SE350: Operating Systems

Lecture 4: Concurrency

Feedback

https://forms.gle/L6oS18zZApNF3ERb8

- Will be available until the end of term
- Will be checked regularly

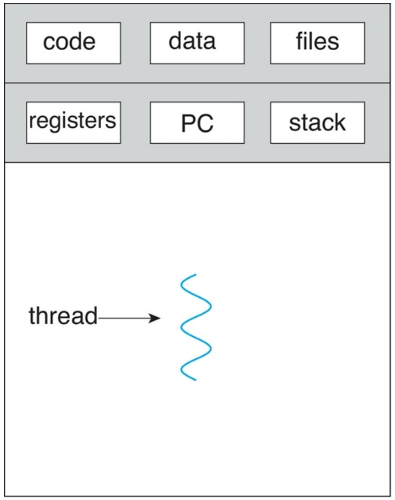
Outline

- Multi-threaded processes
- Thread data structure and life cycle
- Simple thread API
- Thread implementation

Recall: Traditional UNIX Process

- Process is OS abstraction of what is needed to run single program
  - Often called "heavyweight process"

- Processes have two parts
  - Sequential program execution stream (active part)
    - Code executed as sequential stream of execution (i.e., thread)
    - Includes state of CPU registers
  - Protected resources (passive part)
    - Main memory state (contents of Address Space)
    - I/O state (i.e. file descriptors)

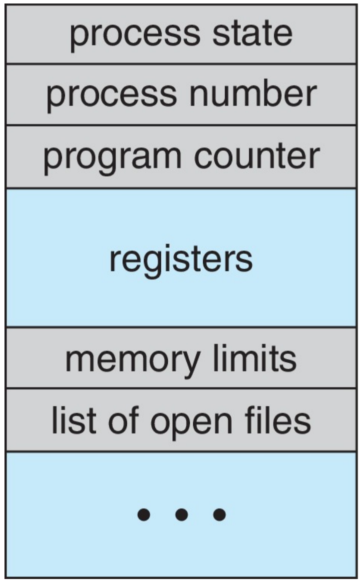


single-threaded process

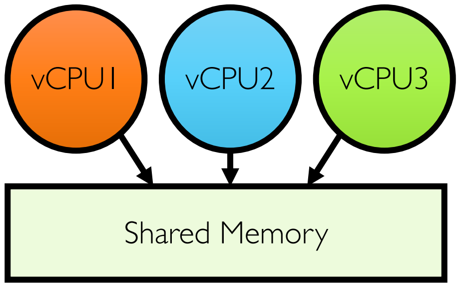
Process Control Block (PCB)

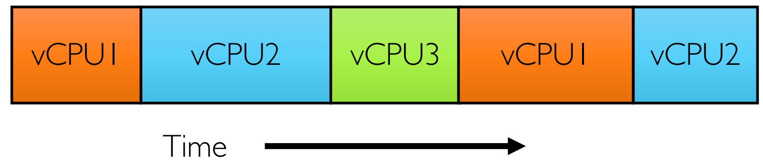
(Assume single threaded processes for now)

- OS represents each process as process control block (PCB)
  - Status (running, ready, blocked, …)
  - Registers, SP, … (when not running)
  - Process ID (PID), user, executable, priority, …
  - Execution time, …
  - Memory space, translation tables, …



Recall: Time Sharing





- How can we give illusion of multiple processors with single processor?
  - Multiplex in time!
- Each virtual "CPU" needs structure to hold PCBs
  - PC, SP, and rest of registers (integer, floating point, …)
- How do we switch from one vCPU to next?
  - Save PC, SP, and registers in current PCB
  - Load PC, SP, and registers from new PCB
- What triggers switch?
  - Timer, voluntary yield, I/O, …

How Do We Multiplex Processes?

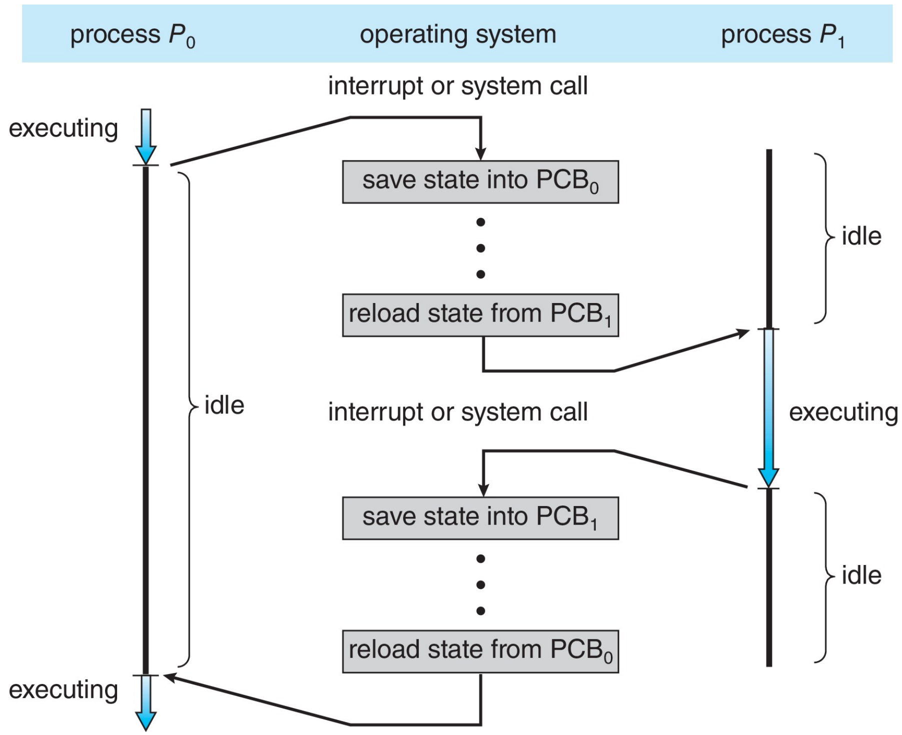
- Current state of process is held in PCB
  - This is "snapshot" of execution and protection environment
  - Only one PCB active at a time (for single-CPU machines)
- OS decides which process uses CPU time (scheduling)
  - Only one process is "running" at a time
  - Scheduler gives more time to important processes
- OS divides resources between processes (protection)
  - This provides 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

