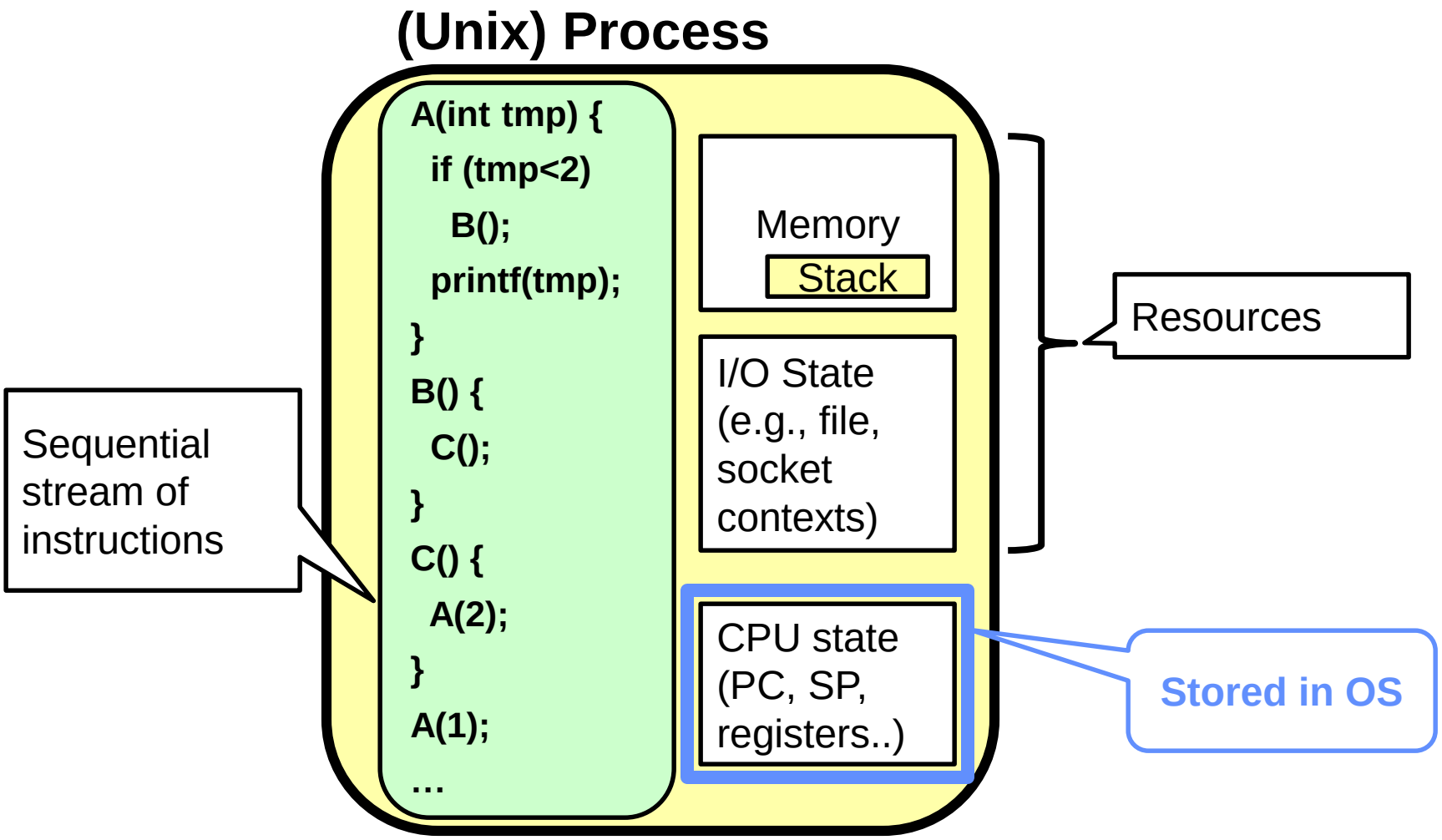
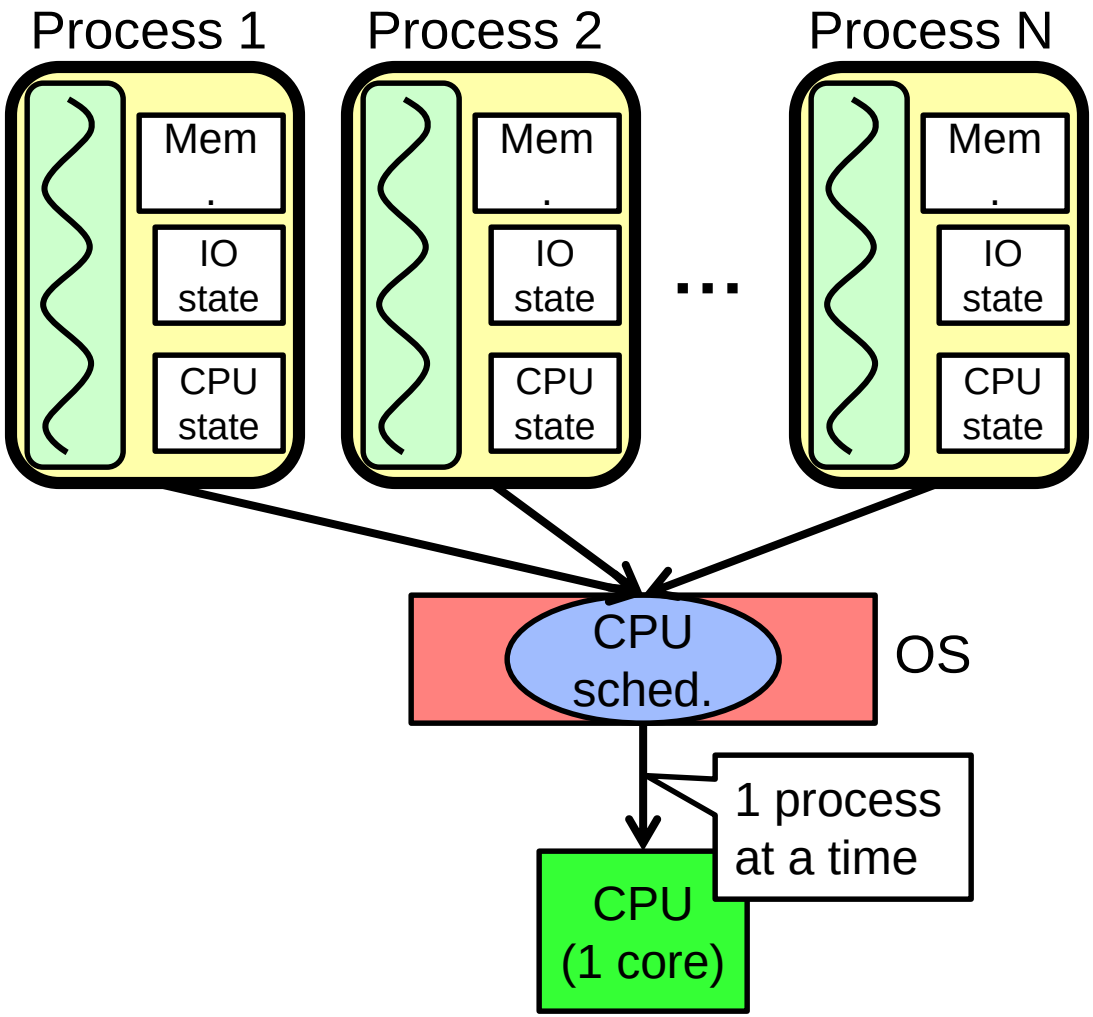
CSE150
Operating Systems
Lecture 5

