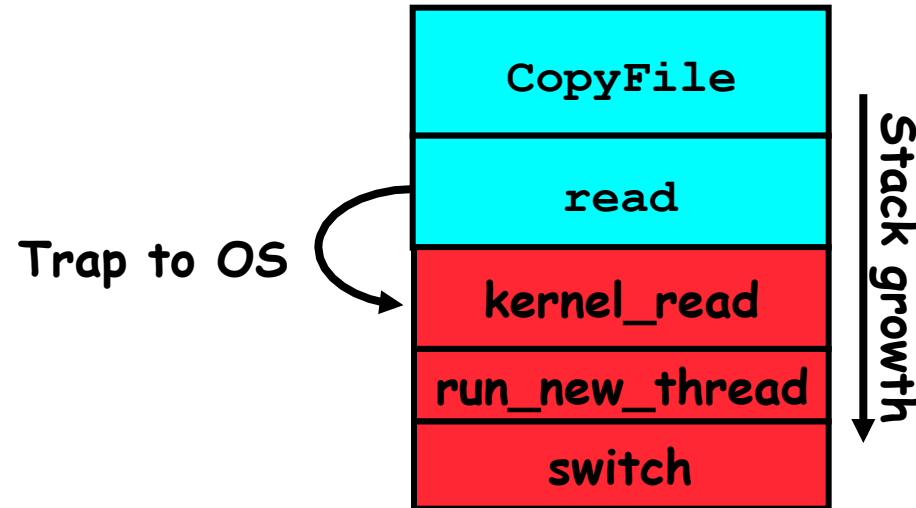
Switch Details

- 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?

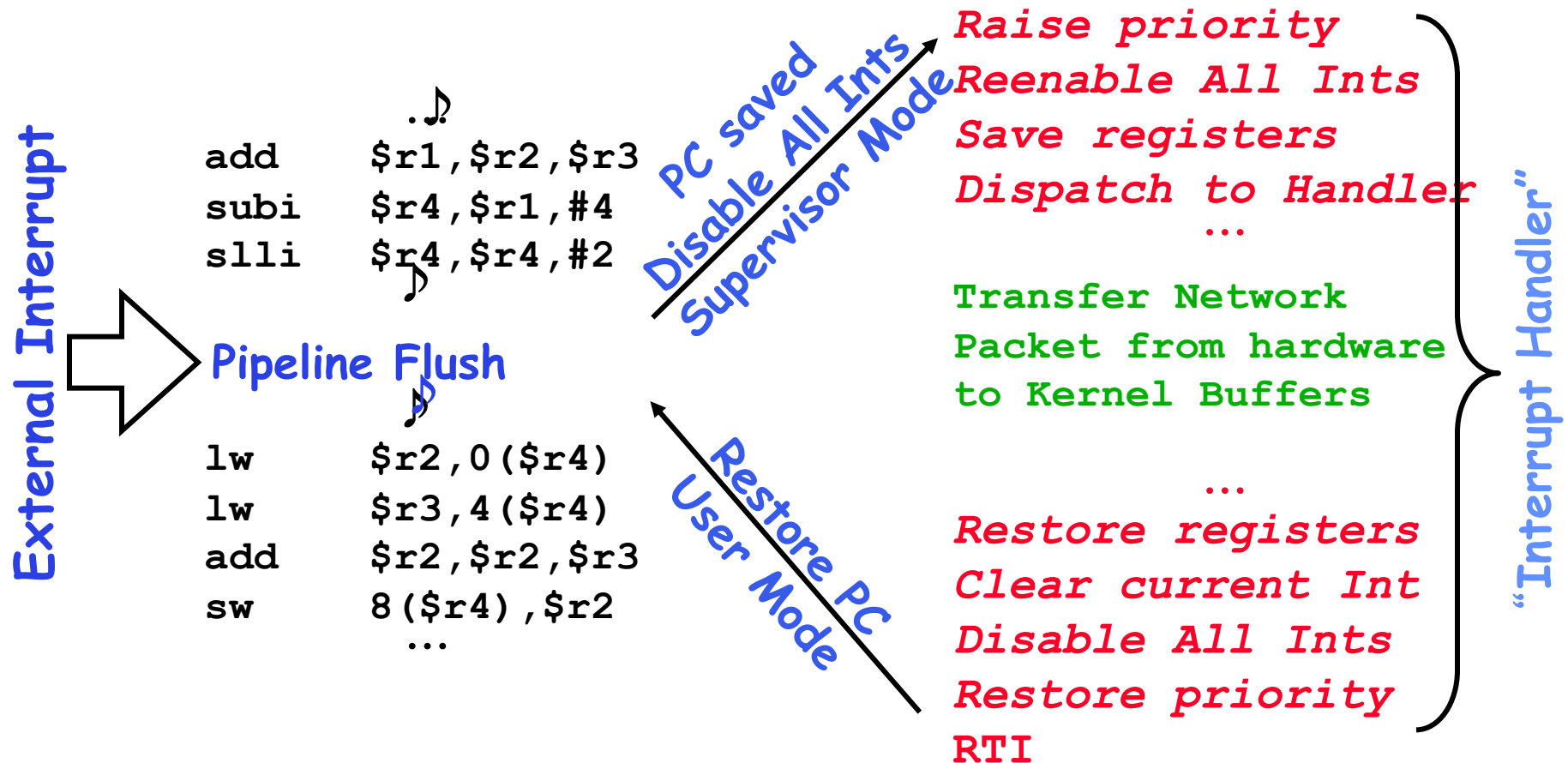


- What happens when a thread requests a block of data from the file system?
 - User code invokes a system call
 - Read operation is initiated
 - Run new thread/switch
- Thread communication similar
 - Wait for Signal/Join
 - Networking

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

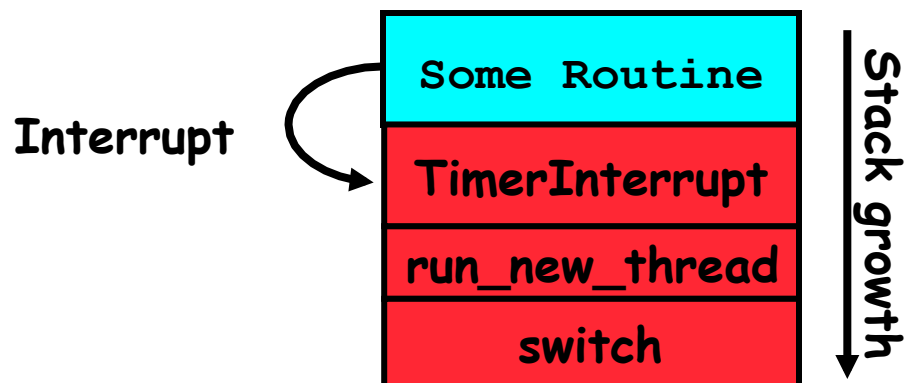
Example: Network Interrupt



- 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();  
}
```
- 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

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