Scheduling

- Kernel scheduler decides which processes/threads receive CPU
- There are lots of different scheduling policies providing …
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or …
- Kernel Scheduler maintains data structure containing PCBs

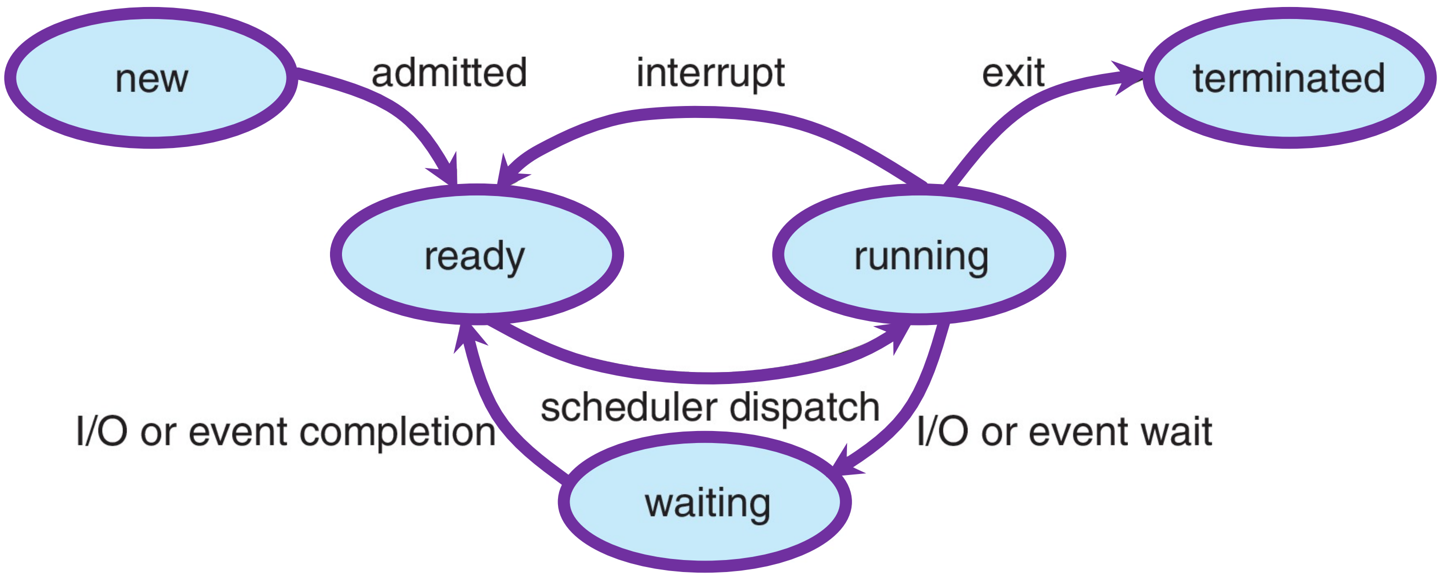
```
if (readyProcesses(PCBs)) {
     nextPCB = selectProcess(PCBs);
     run(nextPCB);
} else {
     run_idle_process();
}
```