Concurrency and Thread Dispatching (contd.)

Process (Review)

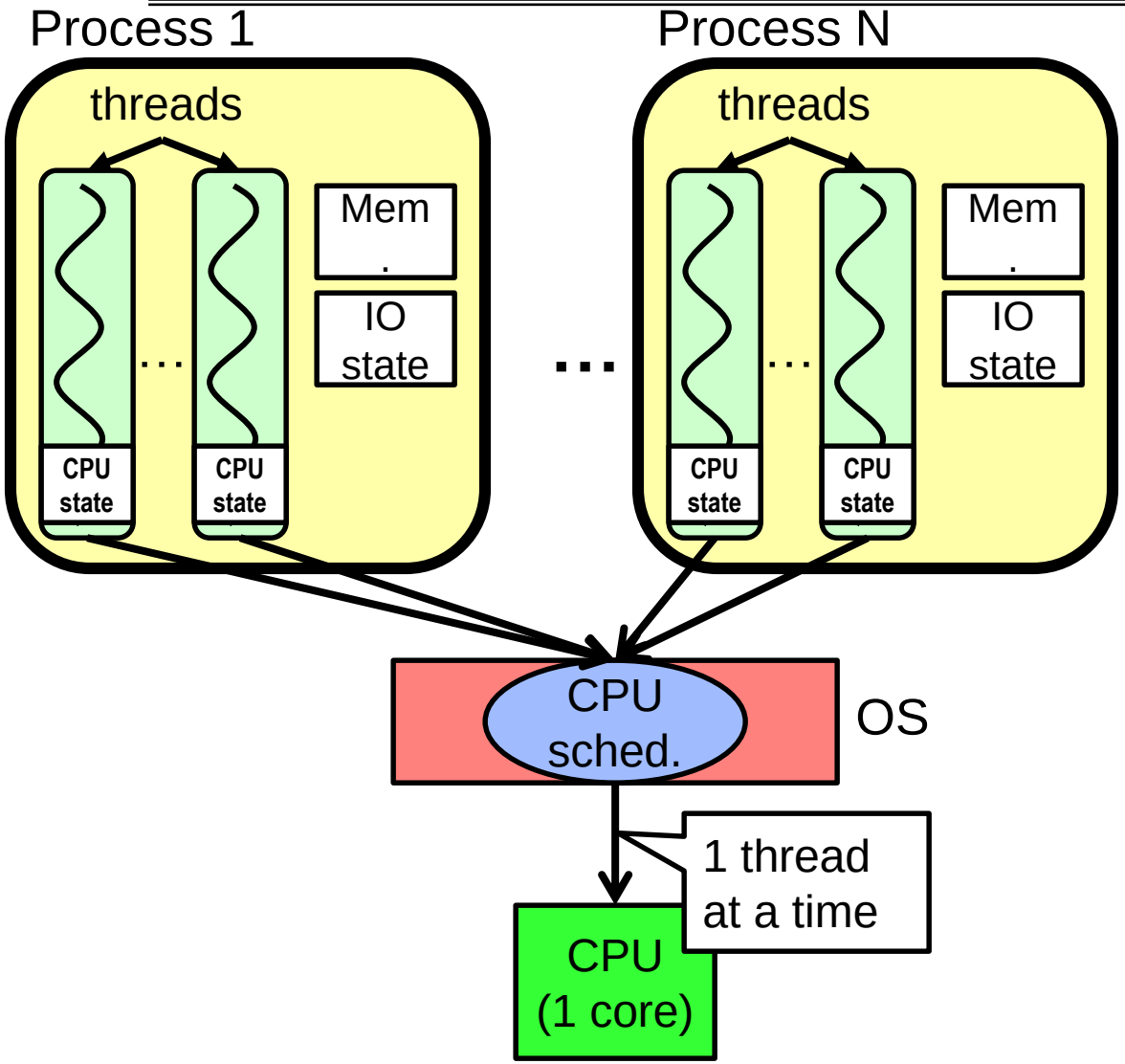


Processes (Review)



- 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)

Multithreaded Processes (Review)



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

Why Processes & Threads? (Review)

Goals:

- **Multiprogramming:** Run multiple applications concurrently
- **Protection:** Don't want a bad application to crash system!

Solution:

Process: unit of execution and allocation

- Virtual Machine abstraction: give process illusion it owns machine (i.e., CPU, Memory, and IO device multiplexing)

Challenge:

- Need concurrency within same app (e.g., web server)
- Process creation, communication & switching expensive

Solution:

Thread: Decouple allocation and execution

- Run multiple threads within same process

Thread State (Review)

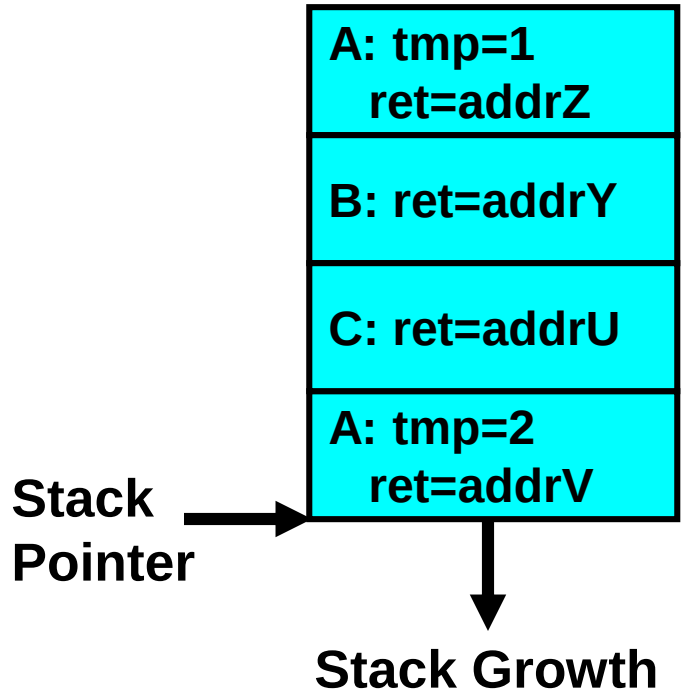
- State shared by all threads in process/address space
  - Content of memory (global variables, heap)
  - I/O state (file system, network connections, etc)