Context Switch: CPU Switch Between Two Processes



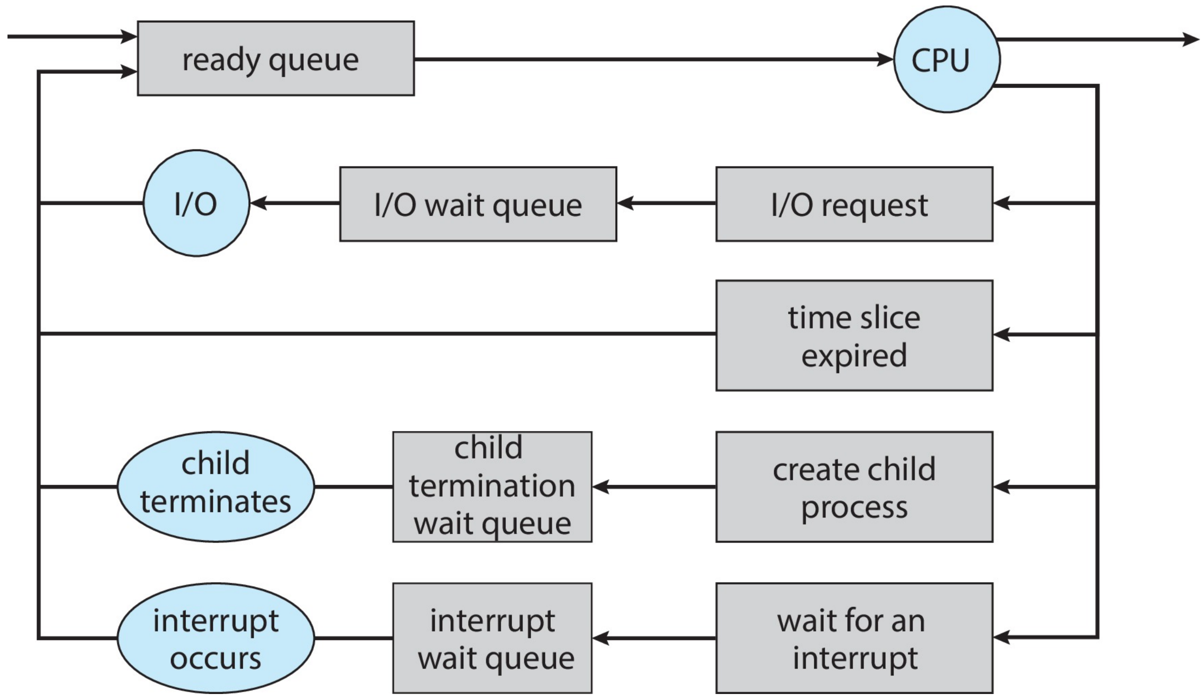
- Code executed in kernel is overhead
    - Overhead sets minimum practical switching time
    - Less overhead with SMT/hyperthreading, but … contention for resources

Lifecycle of Processes



- As process is executed, its state changes
  - New: Process is being created
  - Ready: Process is waiting to run
  - Running: Instructions are being executed
  - Waiting: Process waiting for some event to occur
  - Terminated: Process has finished execution

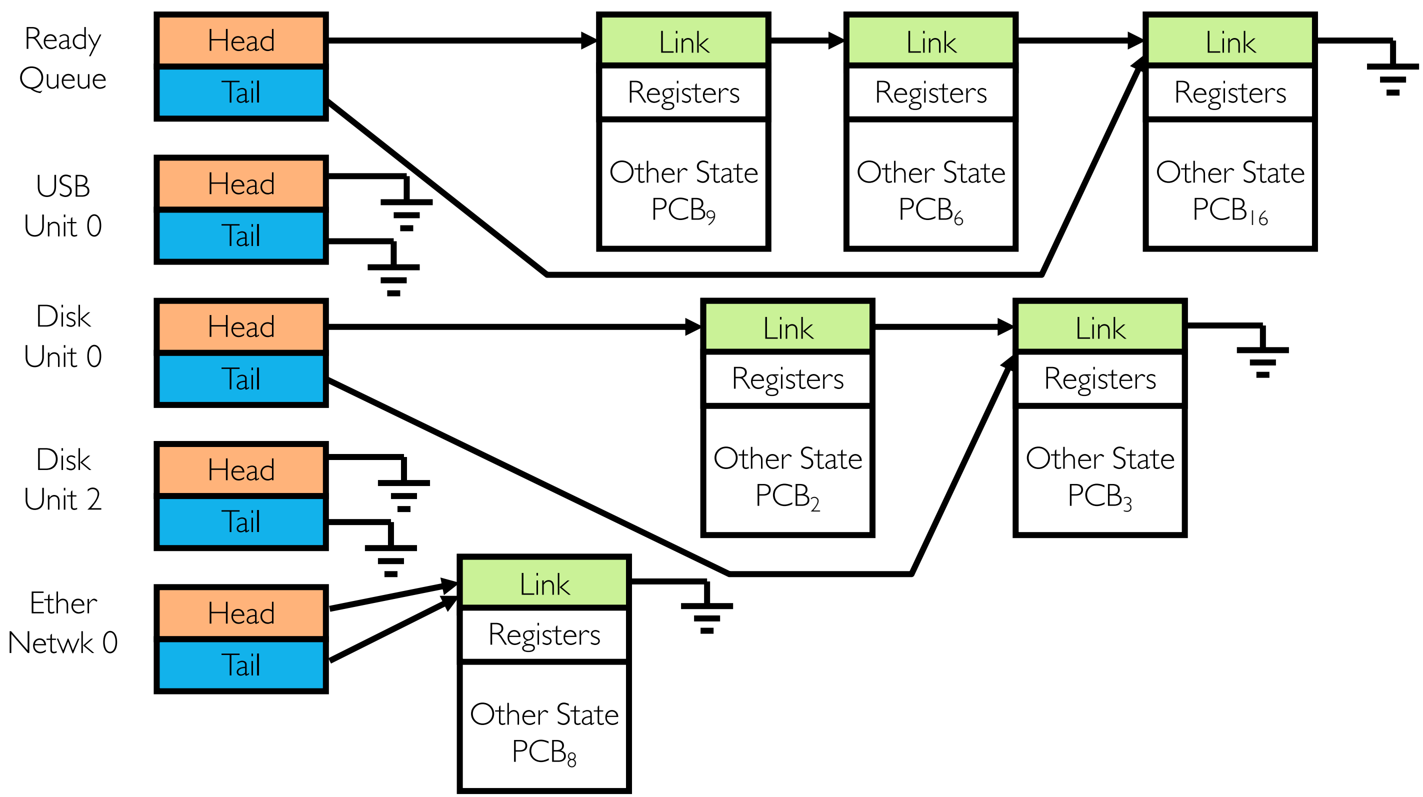
Ready Queue



- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are scheduling decisions
  - Many algorithms possible (more on this in a few weeks)

Ready Queue And I/O Device Queues

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



Drawback of Traditional UNIX Process

- Silly example:

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

- Would program ever print out class list?
  - No! `ComputePI` would never finish!
- Better example:

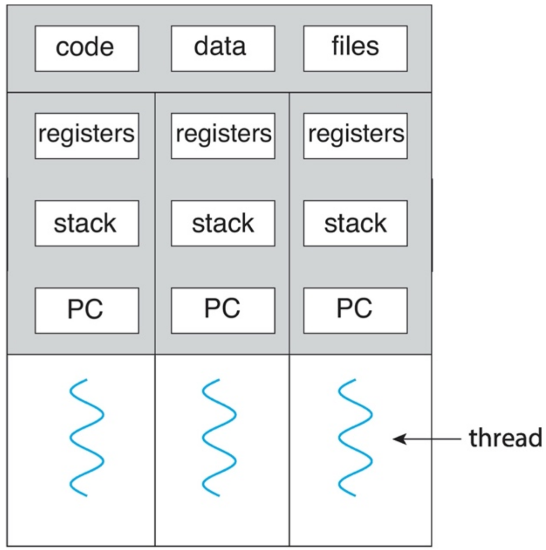
```
main() {
    ReadLargeFile("pi.txt");
    RenderUserInterface();
}
```

Threads Motivation

- OS's need to handle multiple things at once (MTAO)
  - Processes, interrupts, background system maintenance
- Servers need to handle MTAO
  - Multiple connections handled simultaneously
- Parallel programs need to handle MTAO
  - To achieve better performance
- Programs with user interfaces often need to handle MTAO
  - To achieve user responsiveness while doing computation
- Network and disk programs need to handle MTAO
  - To hide network/disk latency

Modern Process with Threads

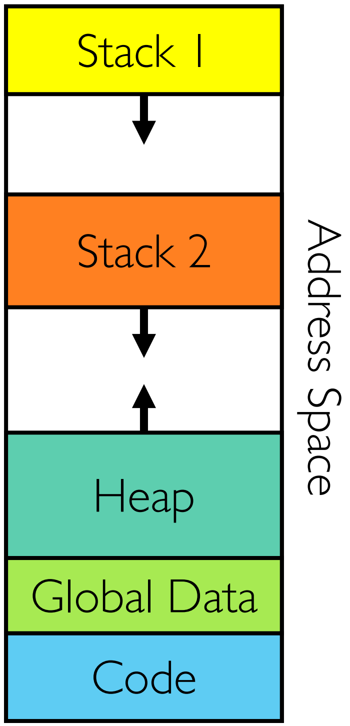
- Thread: sequential execution stream within process (sometimes called "lightweight process")
  - Process still contains single address space
  - No protection between threads
- Multithreading: single program made up of different concurrent activities (sometimes called multitasking)
- Some states are shared by all threads
  - Content of memory (global variables, heap)
  - I/O state (file descriptors, network connections, etc.)
- Some states "private" to each thread
  - CPU registers (including PC) and stack



multithreaded process

A Side Note: Memory Footprint of Multiple Threads

- How do we position stacks relative to each other?
- What maximum size should we choose for stacks?
  - 8KB for kernel-level stacks in Linux on x86
  - Less need for tight space constraint for user-level stacks
- What happens if threads violate this?
  - "… program termination and/or corrupted data"
- How might you catch violations?
  - Place guard values at top and bottom of each stack
  - Check values on every context switch



Per Thread Descriptor (Kernel Supported Threads)