- 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 (Review)

```
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

Single-Threaded Example (Review)

- Imagine the following C program:

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

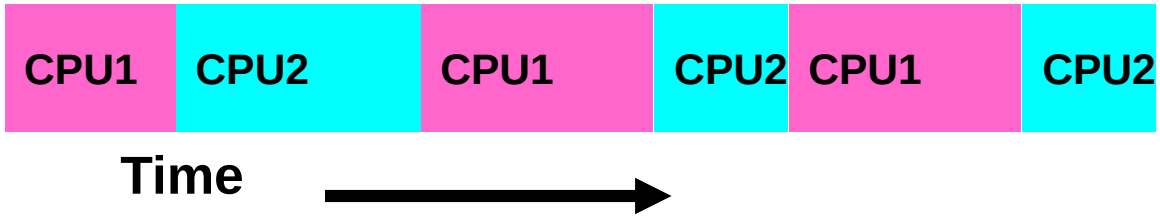
- What is the behavior here?
  - Program would never print out class list
  - Why? ComputePI would never finish

Use of Threads (Review)

- Version of program with Threads:

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

- 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



Today

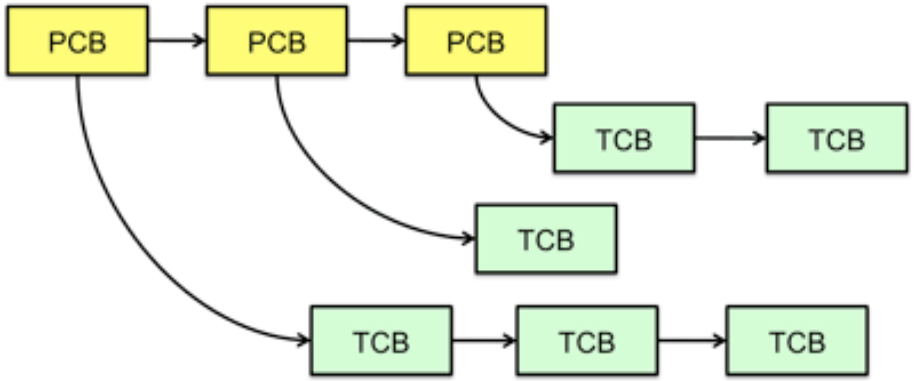
- Further Understanding Threads
  - Dispatching
  - Beginnings of Thread Scheduling

Why allow cooperating threads?

- People cooperate; computers help/enhance people's lives, so computers must cooperate
  - By analogy, the non-reproducibility/non-determinism of people is a notable problem for "carefully laid plans"
- Advantage 1: Share resources