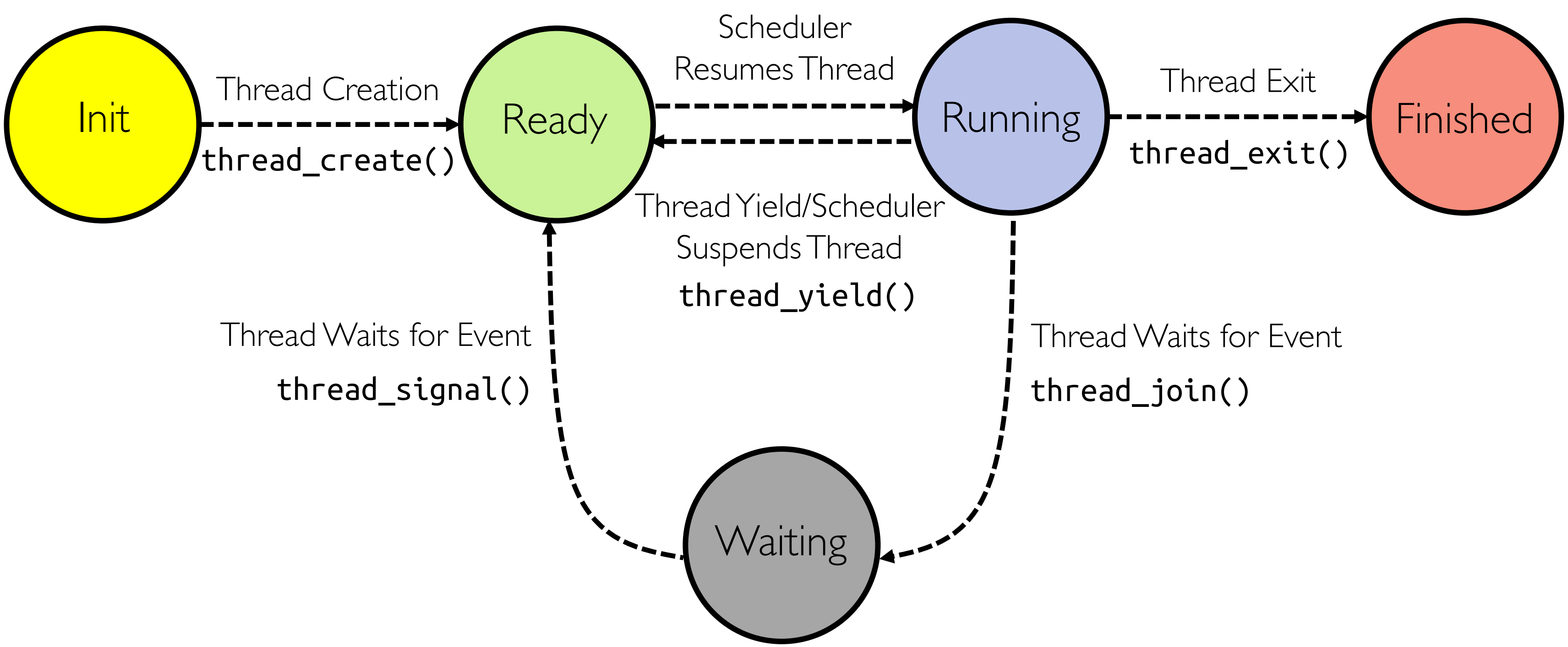
- Each thread has Thread Control Block (TCB)
  - Execution State
    - CPU registers, program counter (PC), pointer to stack (SP)
  - Scheduling info
    - State, priority, CPU time
  - Various pointers (for implementing scheduling queues)
  - Pointer to enclosing process (PCB) – user threads
  - … (add stuff as you find a need)
- OS Keeps track of TCBs in "kernel memory"
  - In array, or linked list, or …

Simple Thread API

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

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

Thread Lifecycle



Use of Threads

- Rewrite program with threads (*loose syntax*)

```
main() {
    thread_t threads[2];
    thread_create(&threads[0], &ComputePI, “pi.txt”);
    thread_create(&threads[1], &PrintClassList, “class.txt”);
}
```

- What does `thread_create` do?
  - Creates independent thread
  - Behaves as if there are two separate CPUs

Dispatch Loop

- Conceptually, dispatching loop of OS looks as follows

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

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

Running Threads

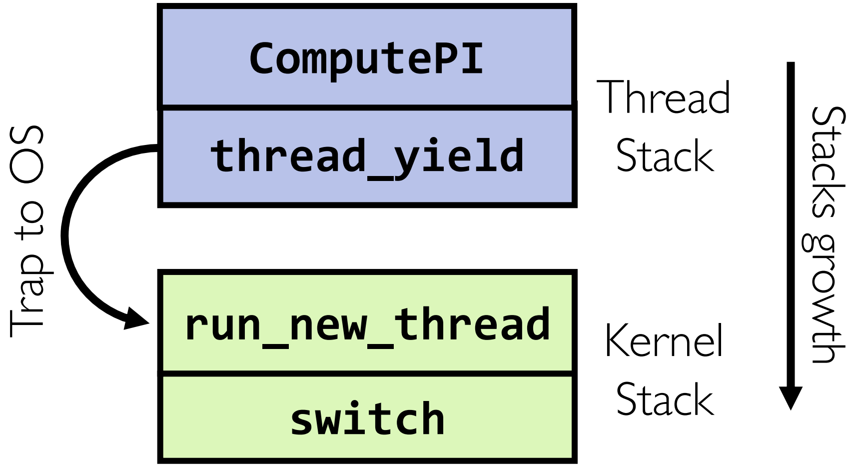
- What does `LoadStateOfCPU()` do?
  - Loads thread's state (registers, PC, stack pointer) into CPU
  - Loads environment (virtual memory space, etc.)
- What does `RunThread()` do?
  - Jump to PC
- How does dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets preempted

Internal Events

- Blocking on I/O
  - Requesting I/O implicitly yields CPU
- Waiting on "*signal*" from other thread
  - Thread asks to wait and thus yields CPU
- Thread executes `thread_yield()`
  - Thread volunteers to give up CPU

```
ComputePI() {
    while(TRUE) {
        ComputeNextDigit();
        thread_yield();
    }
}
```

Stack for Yielding Thread



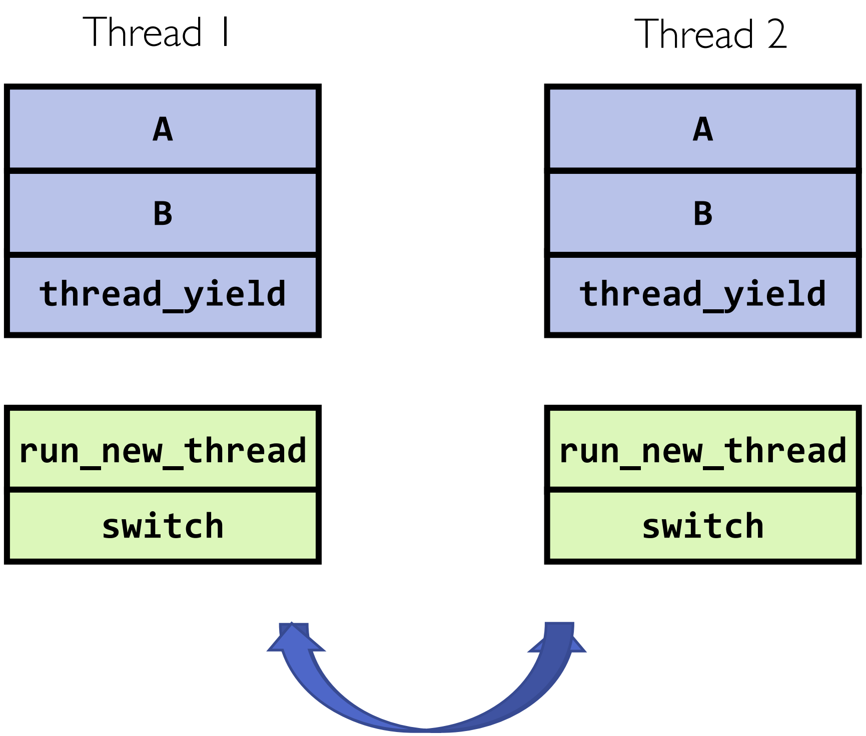

```
run_new_thread() {
    newTCB = PickNewThread();
    switch(curTCB, newTCB);
    thread_house_keeping(); /* Do any cleanup */
}
```

How Do Stacks Look Like?

- Suppose we have 2 threads

```
A() {
    B();
}

B() {
    while(TRUE) {
        thread_yield();
    }
}
```




```
run_new_thread() {
    newThread = PickNewThread();
    switch(curTCB, newTCB);
    thread_house_keeping(); /* Do any cleanup */
}
```

Saving/Restoring State: Context Switch

```
// We enter as curTCB, but we return as newTCB
// Returns with newTCB's registers and stack

switch(curTCB, newTCB) {
    pushad;                 // Push regs onto kernel stack for curTCB
    curTCB->sp = sp;        // Save curTCB's stack pointer
    sp = newTCB->sp;        // Switch to newTCB's stack
    popad;                  // Pop regs from kernel stack for newTCB
    return();
}
```

Where does this return to?

Switch Details

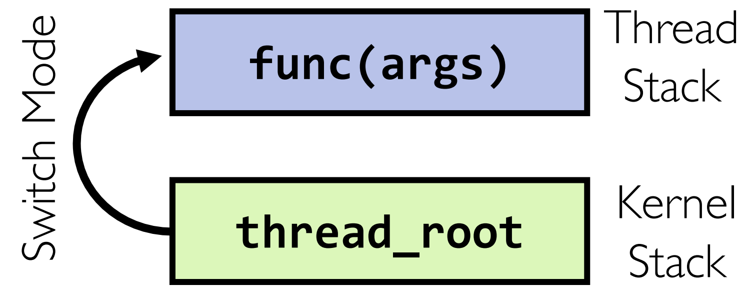
- What if you make mistakes in implementing switch?
  - Suppose you forget to save/restore register 32
  - Get intermittent failures depending on when context switch occurred and whether new thread uses register 32
  - System will give wrong result without warning
- Can you devise exhaustive test to test switch code?
  - No! Too many combinations and inter-leavings

Creating New Threads

- Implementation
  - Sanity check arguments and copy them to kernel memory
  - Enter Kernel-mode and sanity check arguments again
  - Allocate new stack and TCB
  - Initialize TCB
  - Place new TCB on ready list (runnable)
- How do we initialize TCB and stack?
  - `newTCB->sp` points to newly allocated stack
  - `newTCB->pc` points to OS routine `thread_root()`
  - Push `func` and `args` pointers into stack
  - Call `dummy_switch_frame(newTCB)` (more on this soon)

How Does thread_root() Look Like?

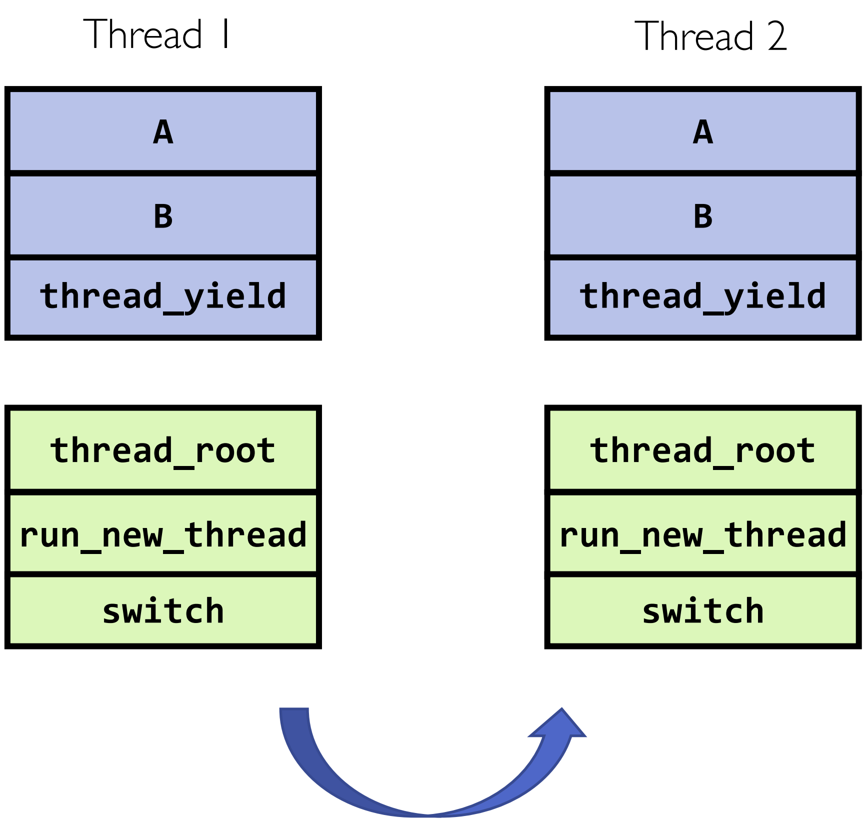
```
thread_root(func*, args*) {
    DoStartupHousekeeping();
    UserModeSwitch();  // enter user mode */
    Call func(args);
    thread_finish();
}
```