  - One computer, many users
  - One bank balance, many ATMs
    - » What if ATMs were only updated at night?
  - Embedded systems (robot control: coordinate arm & hand)
- Advantage 2: Speedup
  - Overlap I/O and computation
  - Multiprocessors – chop up program into parallel pieces
- Advantage 3: Modularity
  - Chop large problem up into simpler pieces
    - » Makes system easier to extend

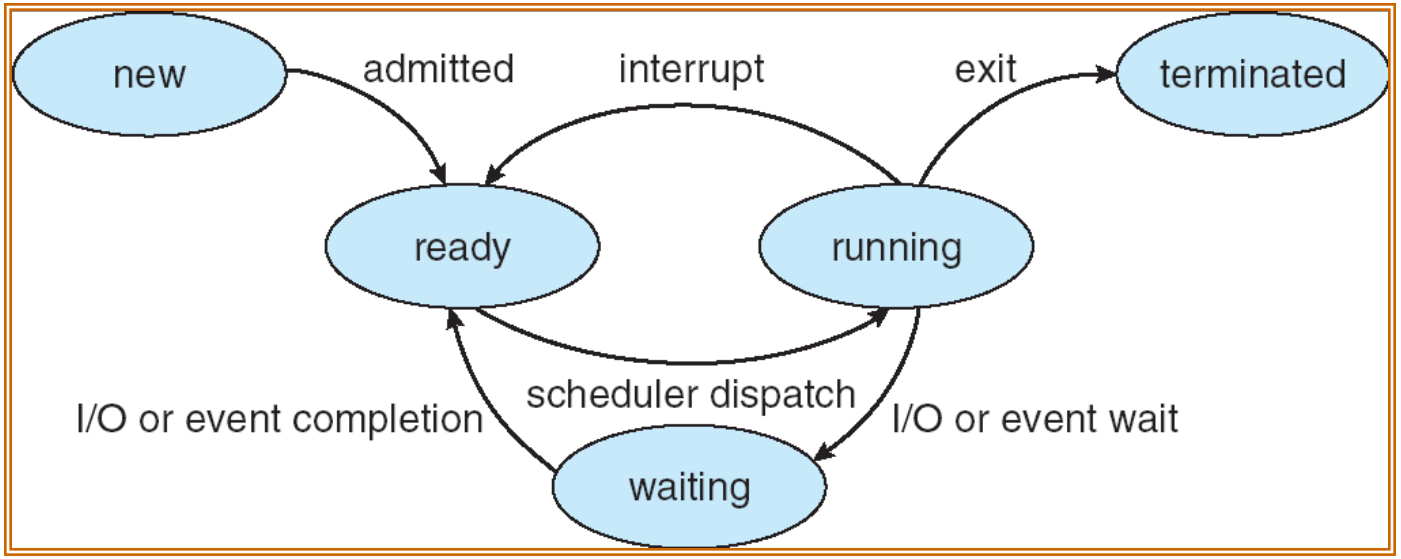
Multithreaded Processes

- PCB points to multiple TCBs:



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

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

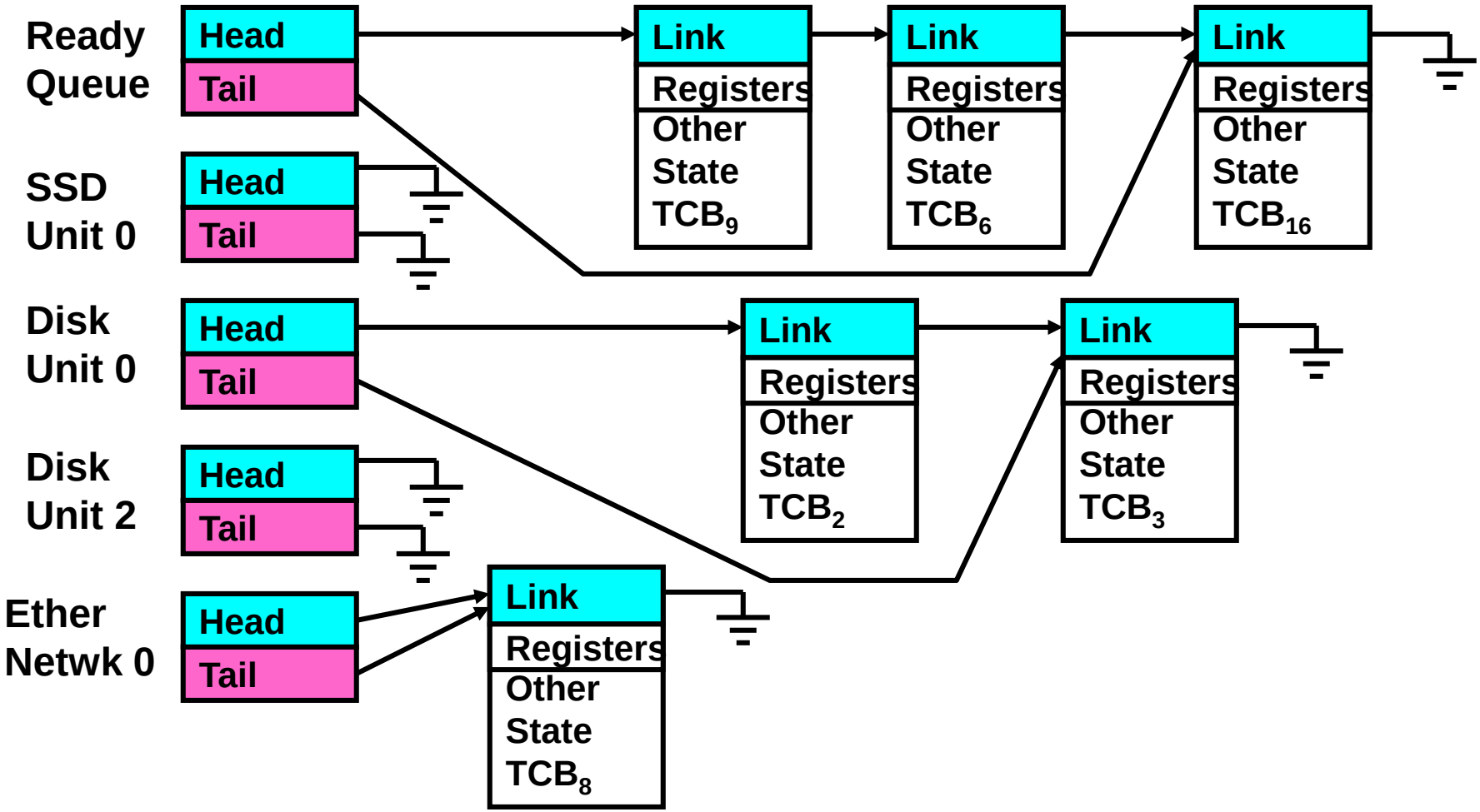
Ready Queues

- Note because of the actual number of live threads in a typical OS, and the (much smaller) number of running threads, most threads will be in a “ready” state.

- Thread not running $\Rightarrow$ TCB is in some scheduler queue

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



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

Per Thread State

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

- OS Keeps track of TCBs in protected memory
  - In Array, or Linked List, or …

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

Running a thread

Consider first portion: `RunThread()`

- How do I run a thread?
  - Load its state (registers, 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*

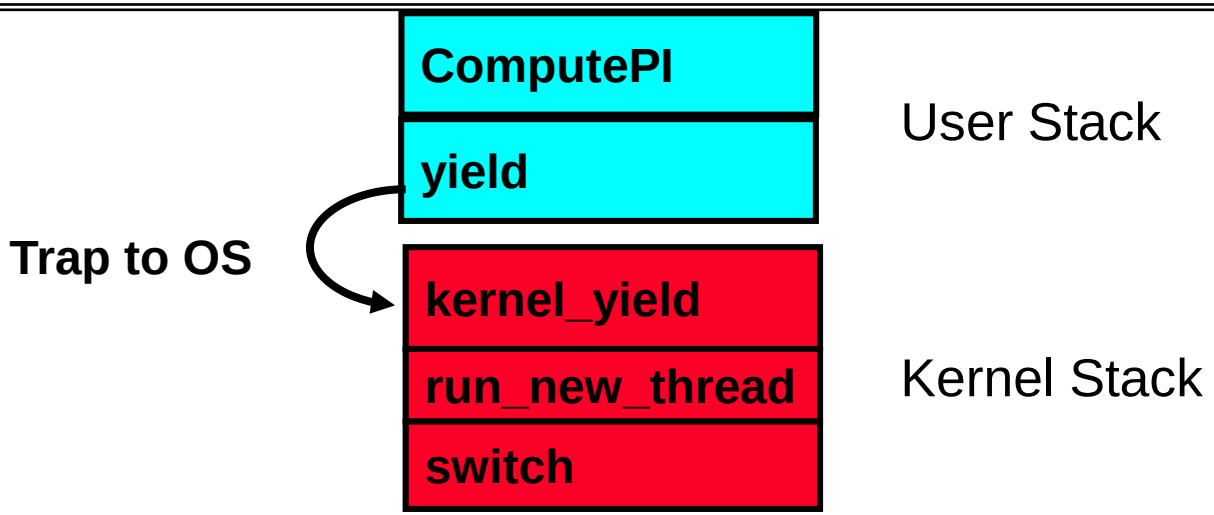
Yielding through Internal Events

- Blocking on I/O
  - The act of requesting I/O implicitly yields the CPU (e.g., printf)
- Waiting on a "signal" from other thread (e.g., join)
  - Thread asks to wait and thus yields the CPU
- Thread executes a `yield()`
  - Thread volunteers to give up CPU

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

  - Note that `yield()` must be called by programmer frequently enough!

Stack for Yielding a Thread

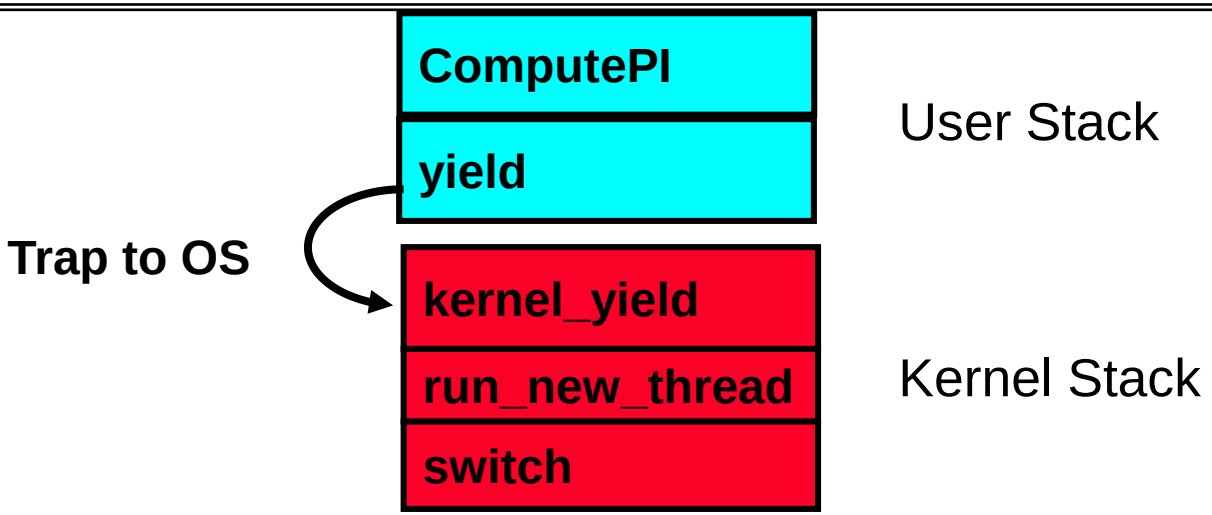


- How do we run a new thread?

```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* deallocates finished
                             threads */
}
```

- Finished thread not killed right away.
  - Move them in "exit/terminated" state
  - `ThreadHouseKeeping()` deallocates finished threads

Stack for Yielding a Thread



- How do we run a new thread?

```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* deallocates finished
                             threads */
}
```

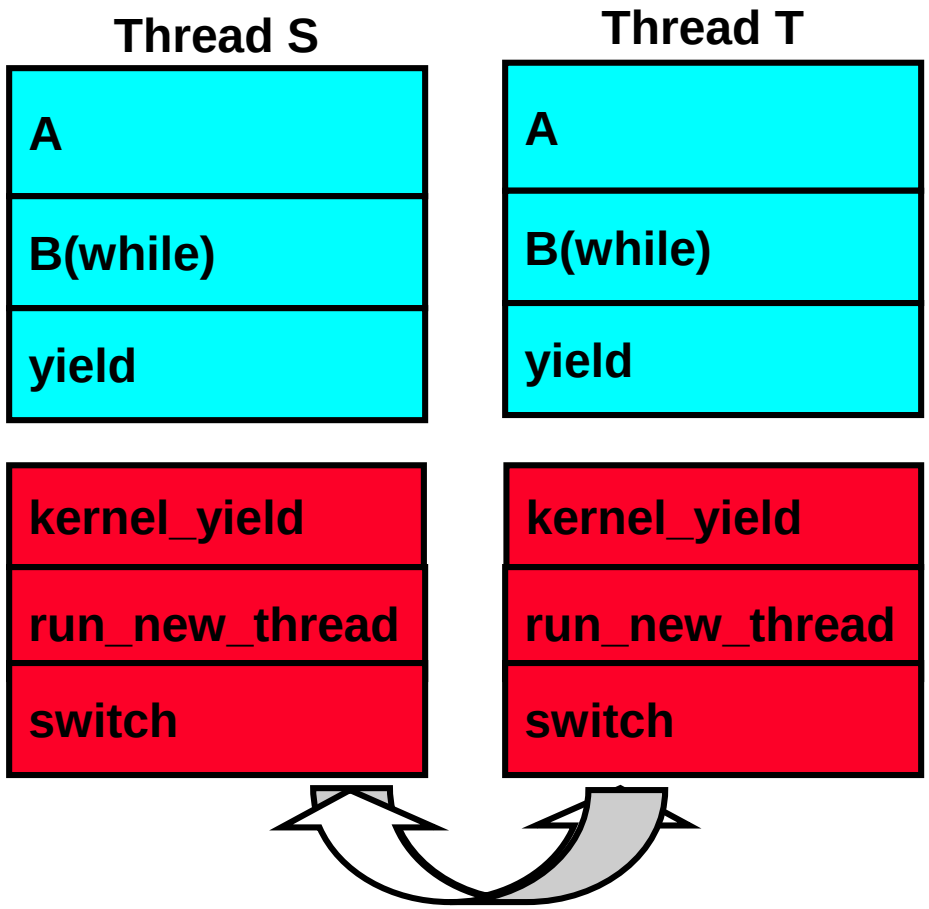
- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, SP
  - Maintain isolation for each thread

Two Thread Yield Example

- Consider the following code blocks:

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

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

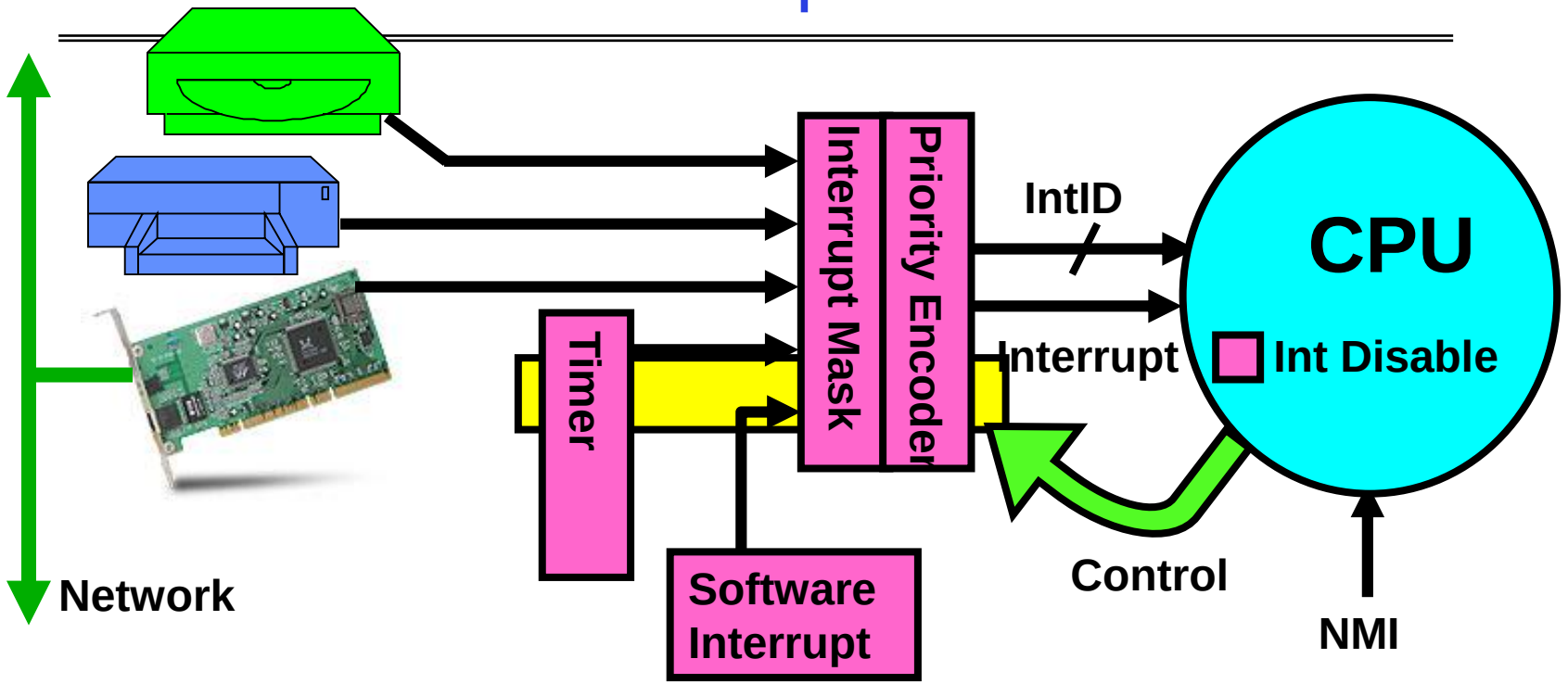
- Suppose we have two threads:
  - Threads S and T



Need for 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

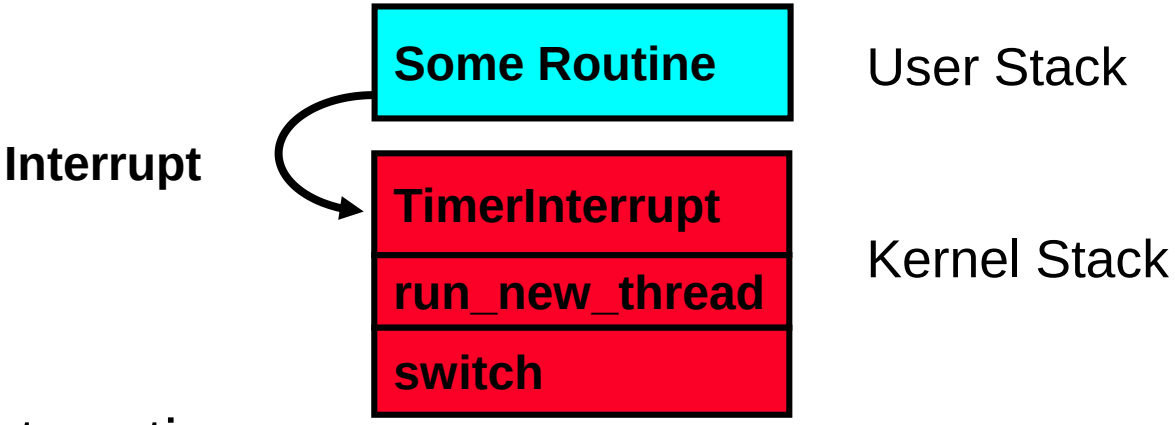
Detour: Interrupt Controller



- 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

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();`

Two important functions

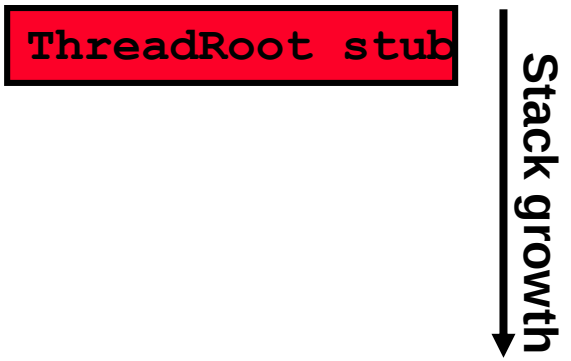
```
ThreadFork(aNewThread, null);

ThreadJoin(tid);
```

`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).

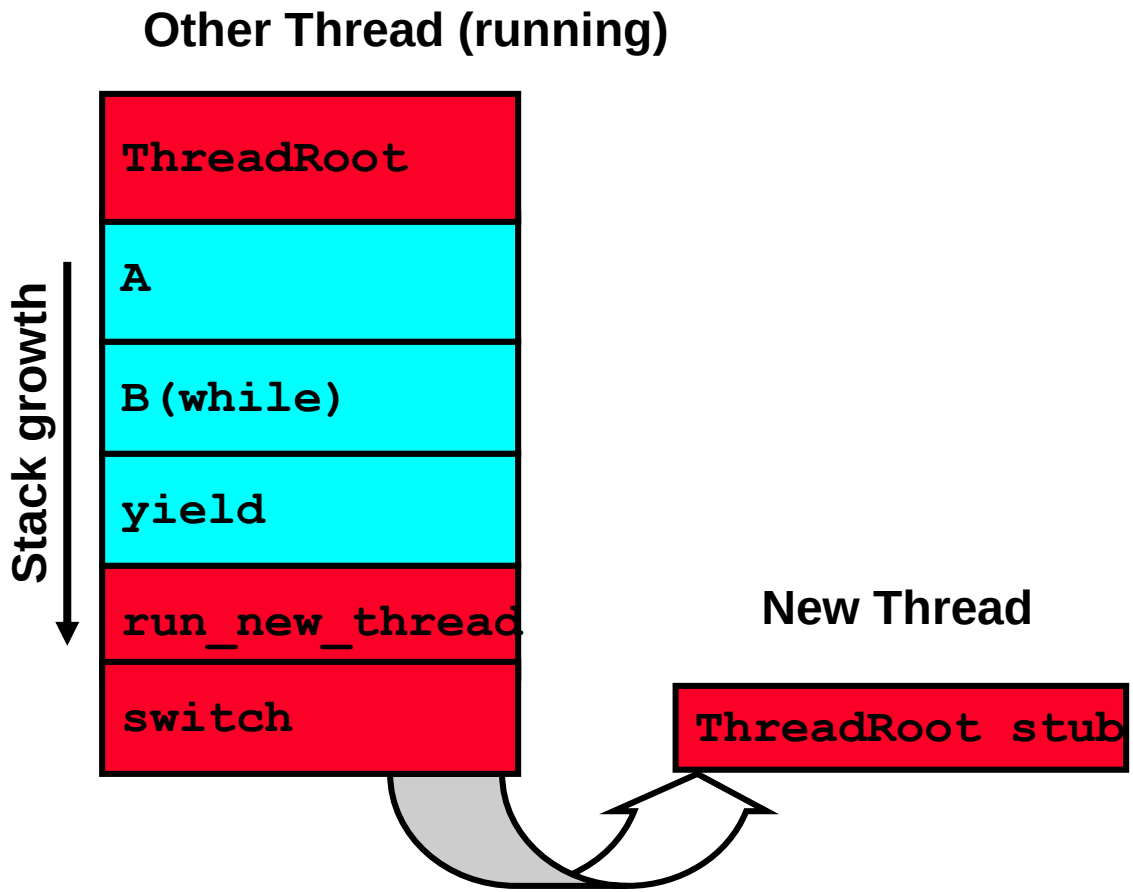
How do we initialize TCB and Stack?

- Initialize (4) Register fields of TCB (e.g., MIPS)
  - Stack pointer (sp) made to point at stack
  - PC return address (ra) set to OS (asm) routine `ThreadRoot()`
  - Two argument registers (a0 and a1) initialized to `fcnPtr` and `fcnArgPtr`, respectively
- Initialize stack data?
  - No. Important part of stack frame is in registers (ra)



Initial Stack

How does Thread get started?



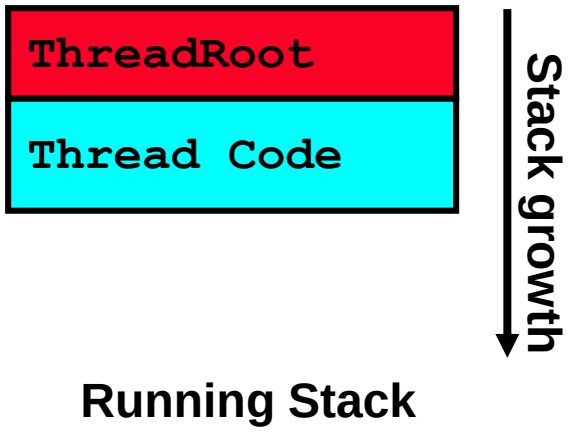
- Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`
  - This really starts the new thread

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! A thread cannot deallocate itself while running!
  - 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

`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`

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

Summary

- The state of a thread is contained in the TCB
  - Registers, PC, stack pointer
  - States: New, Ready, Running, Waiting, or Terminated
- 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()`
- Many scheduling options
  - Decision of which thread to run complex enough for complete lecture

Next

- Reading: Chapter 5.