- 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 **thread_root()** which calls **thread_finish()** which wakes up sleeping threads

Putting it All Together

- Eventually, `run_new_thread` will select newly created TCB and return into beginning of `thread_root`
  - This really starts the new thread



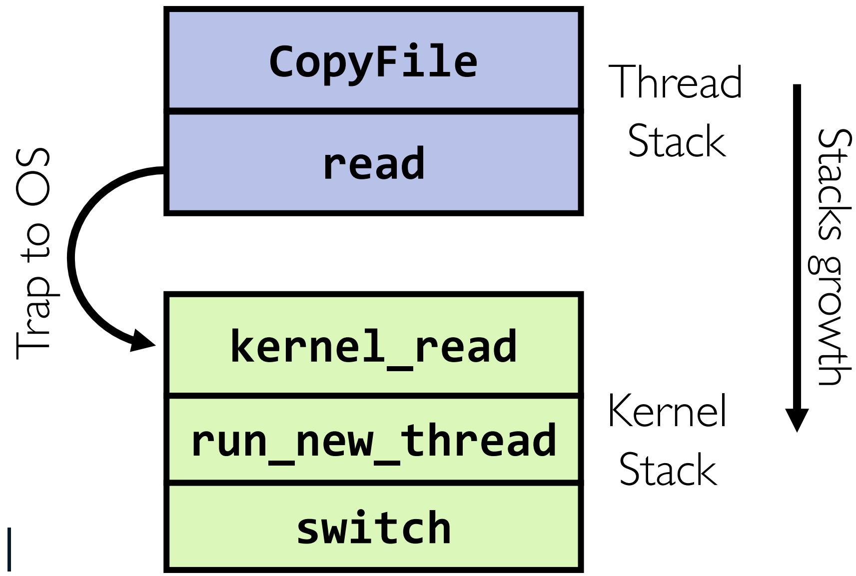
A Subtlety: `dummy_switch_frame(newTCB)`

- Newly created thread will run after OS runs **`switch`**
- Kernel stack of new thread should be the same as others
- Recall:

```
switch(curTCB, newTCB) {
    pushad;
    curTCB->sp = sp;
    sp = newTCB->sp;
    popad;
    return();
}

dummy_switch_frame(newTCB) {
    *(newTCB->sp) = thread_root;
    newTCB->sp--;
    newTCB->sp -= SizeOfPopad;
}
```

What Happens When Threads Blocks on I/O?



- User code invokes system call
- Read operation is initiated
- OS runs new thread or switches to ready thread

Recall: Running Threads

- What does `LoadStateOfCPU()` do?
  - Loads thread's state (registers, PC, stack pointer) into CPU
  - Loads environment (virtual memory space, etc.)
- What does `RunThread()` do?
  - Jump to PC
- How does dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets preempted

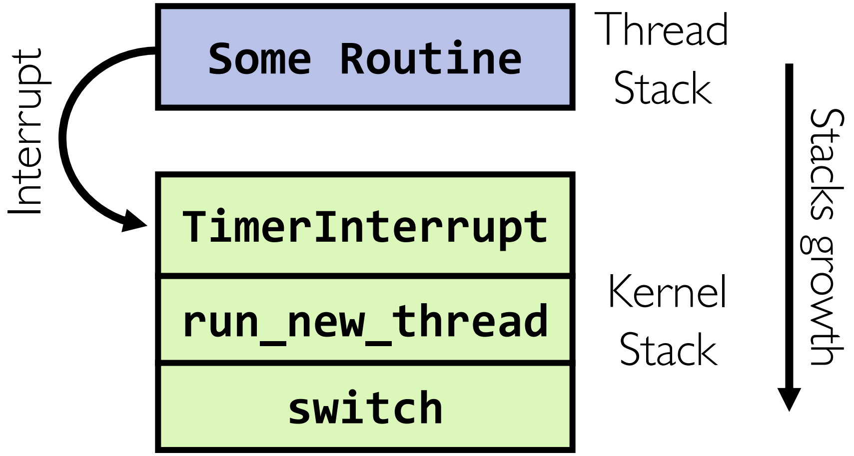
External Events

- What happens if thread never does any I/O, never waits, and never yields?
- Could `ComputePI` grab all resources and never release processor?
  - Must find way that dispatcher can regain control!
- OS utilizes external events
- Interrupts are signals from hardware or software that stop running code and transfer control to kernel
  - E.g., timer is like alarm clock that goes off every some milliseconds
- Interrupts are hardware-invoked context switch
- Interrupt handlers are not threads
  - No separate step to choose what to run next
  - Always run the interrupt handler immediately

Timer Interrupt to Return Control

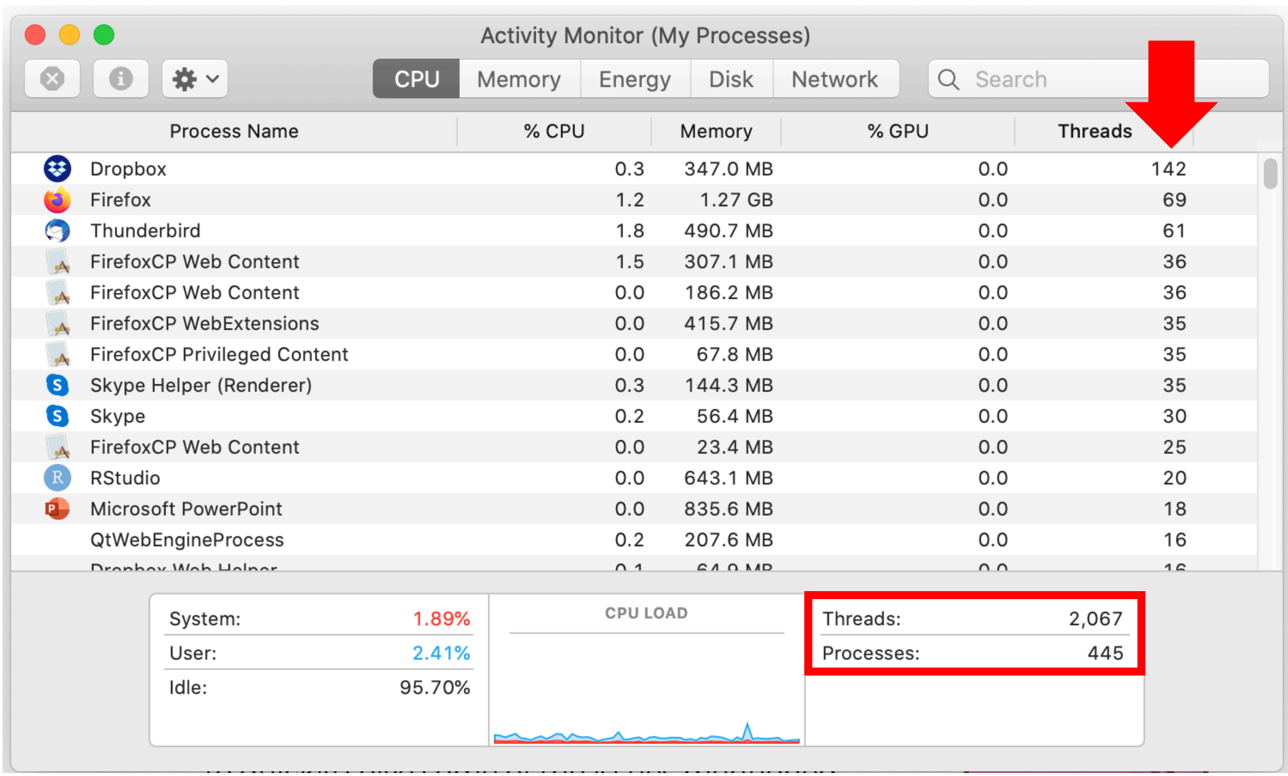
- Solution to our dispatcher problem
  - Use the timer interrupt to force scheduling decisions

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



Some Numbers

- Many process are multi-threaded, so thread context switches may be either within-process or across-processes



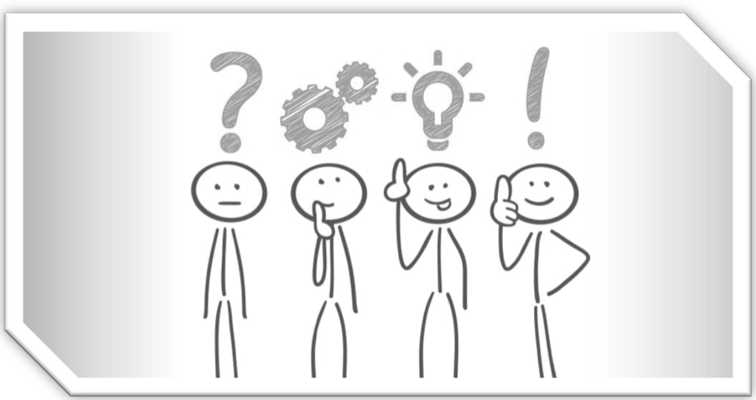
Some Numbers (cont.)

- Frequency of performing context switches is ~10-100ms
- Context switch time in Linux is ~3-4 us (Intel i7 & Xeon E5)
  - Thread switching faster than process switching (~100 ns)
- Switching across cores is ~2x more expensive than within-core
- Context switch time increases sharply with size of working set*
  - Can increase ~100x or more

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

* Working set is subset of memory used by process in time window

Questions?



Acknowledgment

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny