CS162
Operating Systems and Systems Programming
Lecture 6

Synchronization 1: Concurrency and Mutual Exclusion

February 2nd, 2023

Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

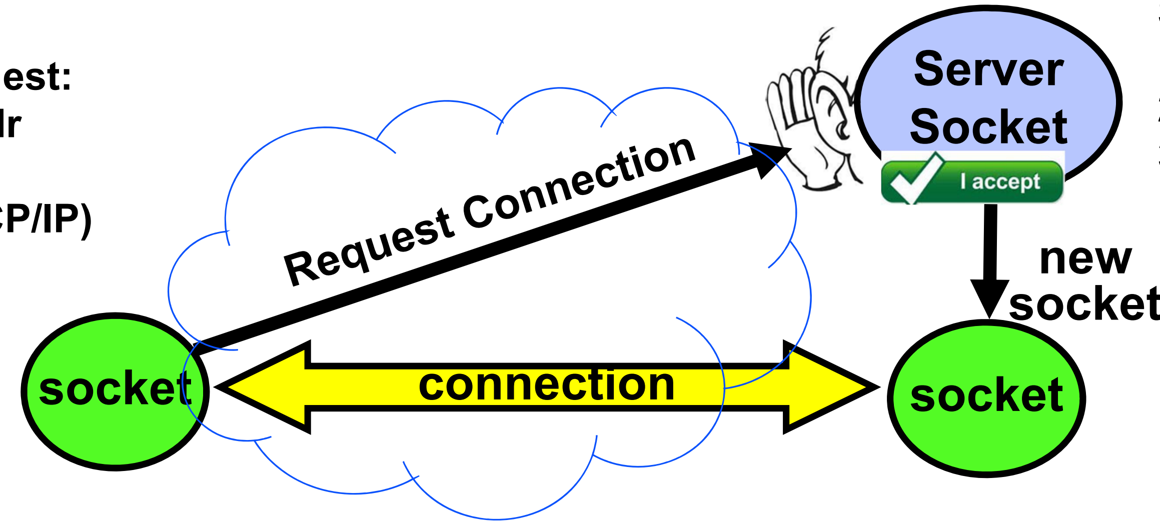
Recall: Connection Setup over TCP/IP

Client Side

Server Side

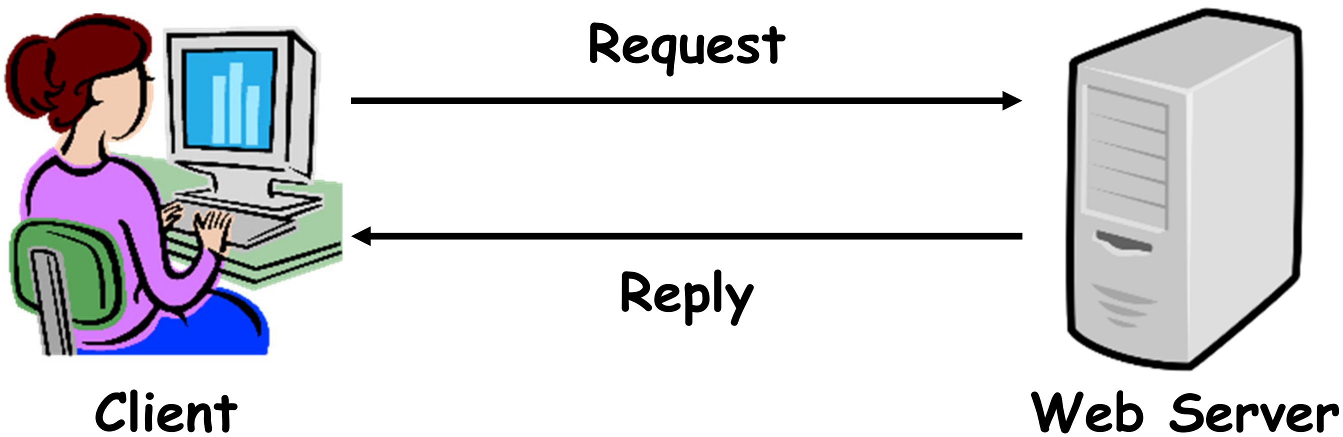
Connection request:
1. **Client IP addr**
2. **Client Port**
3. **Protocol (TCP/IP)**

Server Listening:
1. **Server IP addr**
2. **well-known port,**
3. **Protocol (TCP/IP)**

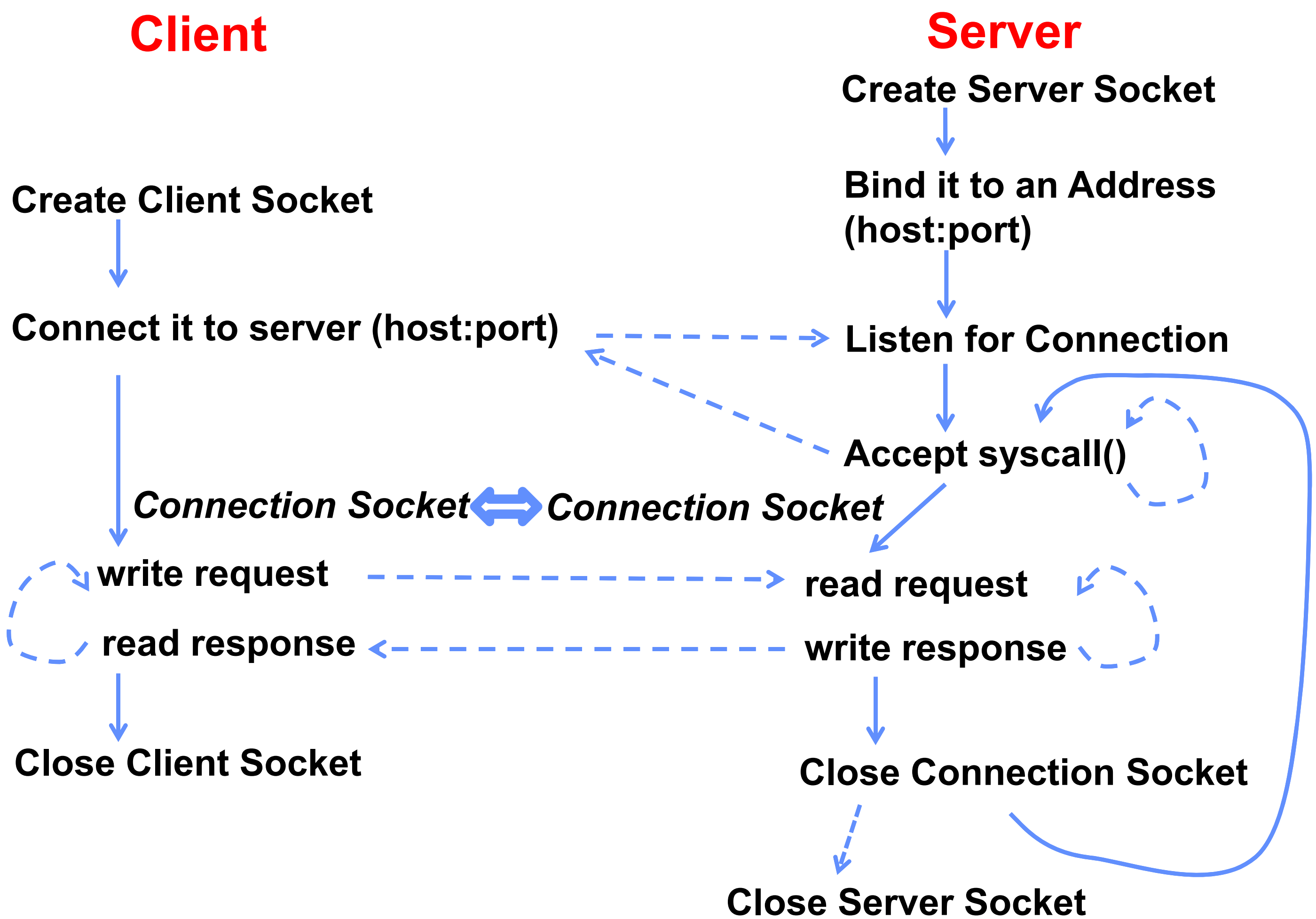


- 5-Tuple identifies each connection:
  1. Source IP Address
  2. Destination IP Address
  3. Source Port Number
  4. Destination Port Number
  5. Protocol (always TCP here)
- Often, Client Port “randomly” assigned
  – Done by OS during client socket setup
- Server Port often “well known”
  – 80 (web), 443 (secure web), 25 (sendmail), etc
  – Well-known ports from 0—1023

Recall: Web Server



Recall: Sockets in concept



Recall: Client Protocol

```
char *host_name, *port_name;

// Create a socket
struct addrinfo *server = lookup_host(host_name, port_name);
int sock_fd = socket(server->ai_family, server->ai_socktype,
                     server->ai_protocol);

// Connect to specified host and port
connect(sock_fd, server->ai_addr, server->ai_addrlen);

// Carry out Client-Server protocol
run_client(sock_fd);

/* Clean up on termination */
close(sock_fd);
```

Recall Client-Side: Getting the Server Address

```
struct addrinfo *lookup_host(char *host_name, char *port) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;          /* Includes AF_INET and AF_INET6 */
  hints.ai_socktype = SOCK_STREAM;      /* Essentially TCP/IP */

  int rv = getaddrinfo(host_name, port_name,
                       &hints, &server);
  if (rv != 0) {
    printf("getaddrinfo failed: %s\n", gai_strerror(rv));
    return NULL;
  }
  return server;
}
```

Recall: Server Protocol (v1)

```
// Create socket to listen for client connections
char *port_name;
struct addrinfo *server = setup_address(port_name);
int server_socket = socket(server->ai_family,
                           server->ai_socktype, server->ai_protocol);
// Bind socket to specific port
bind(server_socket, server->ai_addr, server->ai_addrlen);
// Start listening for new client connections
listen(server_socket, MAX_QUEUE);

while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  serve_client(conn_socket);
  close(conn_socket);
}
close(server_socket);
```

Recall: Server Address: Itself (wildcard IP), Passive

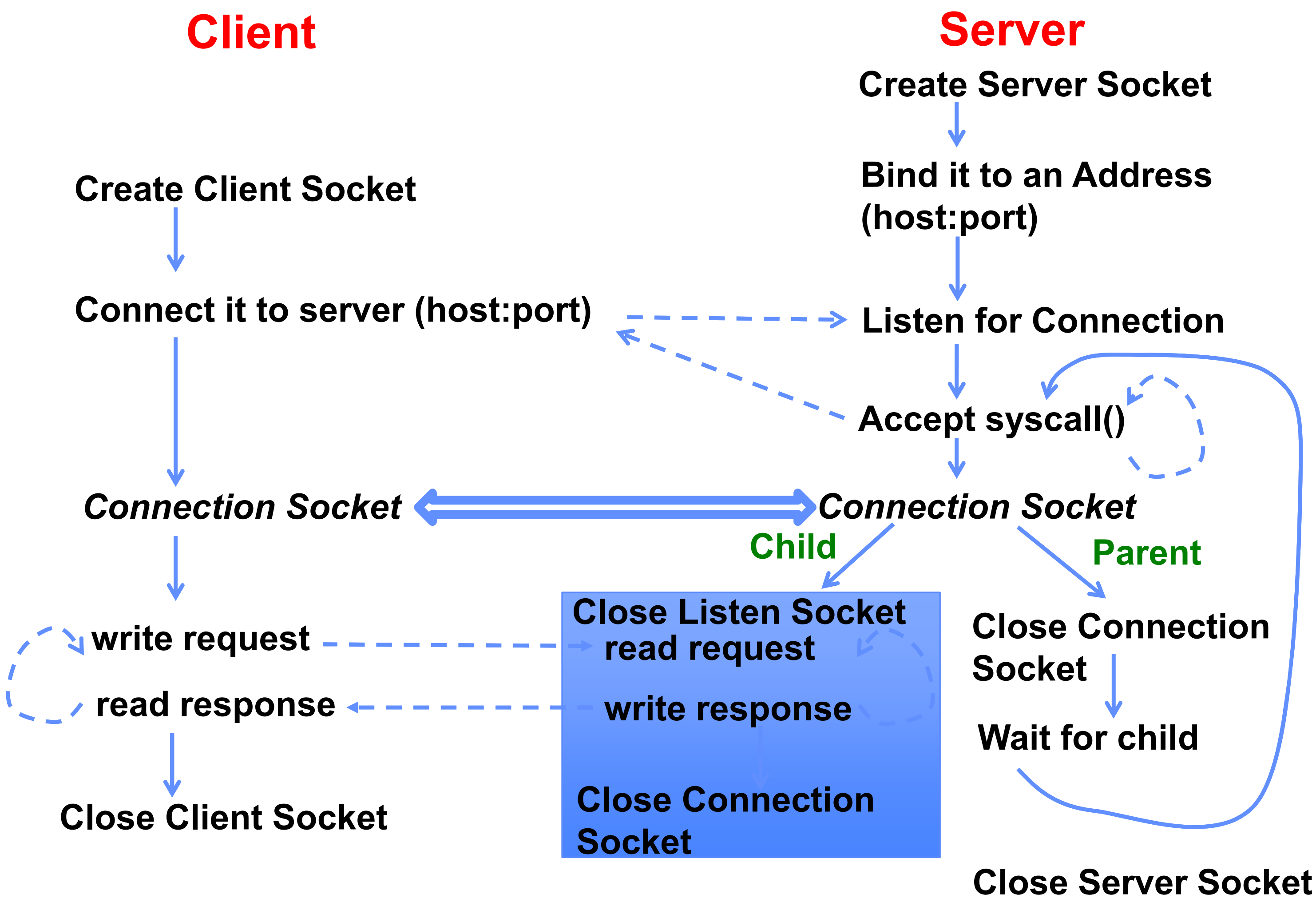
```
struct addrinfo *setup_address(char *port) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;          /* Includes AF_INET and AF_INET6 */
  hints.ai_socktype = SOCK_STREAM;      /* Essentially TCP/IP */
  hints.ai_flags = AI_PASSIVE;          /* Set up for server socket */

  int rv = getaddrinfo(NULL, port, &hints, &server);  /* No address! (any local IP) */
  if (rv != 0) {
    printf("getaddrinfo failed: %s\n", gai_strerror(rv));
    return NULL;
  }
  return server;
}
```

- **Accepts any connections on the specified port**

How Could the Server Protect Itself?

- Handle each connection in a separate process
  - This will mean that the logic serving each request will be "sandboxed" away from the main server process
- In the following code, keep in mind:
  - `fork()` will duplicate *all* of the parent's file descriptors (i.e. pointers to sockets!)
  - We keep control over accepting new connections in the parent
  - New child connection for each remote client

Sockets With Protection (each connection has own process)



Server Protocol (v2)

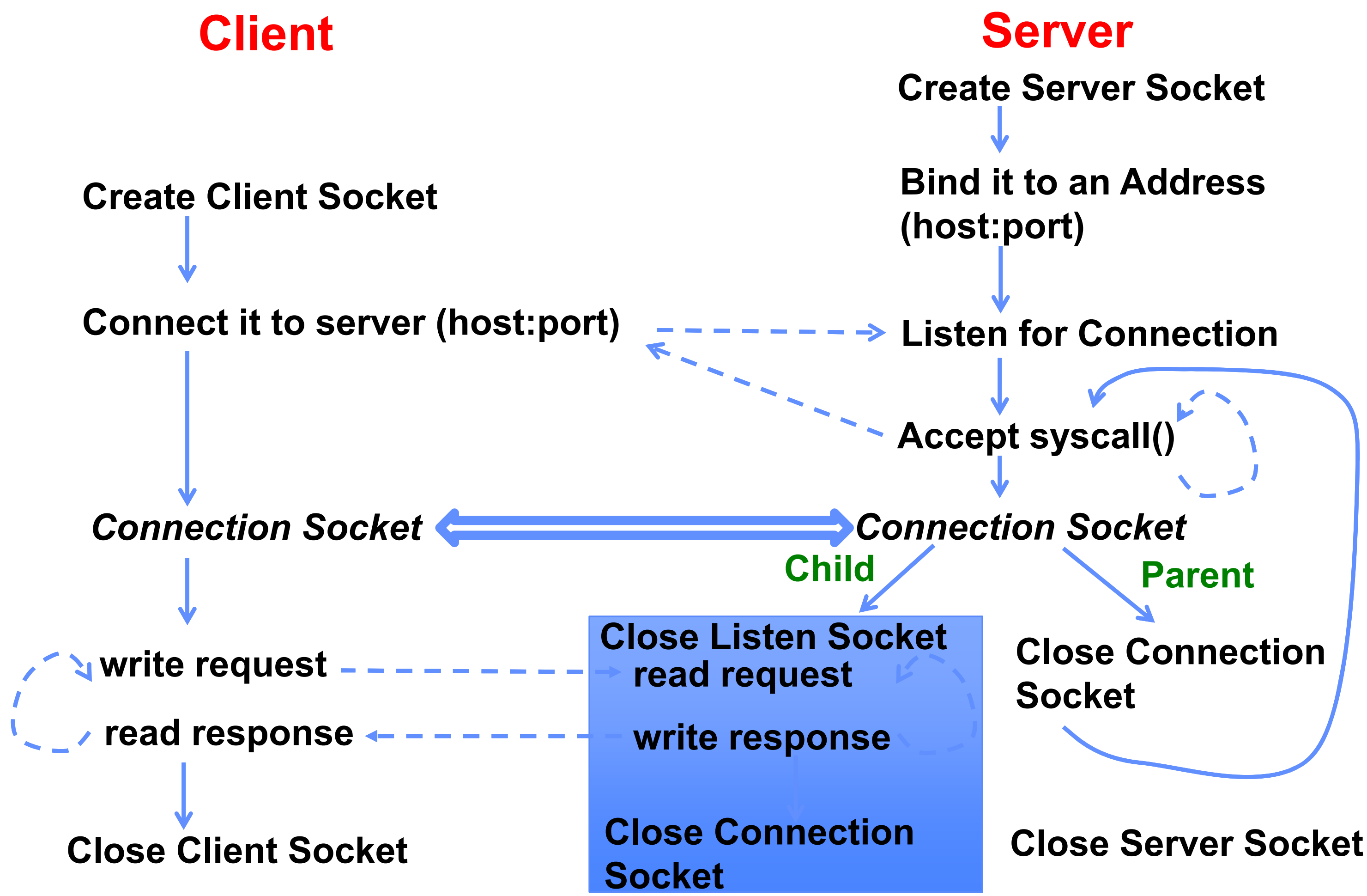
```
// Socket setup code elided…
listen(server_socket, MAX_QUEUE);
while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  pid_t pid = fork();
  if (pid == 0) {
    close(server_socket);
    serve_client(conn_socket);
    close(conn_socket);
    exit(0);
  } else {
    close(conn_socket);
    wait(NULL);
  }
}
close(server_socket);
```

How to make a Concurrent Server

- So far, in the server:
  - Listen will queue requests
  - Buffering present elsewhere
  - But server *waits* for each connection to terminate before servicing the next
    - » This is the standard shell pattern

- A concurrent server can handle and service a new connection before the previous client disconnects
  - Simple – just don't wait in parent!
  - Perhaps not so simple – multiple child processes better not have data races with one another through file system/etc!

Sockets With Protection and Concurrency



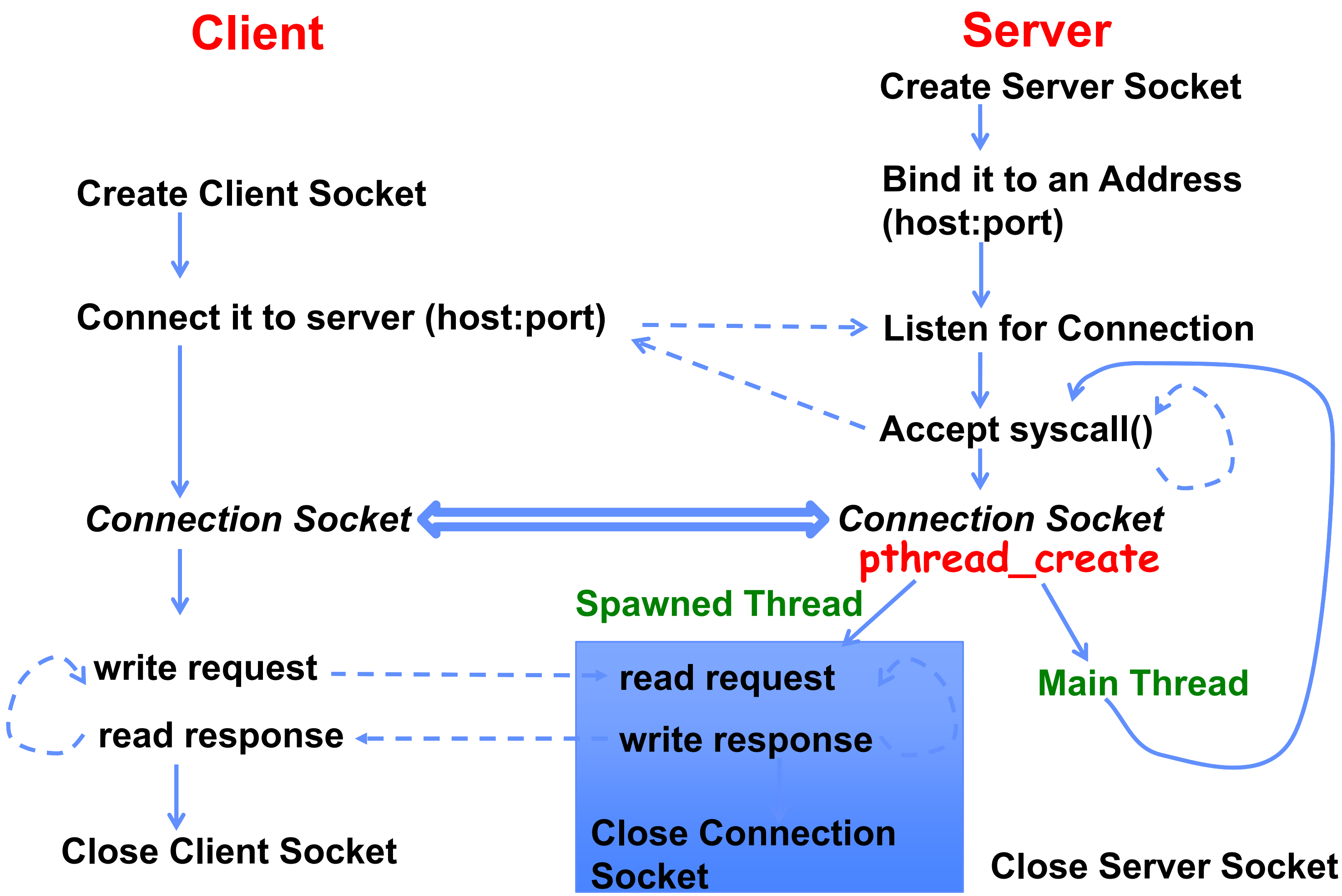
Server Protocol (v3)

```
// Socket setup code elided…
listen(server_socket, MAX_QUEUE);
while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  pid_t pid = fork();
  if (pid == 0) {
    close(server_socket);
    serve_client(conn_socket);
    close(conn_socket);
    exit(0);
  } else {
    close(conn_socket);
    //wait(NULL);
  }
}
close(server_socket);
```

Faster Concurrent Server (without Protection)

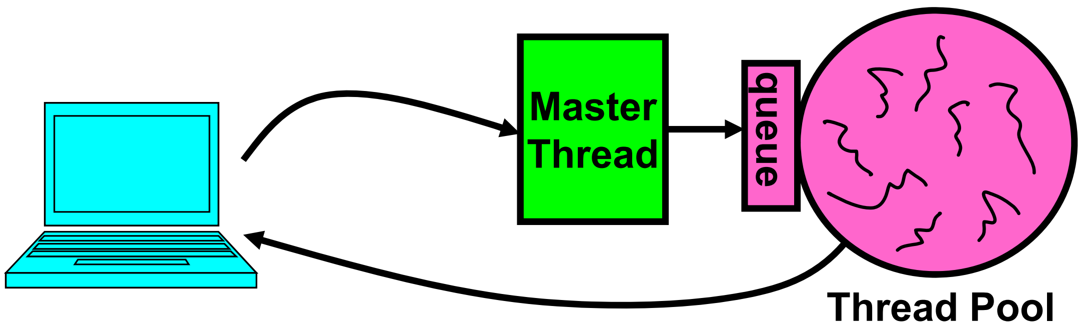
- Spawn a new *thread* to handle each connection
  - Lower overhead spawning process (less to do)
- Main *thread* initiates new client connections without waiting for previously spawned threads
- Why give up the protection of separate processes?
  - More efficient to create new threads
  - More efficient to switch between threads
- Even more potential for data races (need synchronization?)
  - Through shared memory structures
  - Through file system

Sockets with Concurrency, without Protection



Thread Pools: More Later!

- Problem with previous version: Unbounded Threads
  – When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded “pool” of worker threads, representing the maximum level of multiprogramming

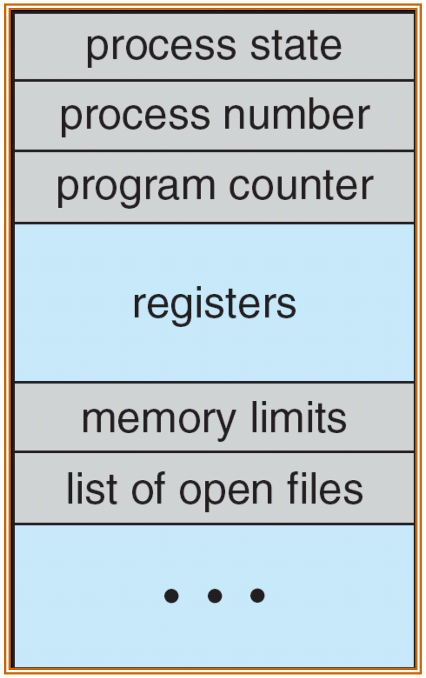



```
master() {
   allocThreads(worker,queue);
   while(TRUE) {
      con=AcceptCon();
      Enqueue(queue,con);
      wakeUp(queue);
   }
}
```

```
worker(queue) {
   while(TRUE) {
      con=Dequeue(queue);
      if (con==null)
         sleepOn(queue);
      else
         ServiceWebPage(con);
   }
}
```

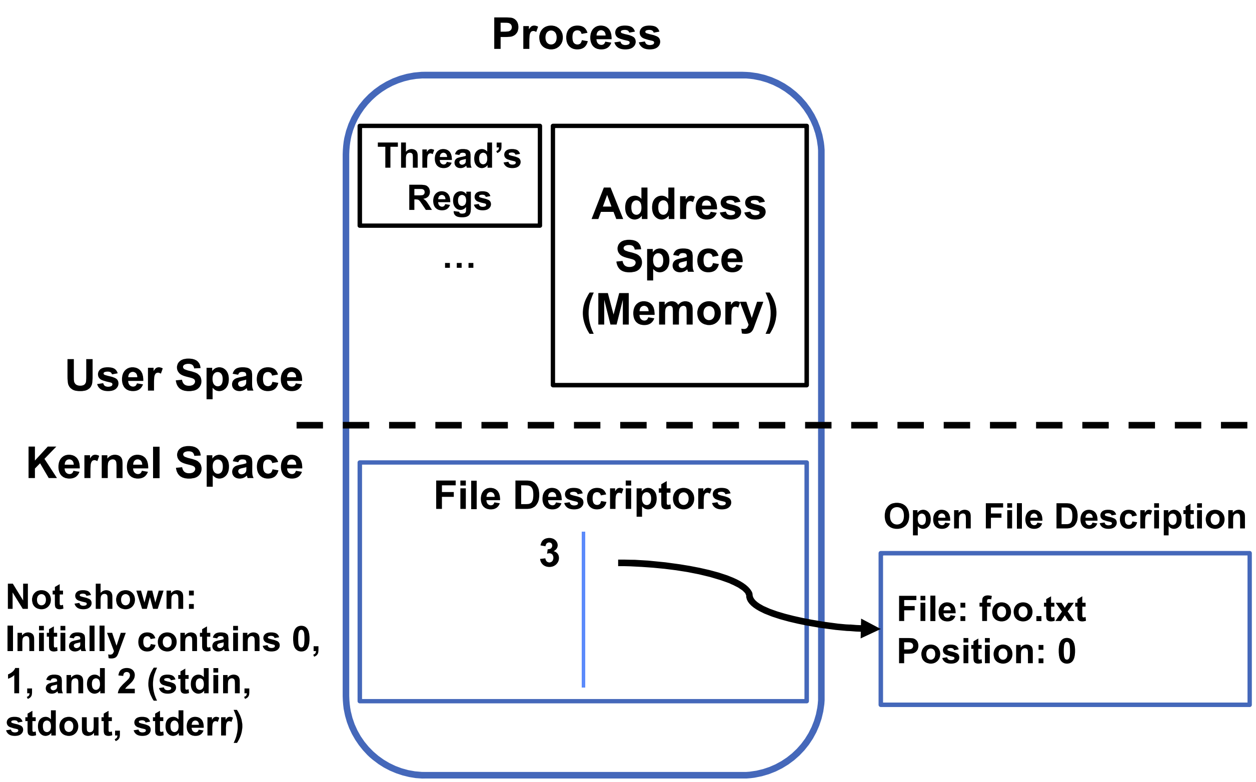
Recall: The Process Control Block

- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, …)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation, …
- Kernel *Scheduler* maintains a data structure containing the PCBs
  - Give out CPU to different processes
  - This is a Policy Decision
- Give out non-CPU resources
  - Memory/IO
  - Another policy decision



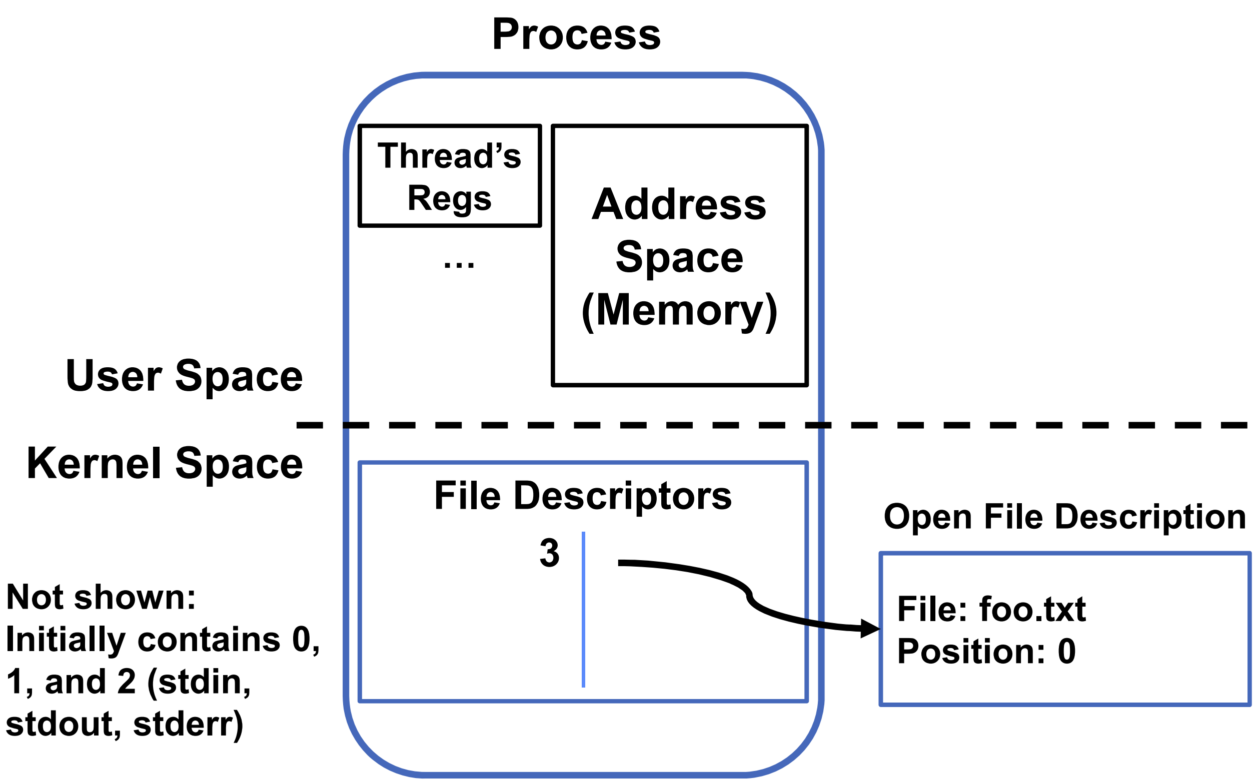
Process Control Block

Process-Specific File Descriptor Table inside Kernel



Suppose that we execute
open("foo.txt")
and that the result is 3

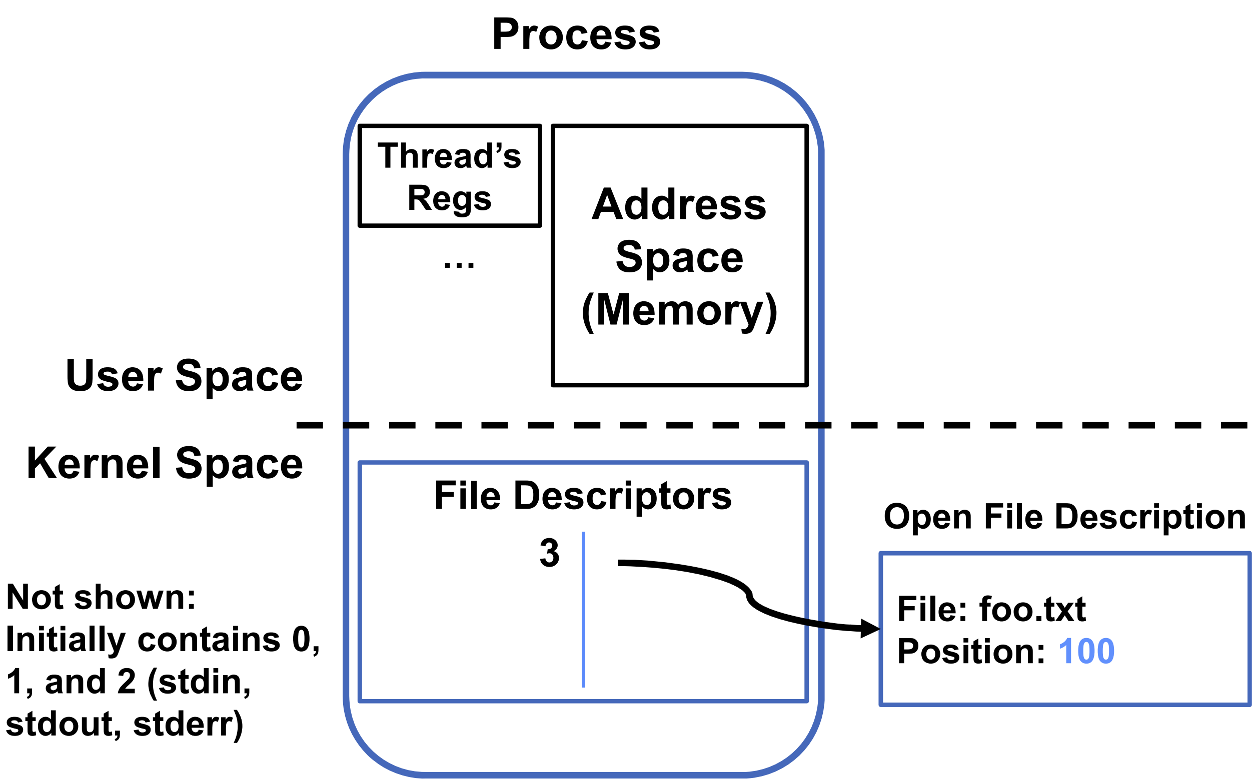
Process-Specific File Descriptor Table inside Kernel



Suppose that we execute
open("foo.txt")
and that the result is 3

Next, suppose that we execute
read(3, buf, 100)
and that the result is 100

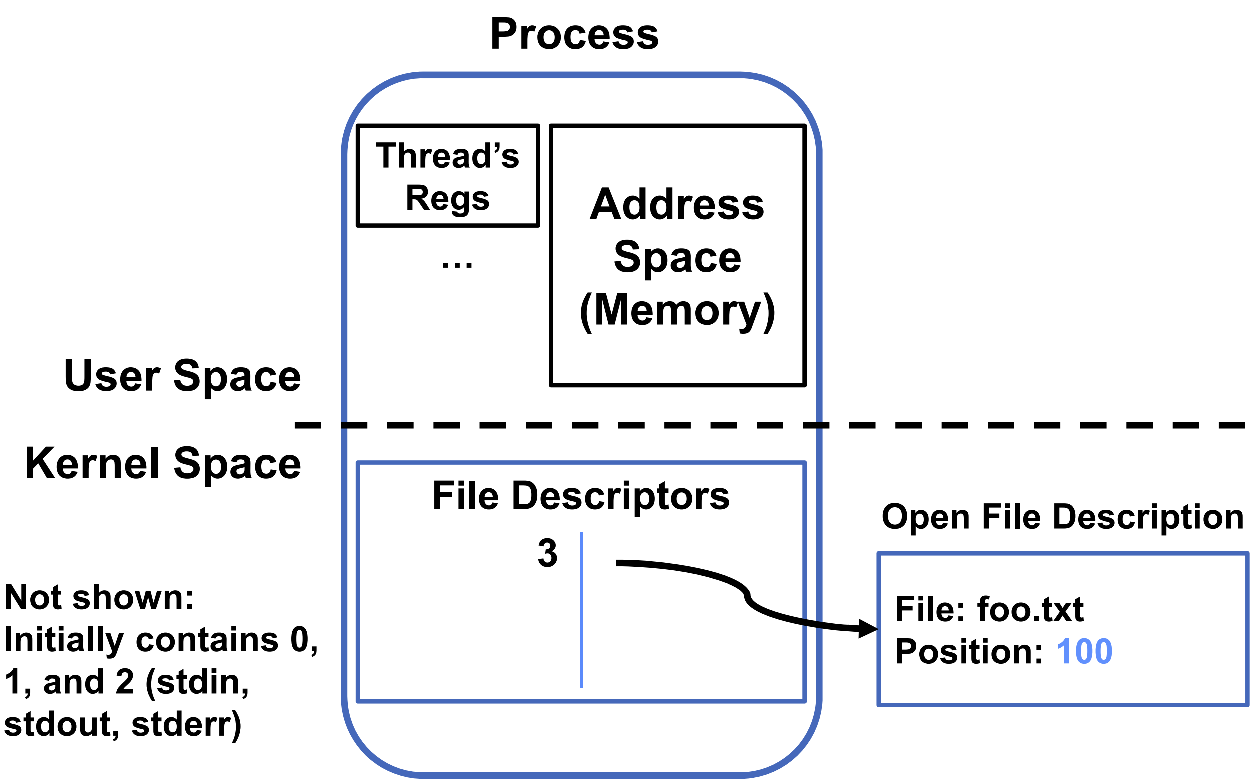
Process-Specific File Descriptor Table inside Kernel



Suppose that we execute
open("foo.txt")
and that the result is 3

Next, suppose that we execute
read(3, buf, 100)
and that the result is 100

Process-Specific File Descriptor Table inside Kernel



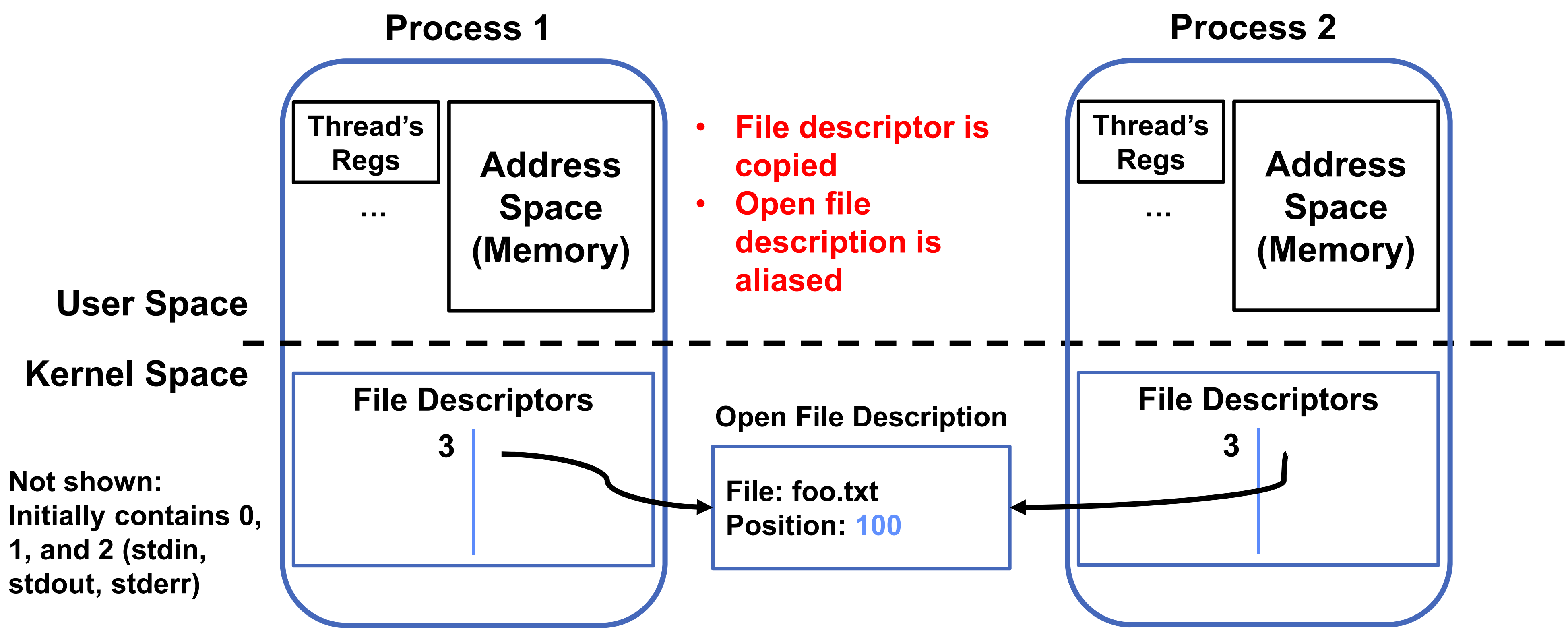
Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

Suppose that we execute
open("foo.txt")
and that the result is 3

Next, suppose that we execute
read(3, buf, 100)
and that the result is 100

Finally, suppose that we execute
close(3)

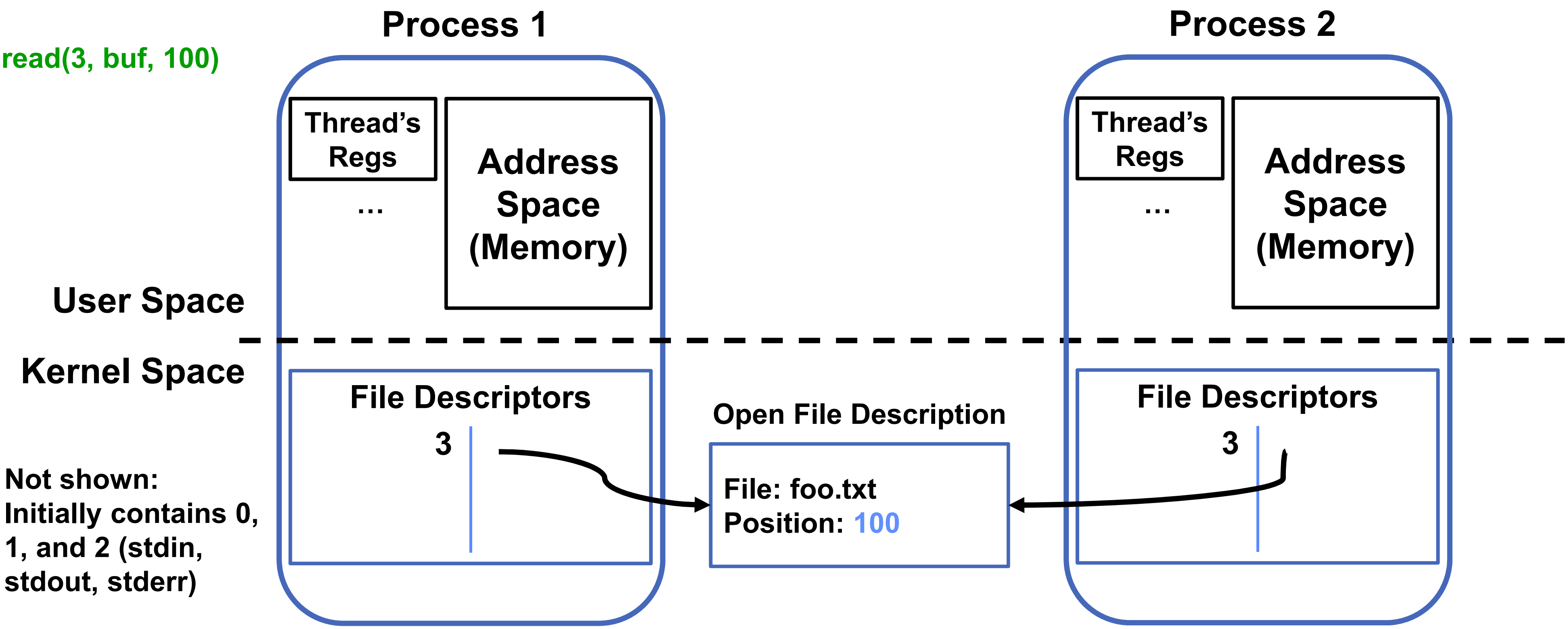
Instead of Closing, let's `fork()`!



- File descriptor is copied
- Open file description is aliased

Open File Description is *Aliased*

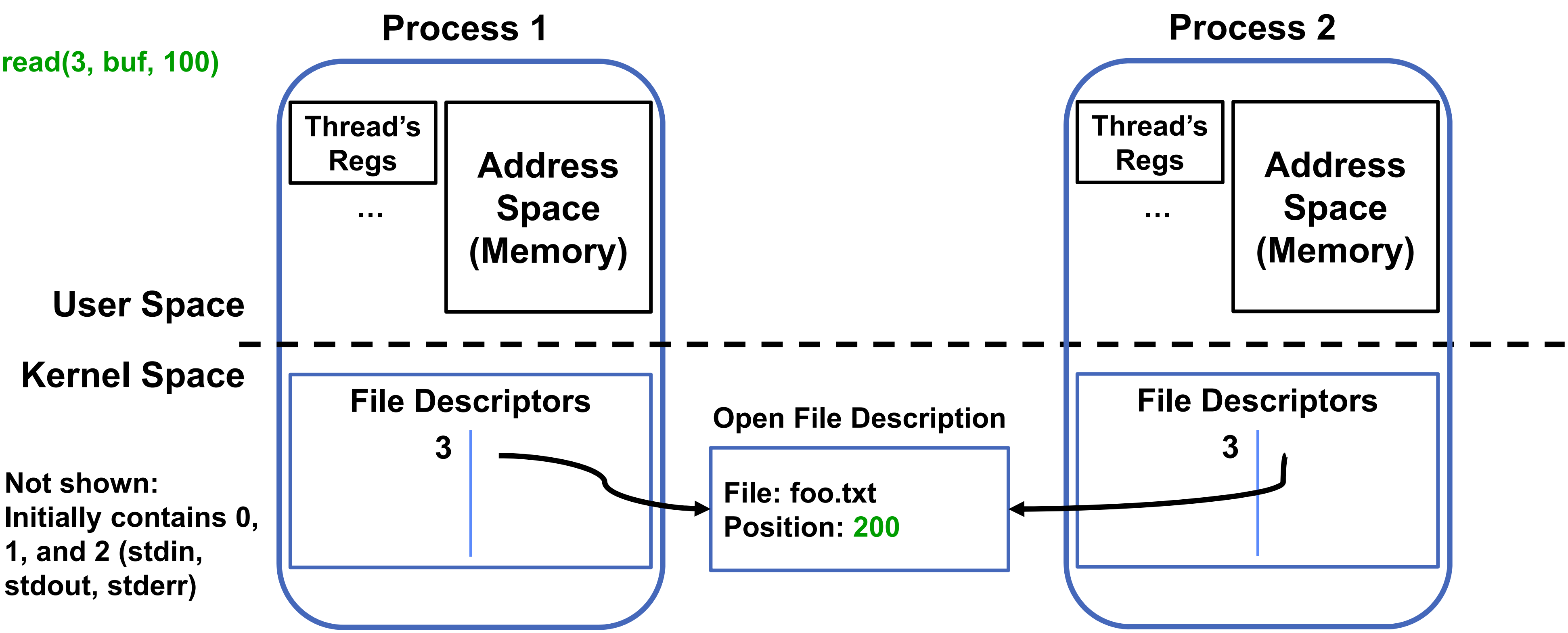
read(3, buf, 100)



Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

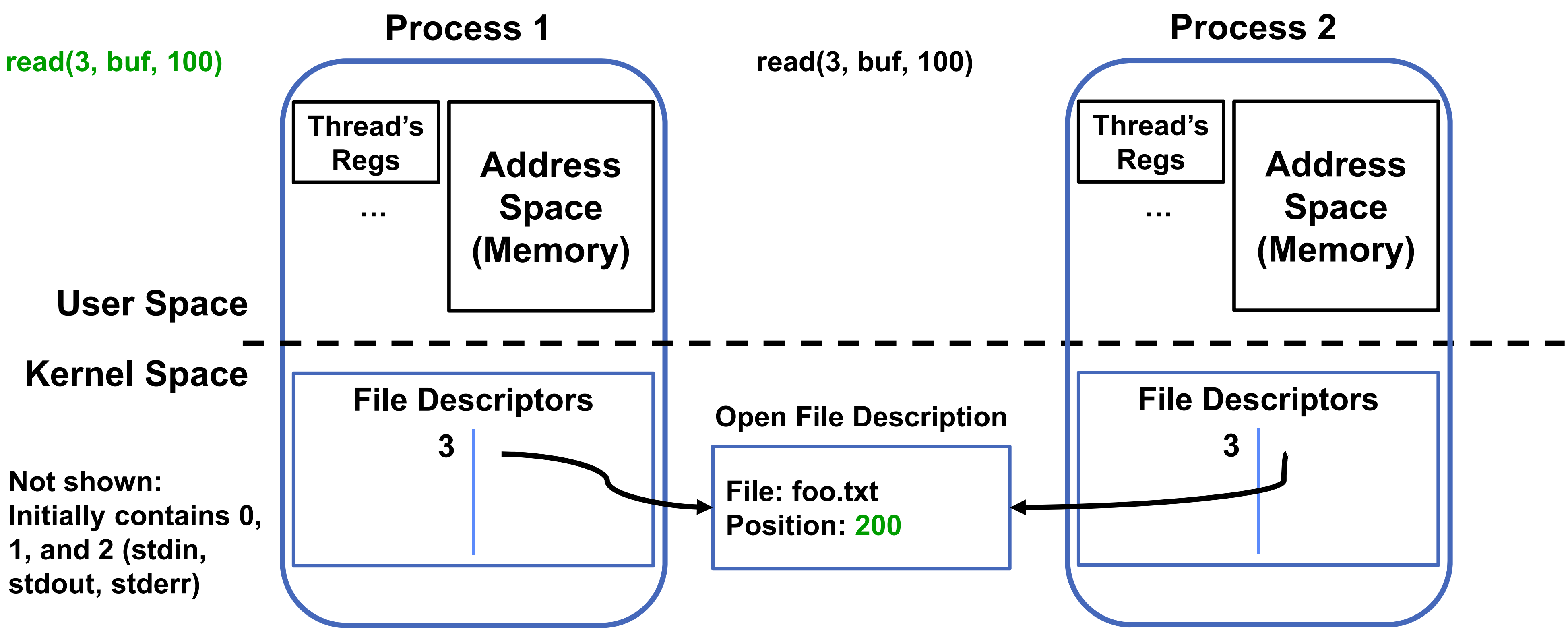
Open File Description is *Aliased*

read(3, buf, 100)



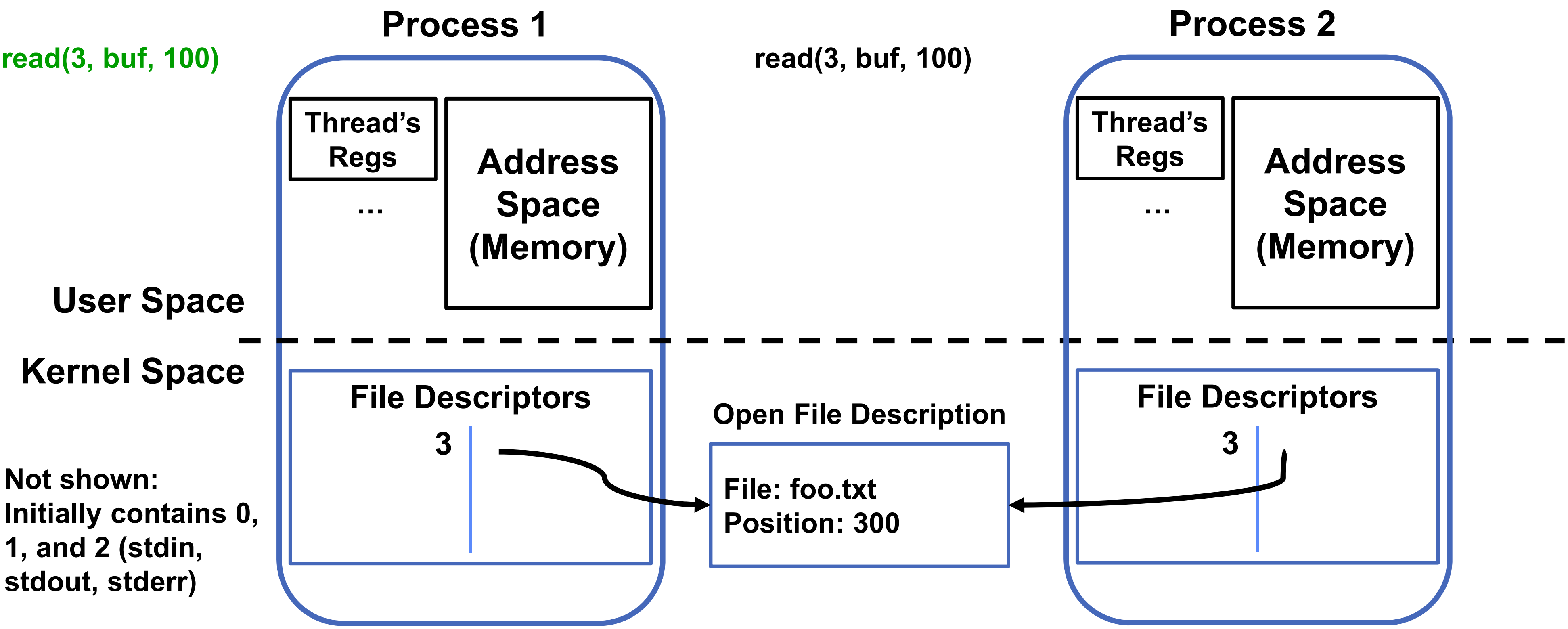
Not shown:
Initially contains 0, 1, and 2 (stdin, stdout, stderr)

Open File Description is *Aliased*



Open File Description is *Aliased*

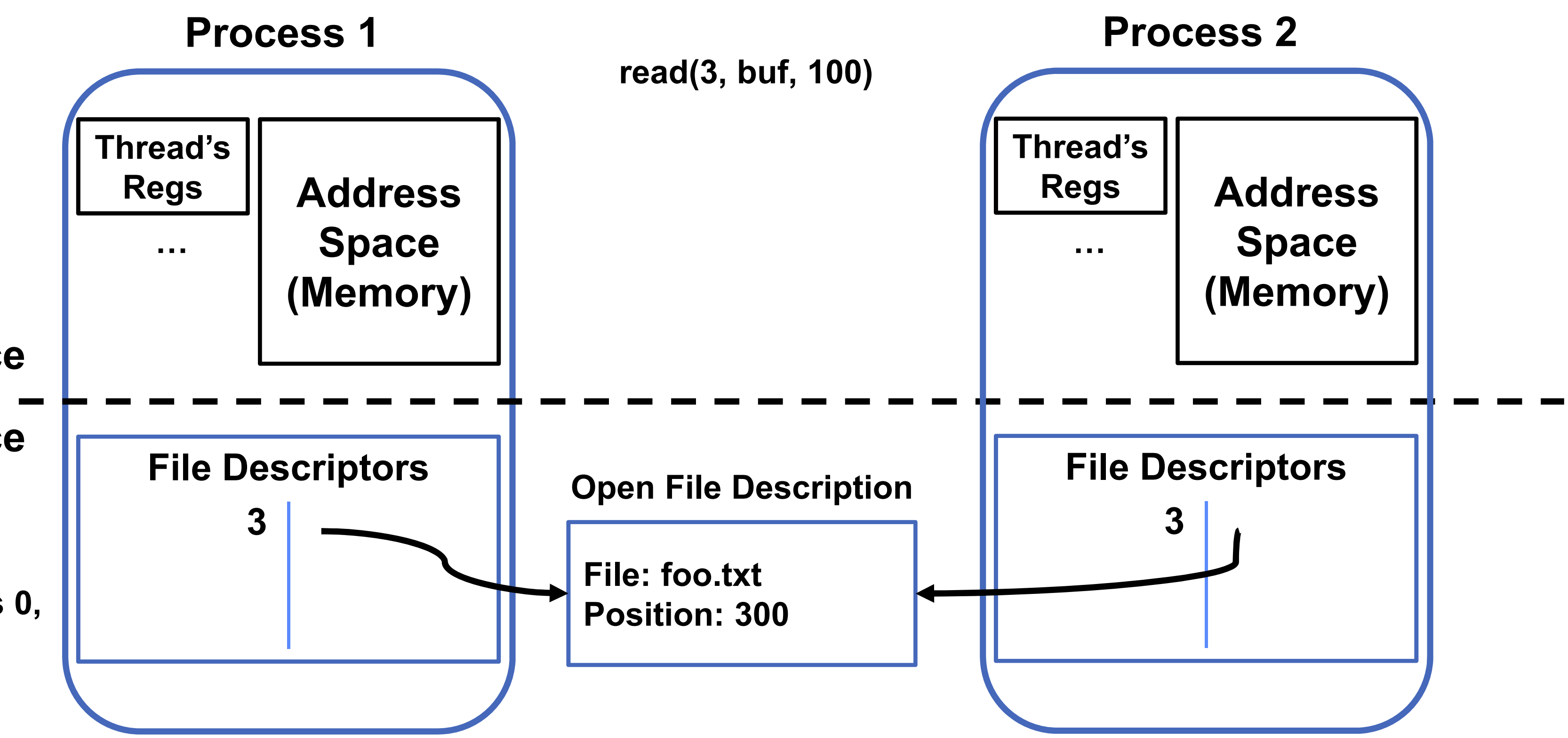
read(3, buf, 100)



Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

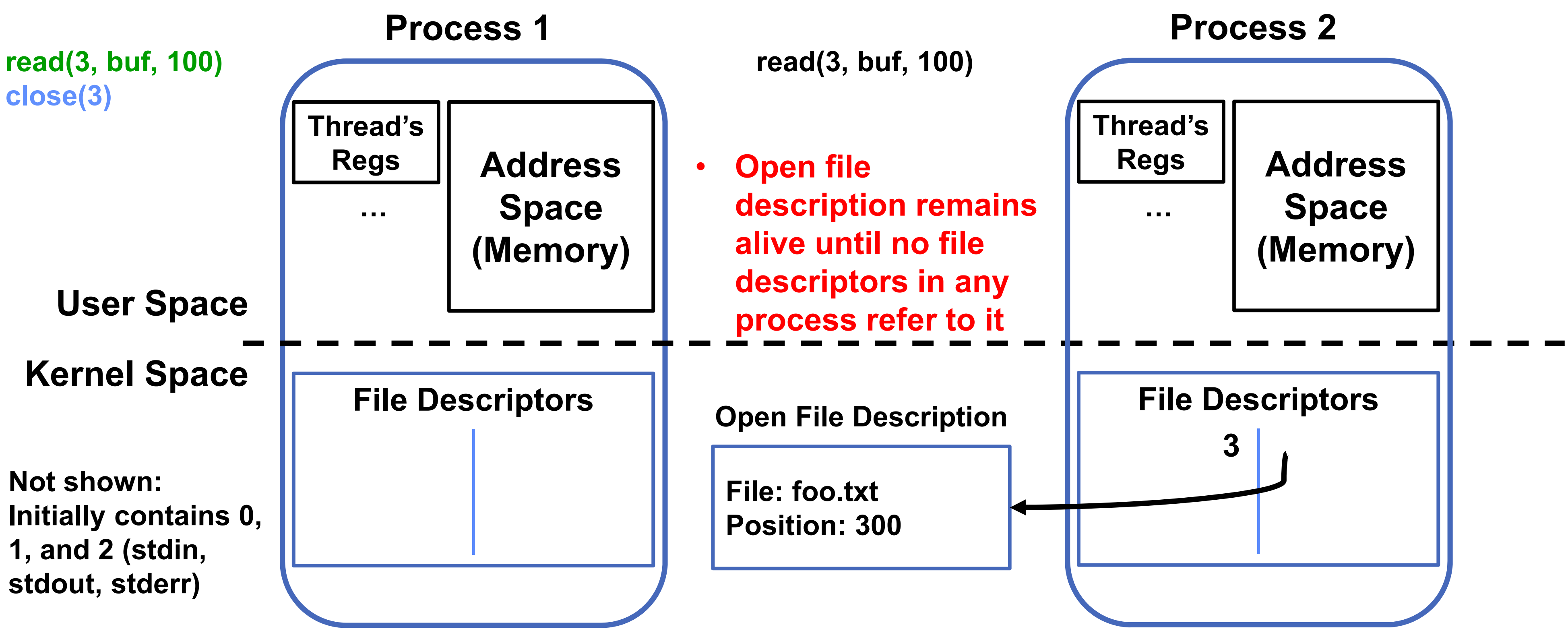
File Descriptor is *Copied*

read(3, buf, 100)
close(3)



Not shown:
Initially contains 0, 1, and 2 (stdin, stdout, stderr)

File Descriptor is *Copied*



- Open file description remains alive until no file descriptors in any process refer to it

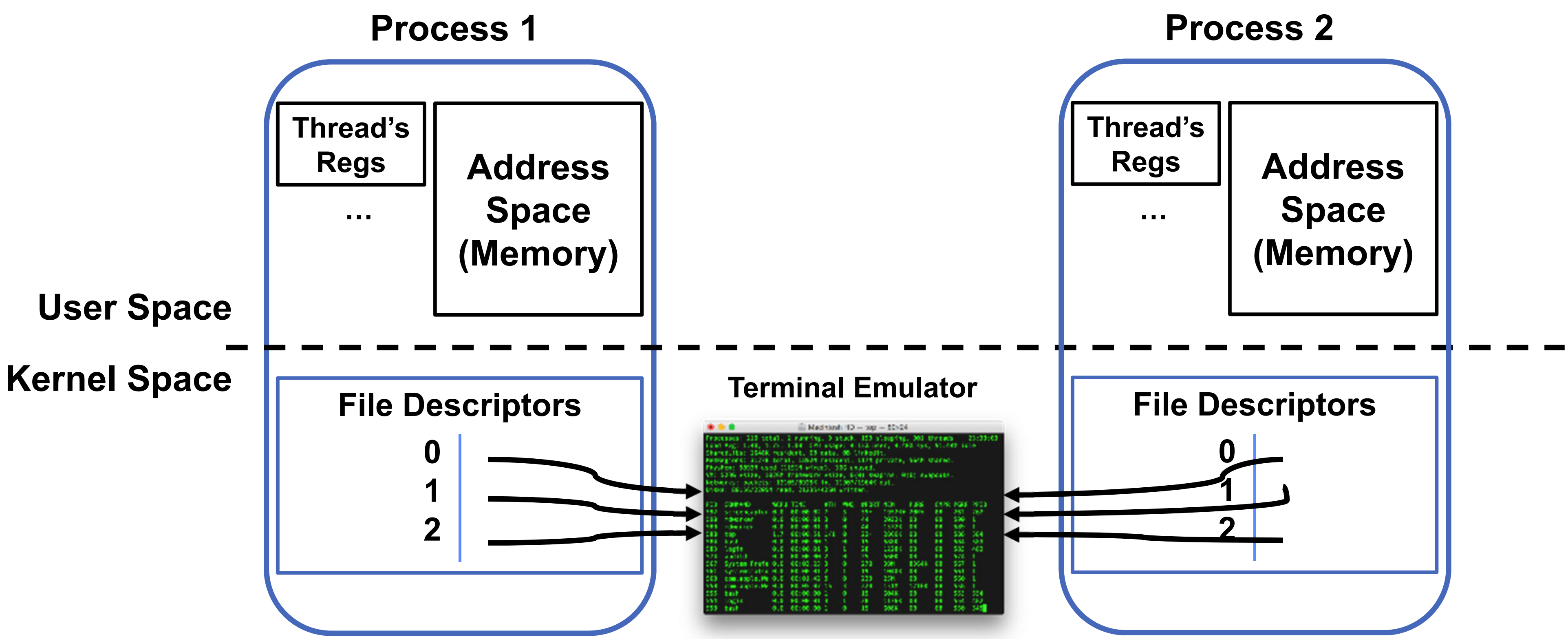
Why is Aliasing the Open File Description a Good Idea?

- It allows for *shared resources* between processes

Example: Shared Terminal Emulator

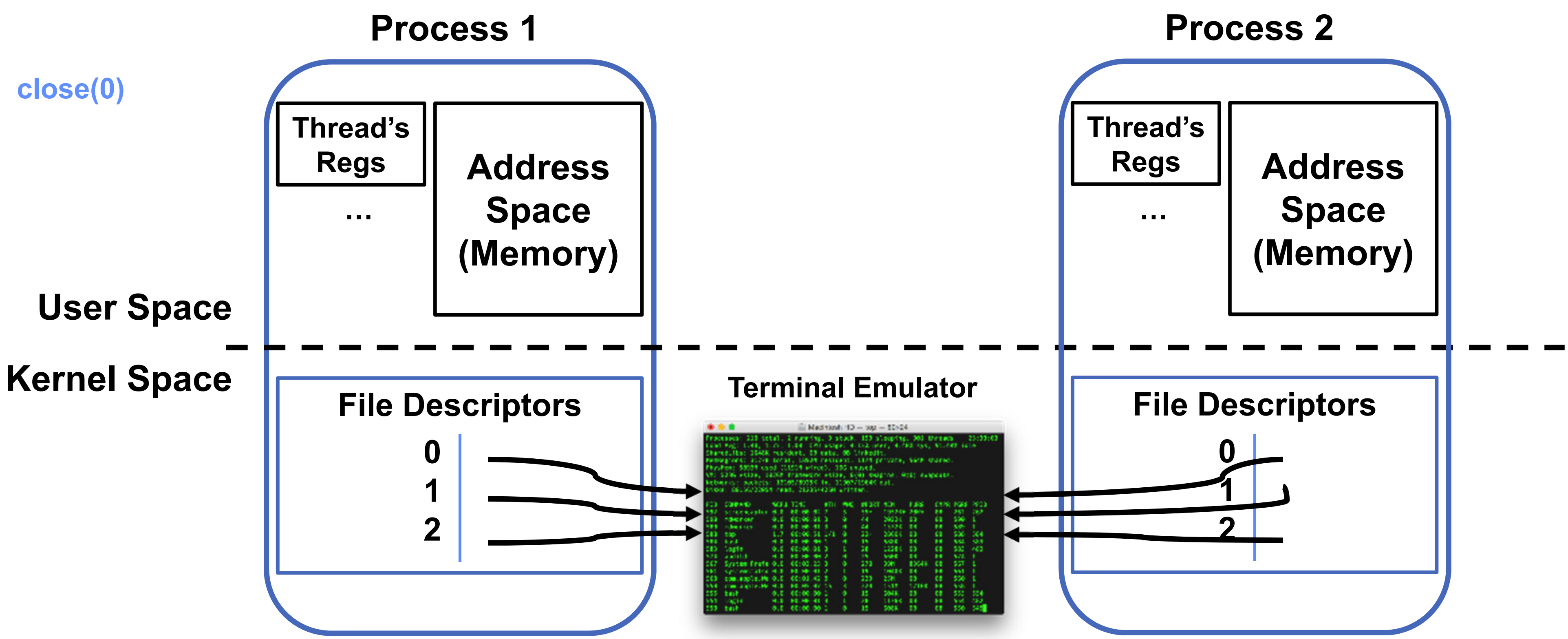
- When you `fork()` a process, the parent's and child's `printf` outputs go to the same terminal

Example: Shared Terminal Emulator

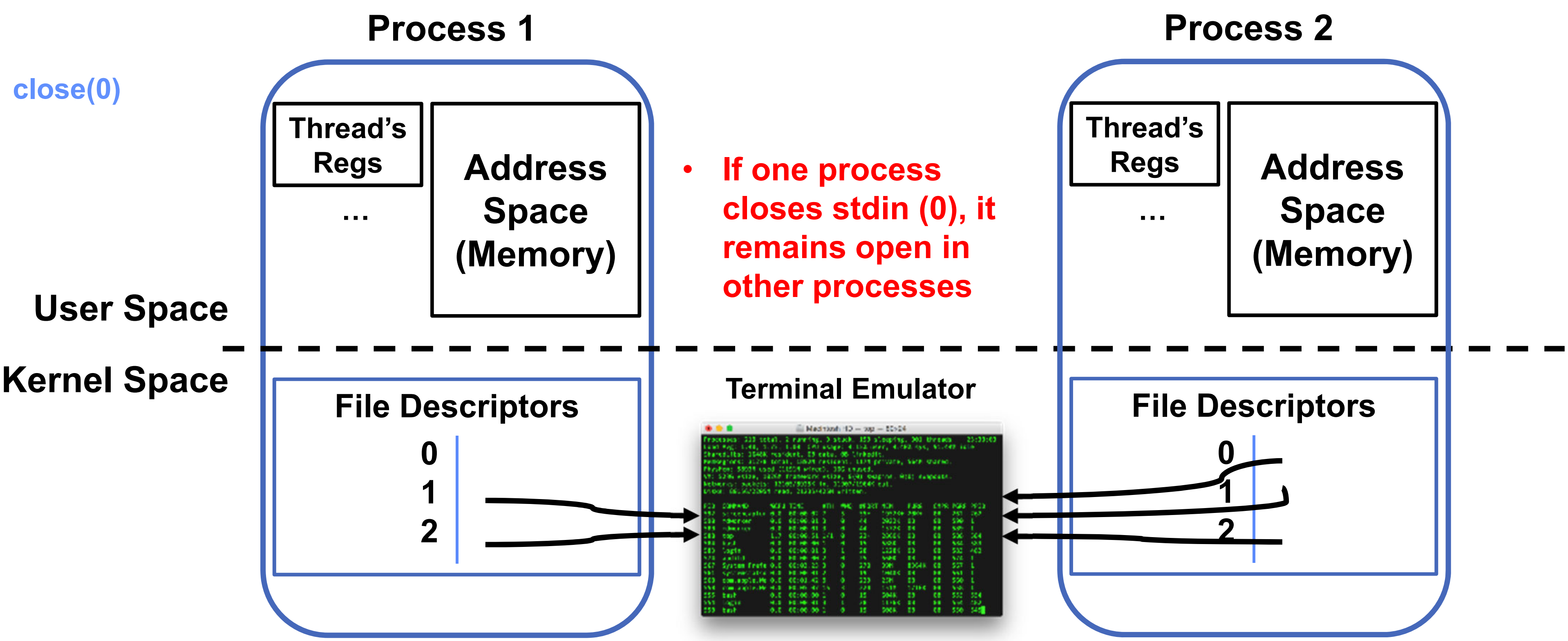


Example: Shared Terminal Emulator

close(0)



Example: Shared Terminal Emulator



- If one process closes stdin (0), it remains open in other processes

Other Examples

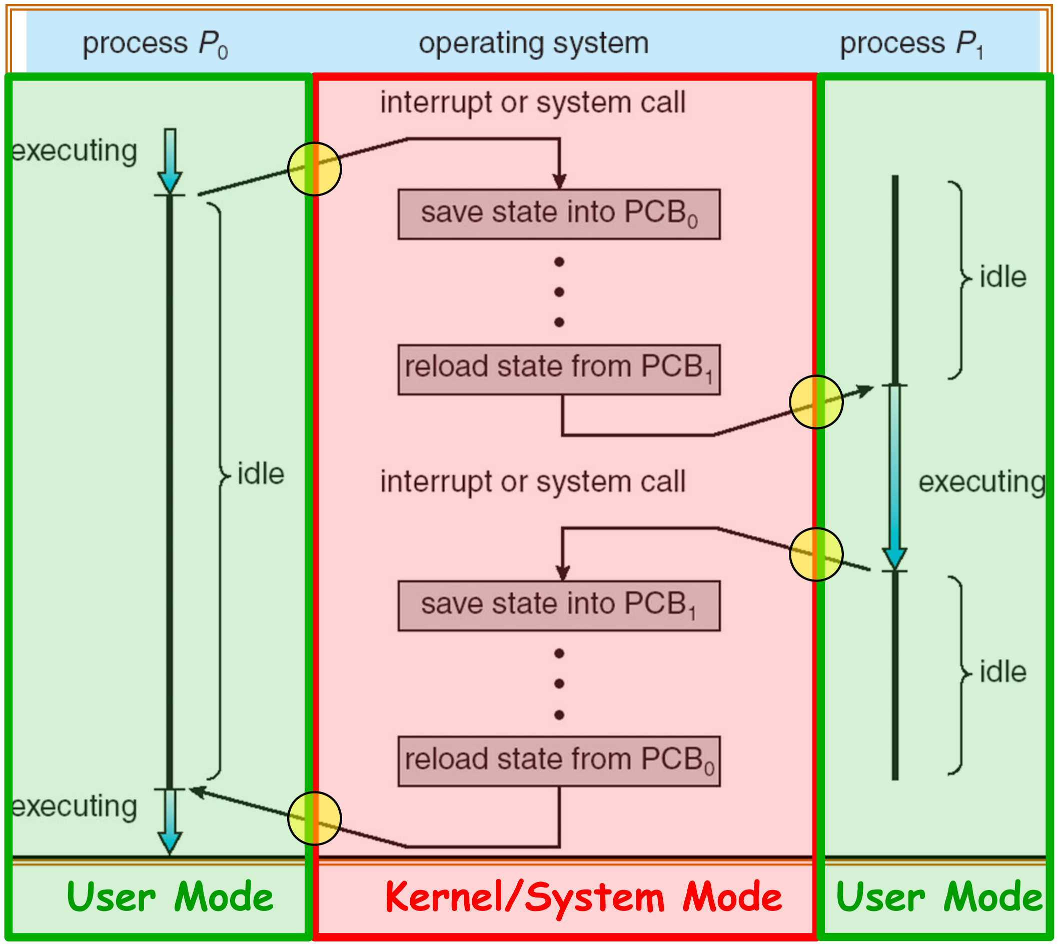
- Shared network connections after `fork()`
  - Allows handling each connection in a separate process
  - We saw this in our Webserver examples

- Pipes – channel for communication
  - `int pipe(int pipefd[2]);  /* Create array of two file descriptors */`
  - Writes to `pipefd[1]` can be read from `pipefd[0]`
  - Useful for interprocess communication:
    - » after `fork()`, both parent and child can communicate (one can read what other one writes)
  - And in writing a shell (Homework 2)

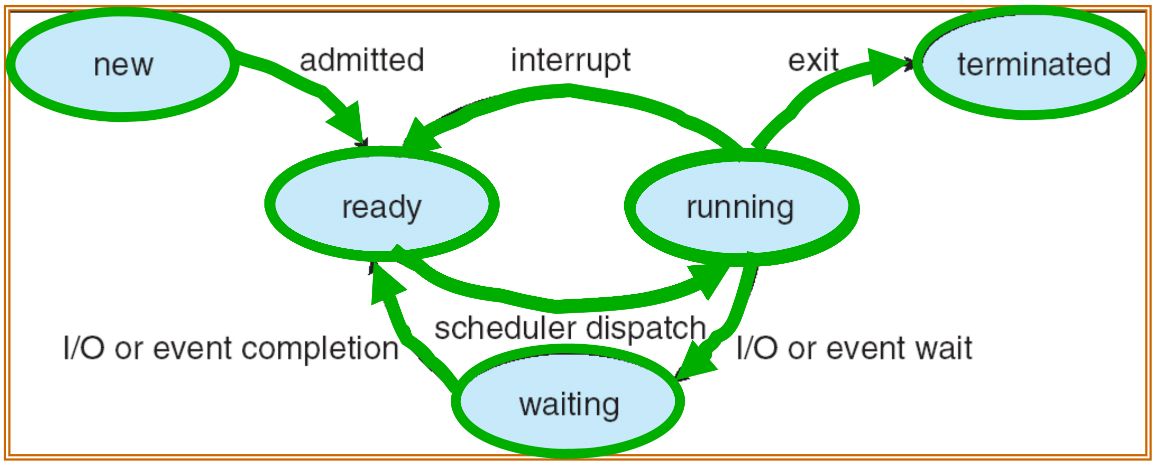
Administrivia

- Project 1 in full swing! Released Yesterday!
  - We expect that your design document will give intuitions behind your designs, not just a dump of pseudo-code
  - Think of this you are in a company and your TA is you manager
- Paradox: need code for design document?
  - Not full code, just enough prove you have thought through complexities of design
- Should be attending your permanent discussion section!
  - Discussion section attendance is mandatory, but don't come if sick!!
    - » We have given a mechanism to make up for missed sections—see EdStem
- Midterm 1: February 16th, 7-9PM (Two weeks from today!)
  - Fill out conflict request by tomorrow!

Recall: CPU Switch From Process A to Process B

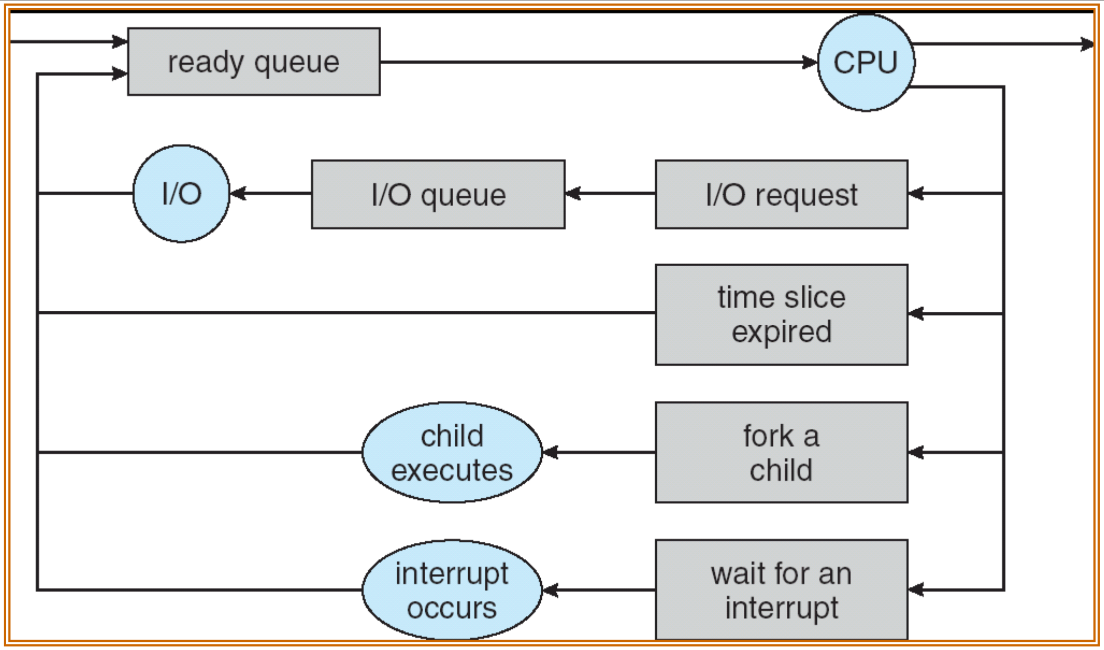


Lifecycle of a Process



- 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

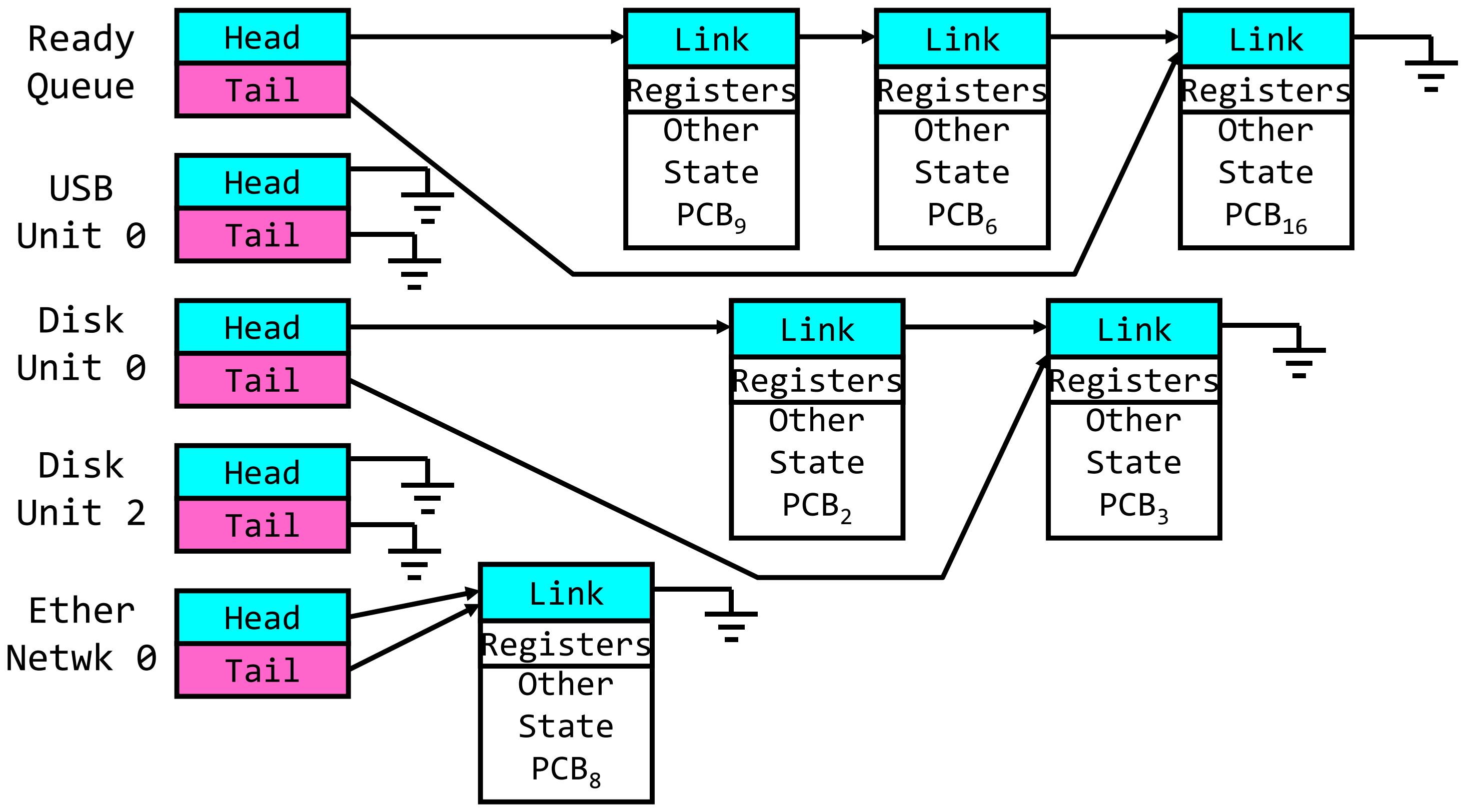
Process Scheduling



- 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

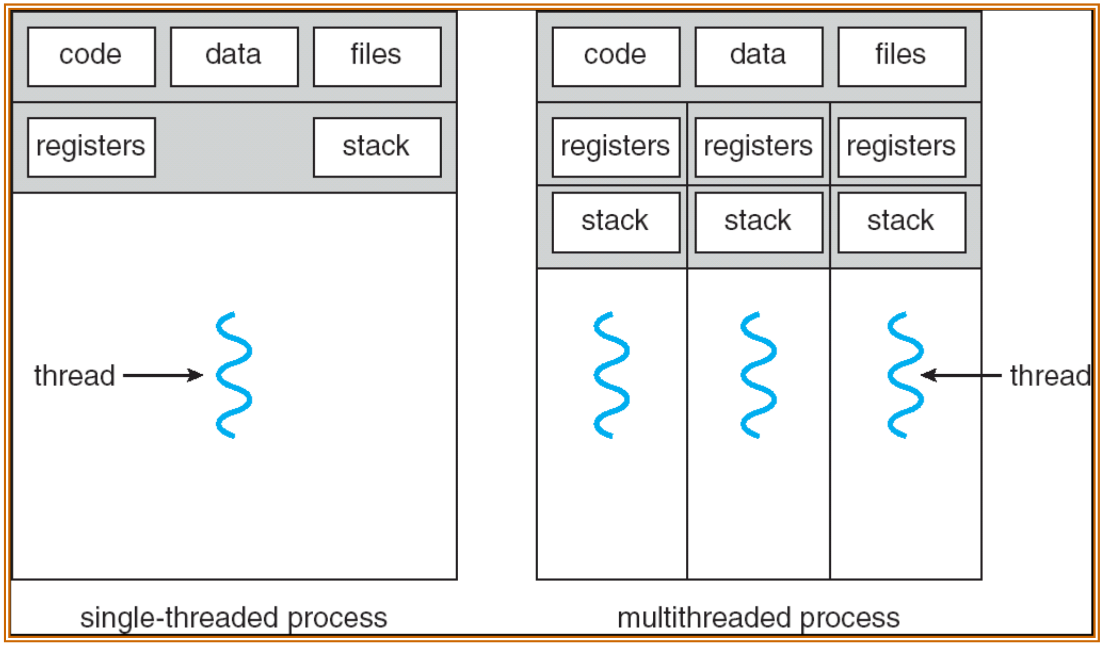
- Process not running ⇒ PCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy



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

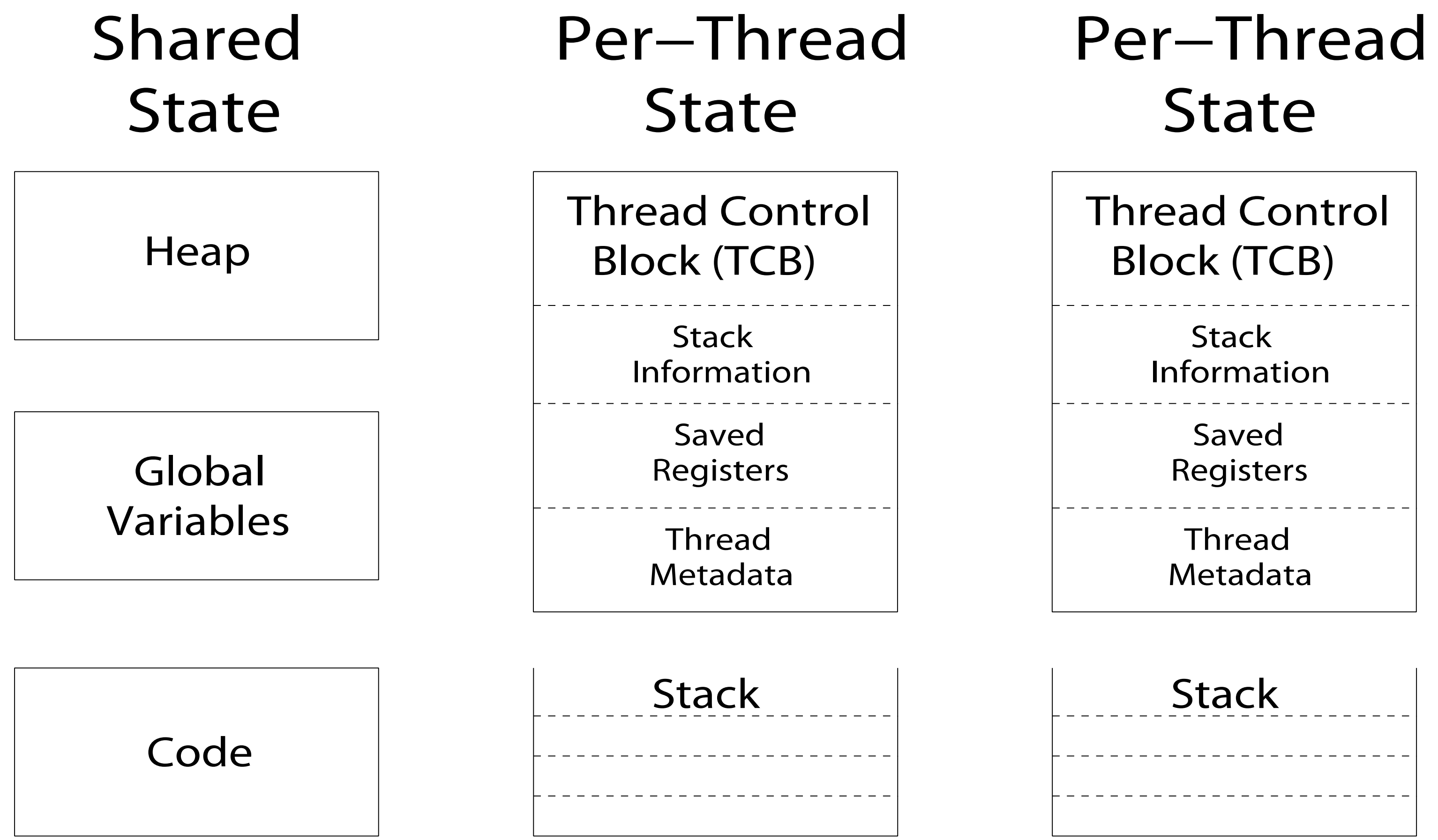


- Threads encapsulate concurrency: “Active” component
- Address spaces encapsulate protection: “Passive” part
  - – Keeps buggy program from trashing the system
- Why have multiple threads per address space?

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



Example: 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 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 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:
```

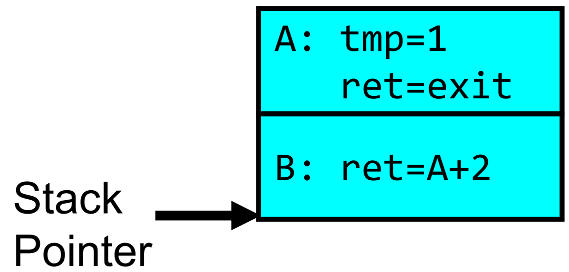
```
A: tmp=1
   ret=exit
B: ret=A+2
```

Stack Pointer

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

Execution Stack Example

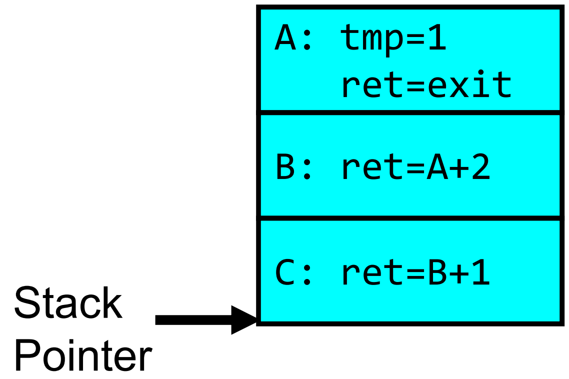
```
         A(int tmp) {
A:         if (tmp<2)
A+1:         B();
A+2:       printf(tmp);
         }
         B() {
B:         C();
B+1:     }
         C() {
C:         A(2);
C+1:     }
         A(1);
exit:
```



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

Execution Stack Example

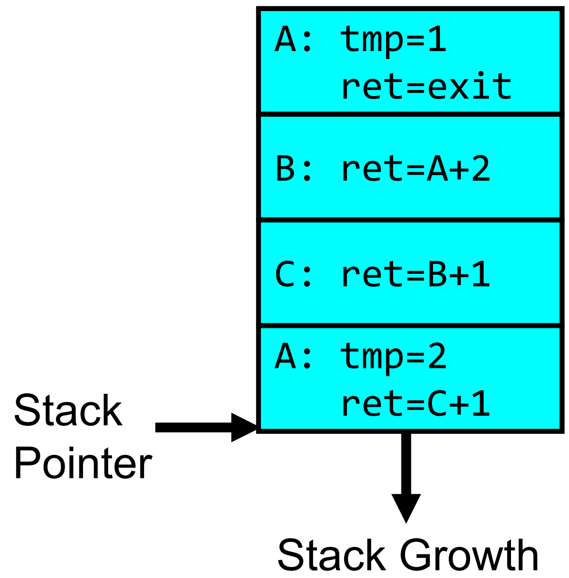
```
         A(int tmp) {
A:         if (tmp<2)
A+1:         B();
A+2:       printf(tmp);
         }
         B() {
B:         C();
B+1:     }
         C() {
C:         A(2);
C+1:     }
         A(1);
exit:
```



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

Execution Stack Example

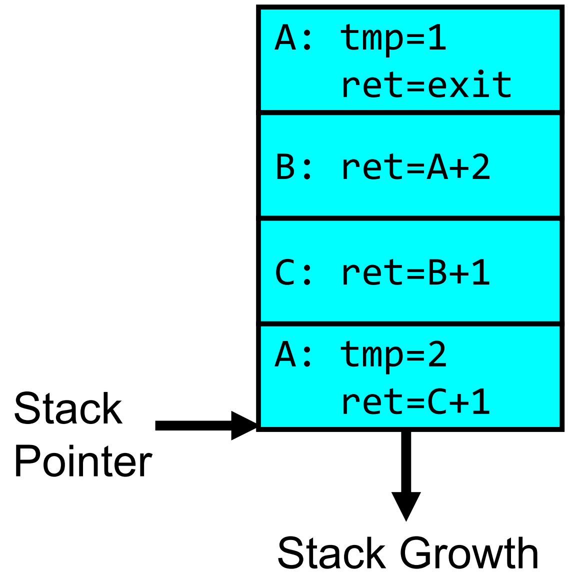
```
         A(int tmp) {
A:         if (tmp<2)
A+1:         B();
A+2:       printf(tmp);
         }
         B() {
B:         C();
B+1:     }
         C() {
C:         A(2);
C+1:     }
         A(1);
exit:
```



- 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:
```

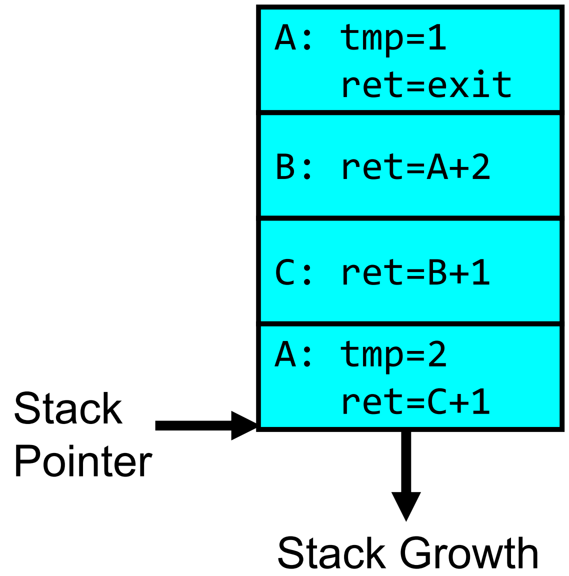


Output: >2

- 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:
```

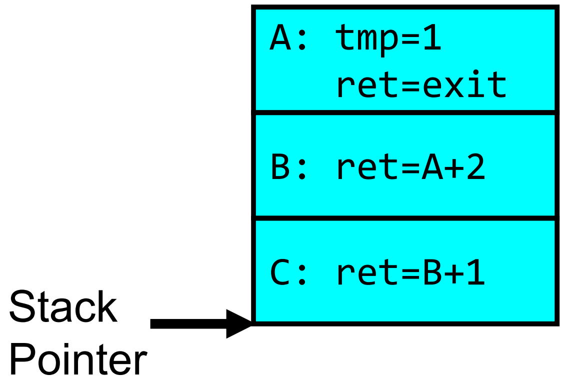


Output: >2

- 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:
```

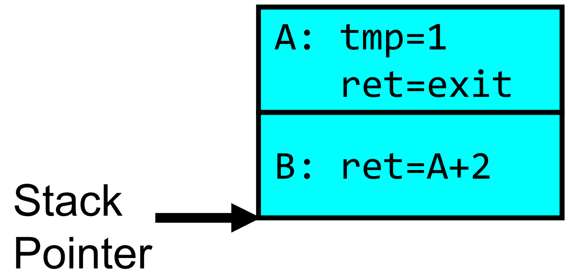


Output: >2

- 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:      A(int tmp) {
          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

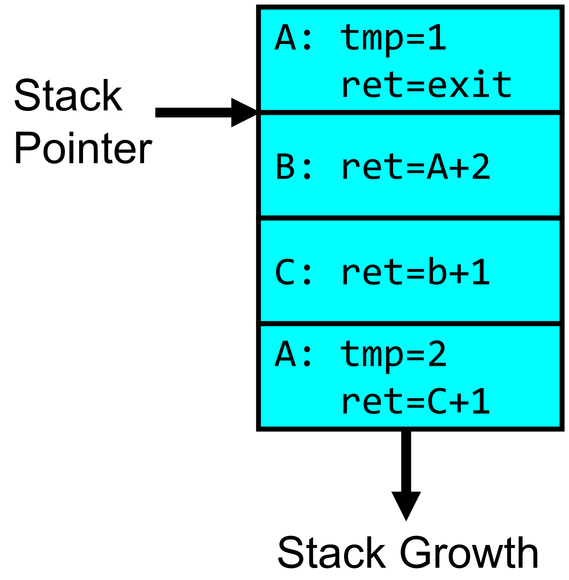
```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```

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");
}
```

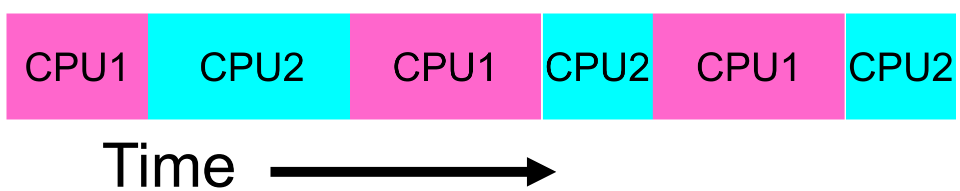
- What is the behavior here?
  – Program would never print out class list
  – Why? `ComputePI` would never finish

Use of Threads

- Version of program with Threads (loose syntax):

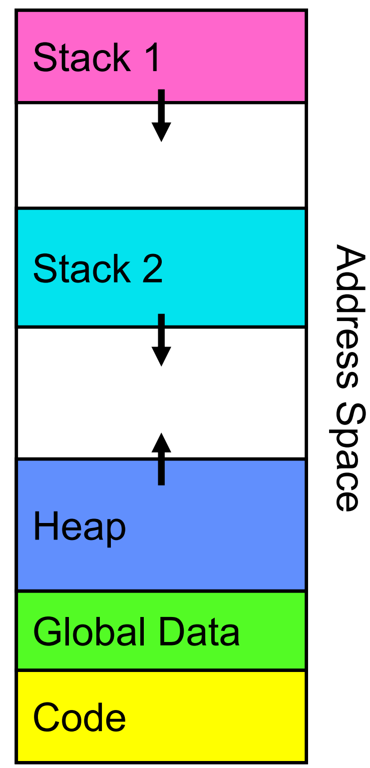
```
main() {
    ThreadFork(ComputePI, "pi.txt" ));
    ThreadFork(PrintClassList, "classlist.txt"));
}
```

- 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?



OS Library API for Threads: *pthreads*

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

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void*), void *arg);
```

– thread is created executing *start_routine* with *arg* as its sole argument.
– return is implicit call to pthread_exit

```
void pthread_exit(void *value_ptr);
```

– terminates the thread and makes *value_ptr* available to any successful join

```
int pthread_yield();
```

– causes the calling thread to yield the CPU to other threads

```
int pthread_join(pthread_t thread, void **value_ptr);
```

– suspends execution of the calling thread until the target *thread* terminates.
– On return with a non-NULL *value_ptr* the value passed to *pthread_exit()* by the terminating thread is made available in the location referenced by *value_ptr*.

prompt% man pthread
https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

The Core of Concurrency: the Dispatch Loop

- Conceptually, the scheduling 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()`

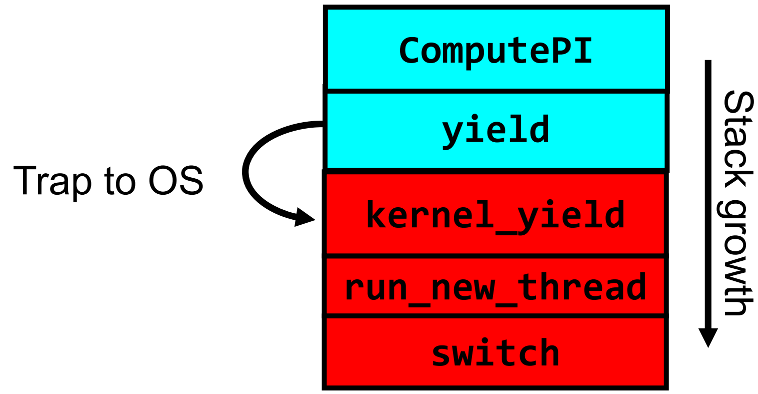
- How do I run a thread?
  - Load its state (registers, PC, stack pointer) into CPU
  - Load environment (virtual memory space, etc)
  - Jump to the PC
- How does the dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets *preempted*

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(); /* Do any cleanup */
}
```

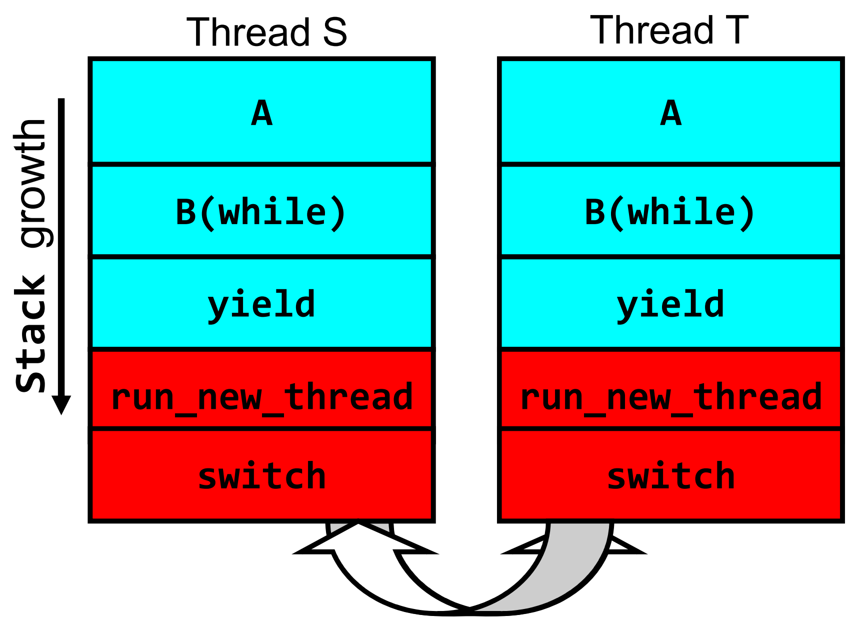
- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack pointer
  - Maintain isolation for each thread

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



Thread S's switch returns to Thread T's (and vice versa)

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 */
}
```

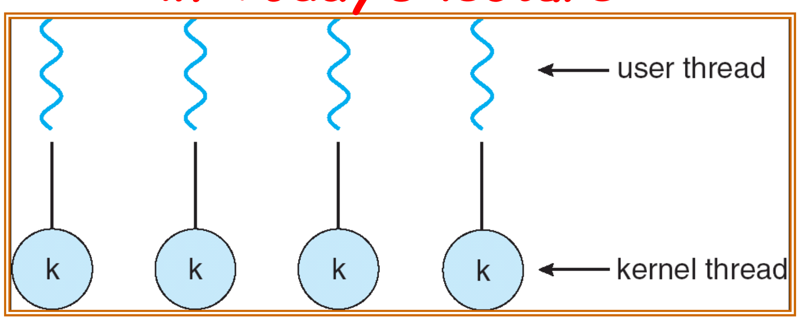
Switch Details (continued)

- 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

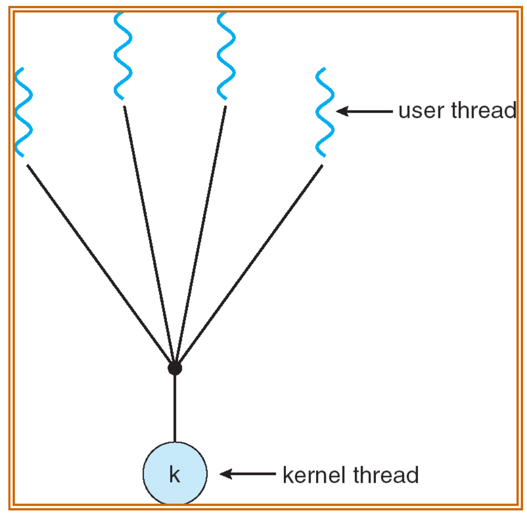
Are we still switching contexts with previous examples?

- Yes, but much cheaper than switching processes
  – No need to change address space
- Some numbers from Linux:
  – Frequency of context switch: 10-100ms
  – Switching between processes: 3-4 μsec.
  – Switching between threads: 100 ns
- Even cheaper: switch threads (using “yield”) in user-space!

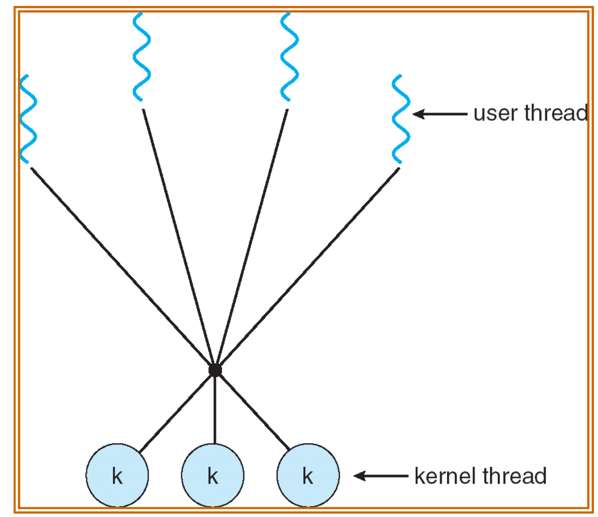
What we are talking about in Today’s lecture



Simple One-to-One Threading Model

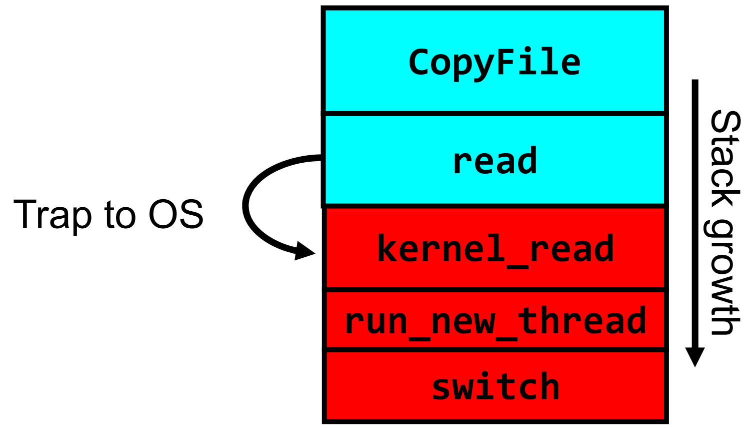


Many-to-One



Many-to-Many

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

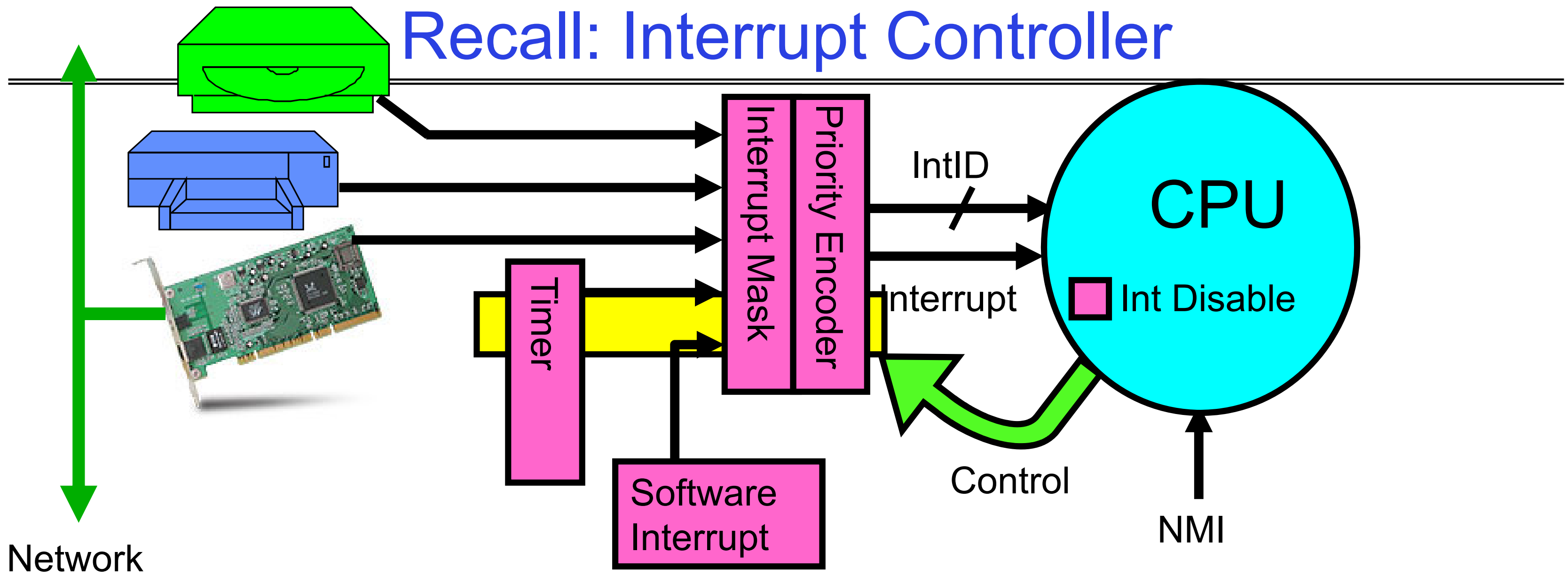
External Events

- 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

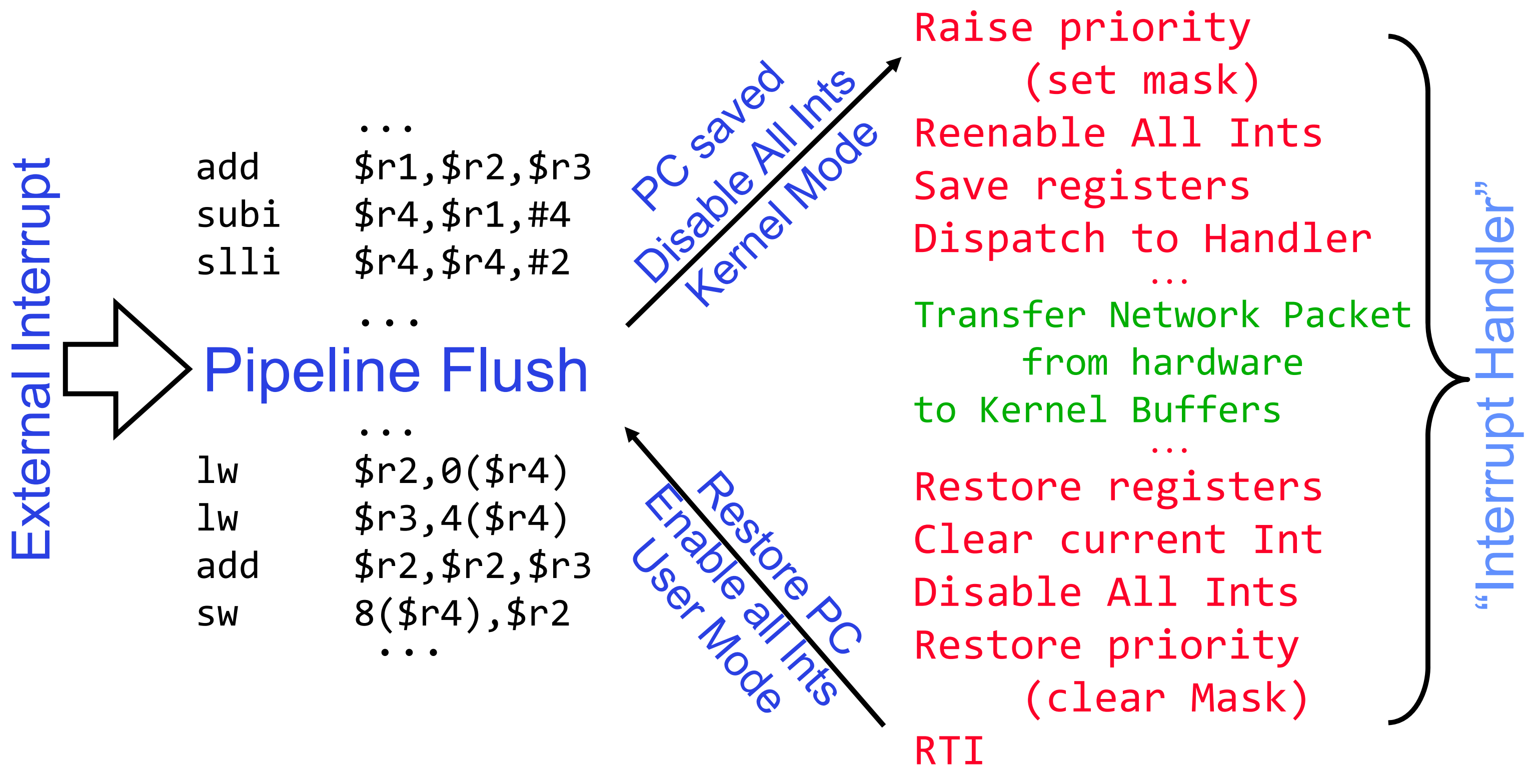
- If we make sure that external events occur frequently enough, can ensure dispatcher runs

Recall: Interrupt Controller



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

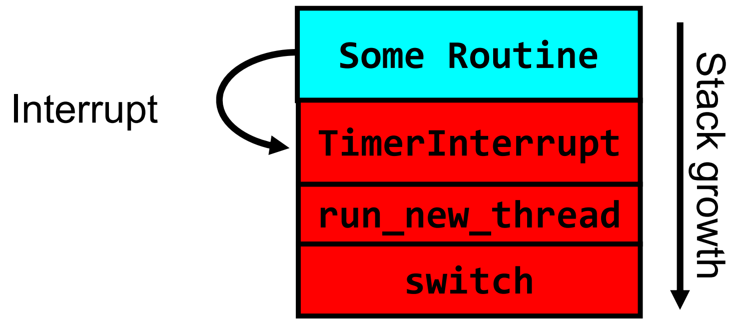
Example: Network Interrupt



- 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:

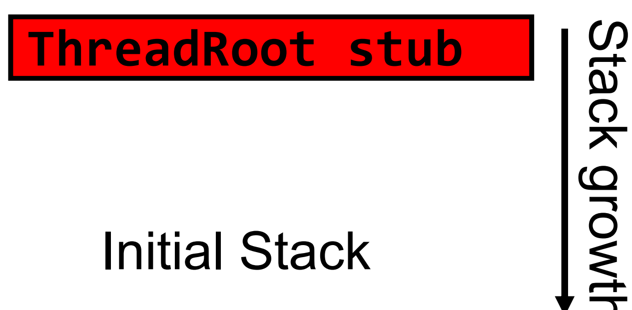
```
TimerInterrupt() {
   DoPeriodicHouseKeeping();
   run_new_thread();
}
```

`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)

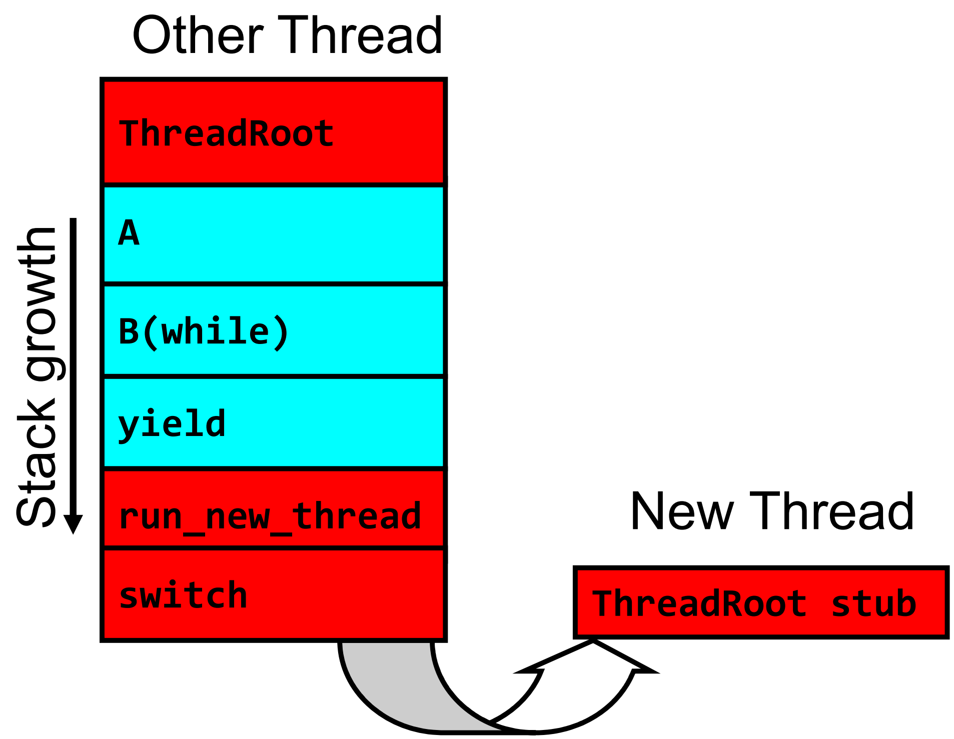
How do we initialize TCB and Stack?

- 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?
  - Minimal initialization ⇒setup return to go to beginning of ThreadRoot()
    - » Important part of stack frame is in registers for RISC-V (ra)
    - » X86: need to push a return address on stack
  - Think of stack frame as just before body of `ThreadRoot()` really gets started



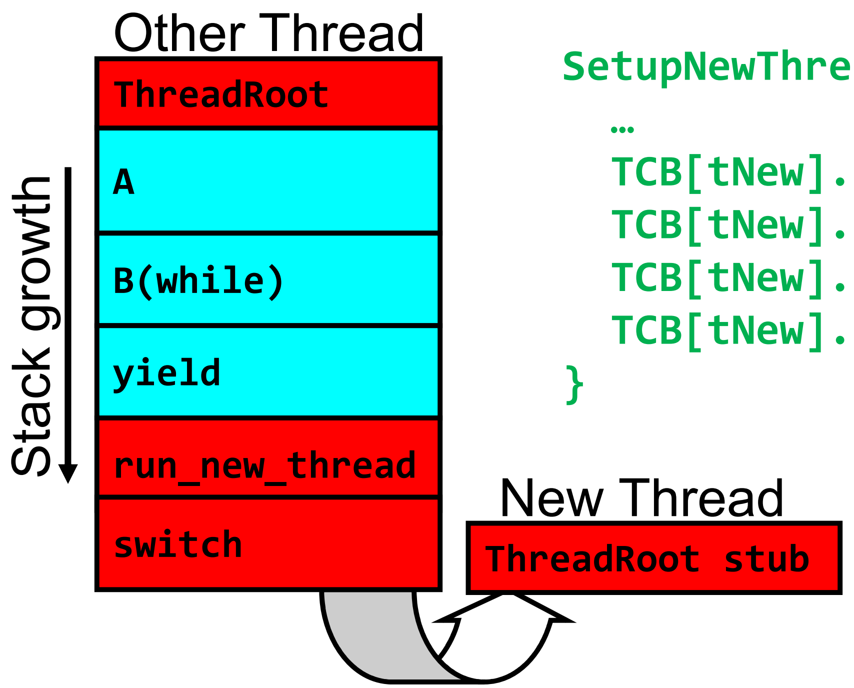
Initial Stack

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

How does a thread get started?



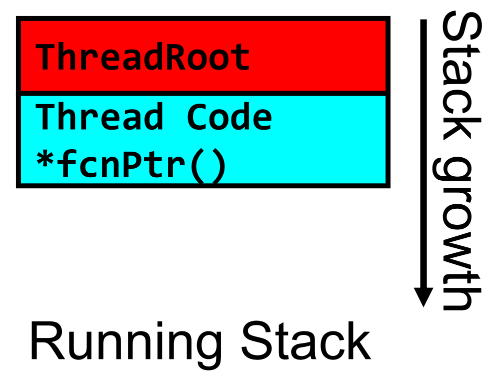

```
SetupNewThread(tNew) {
  …
  TCB[tNew].regs.sp = newStackPtr;
  TCB[tNew].regs.retpc = &ThreadRoot;
  TCB[tNew].regs.r0 = fcnPtr
  TCB[tNew].regs.r1 = fcnArgPtr
}
```

- How do we make a ***new*** thread?
  - Setup TCB/kernel thread to point at new user stack and ThreadRoot code
  - Put pointers to start function and args in registers or top of stack
    - This depends heavily on the calling convention (i.e. RISC-V vs x86)
- 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?

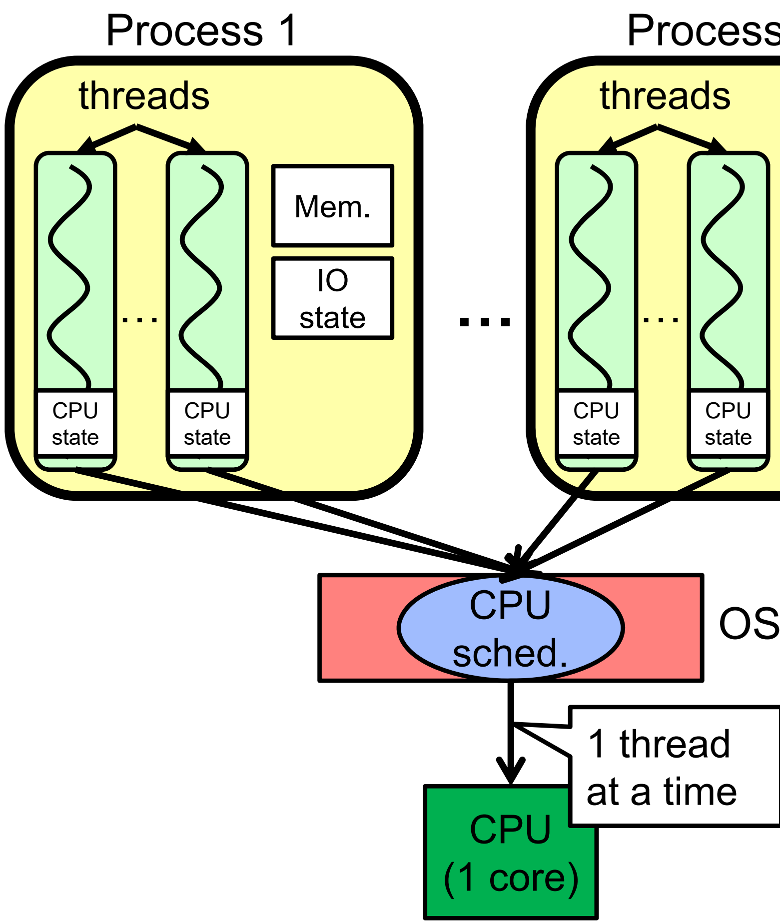
- `ThreadRoot()` is the root for the thread routine:

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



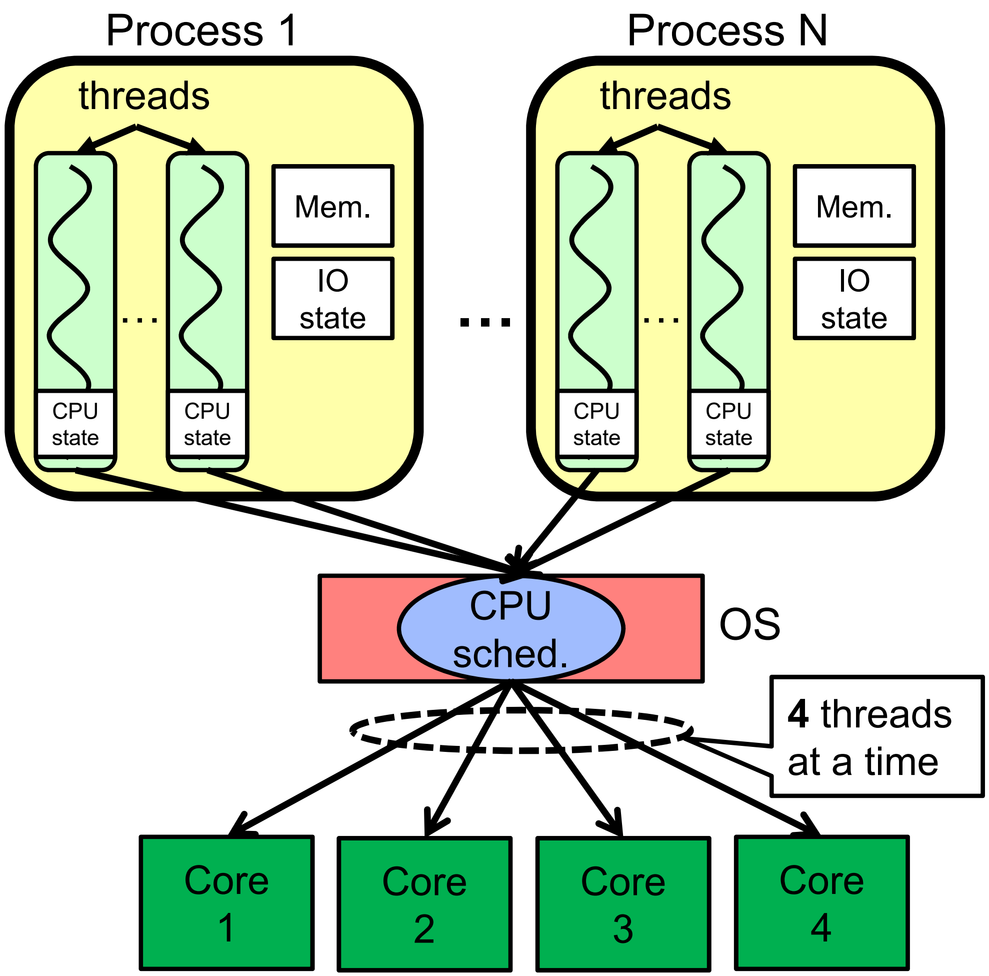
- Startup Housekeeping
  - Includes things like recording start time of thread
  - Other statistics
- Stack will grow and shrink with execution of thread
- Final return from thread returns into `ThreadRoot()` which calls `ThreadFinish()`
  - `ThreadFinish()` wake up sleeping threads

Processes vs. Threads: One Core



- Switch overhead:
  - Same process: **low**
  - Different proc.: **high**
- Protection
  - Same proc: **low**
  - Different proc: **high**
- Sharing overhead
  - Same proc: **low**
  - Different proc: **high**
- Parallelism: **no**

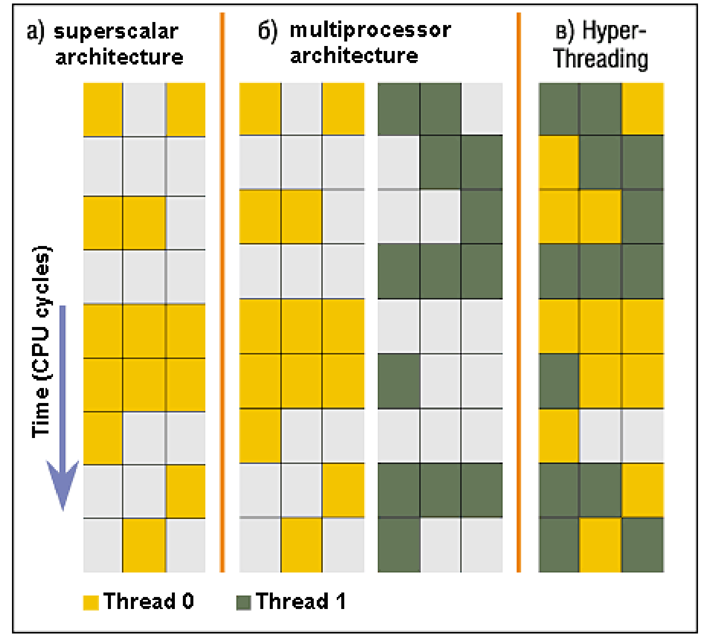
Processes vs. Threads: MultiCore



- Switch overhead:
  - Same process: **low**
  - Different proc.: **high**
- Protection
  - Same proc: **low**
  - Different proc: **high**
- Sharing overhead
  - Same proc: **low**
  - Different proc, simultaneous core: **medium**
  - Different proc, offloaded core: high
- Parallelism: **yes**

Recall: Simultaneous MultiThreading/Hyperthreading

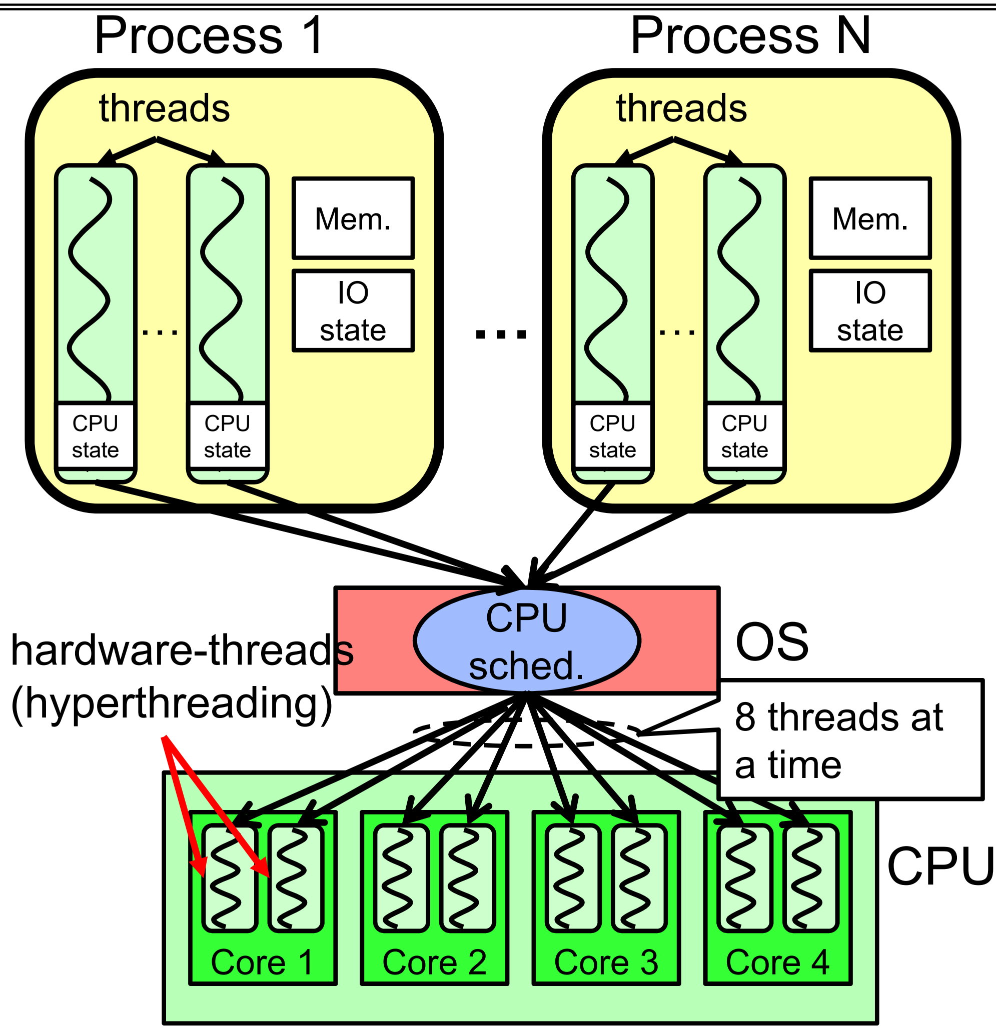
- Hardware scheduling technique
  - Superscalar processors can execute multiple instructions that are independent.
  - Hyperthreading duplicates register state to make a second "thread," allowing more instructions to run.
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!



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

Processes vs. Threads: Hyper-Threading



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

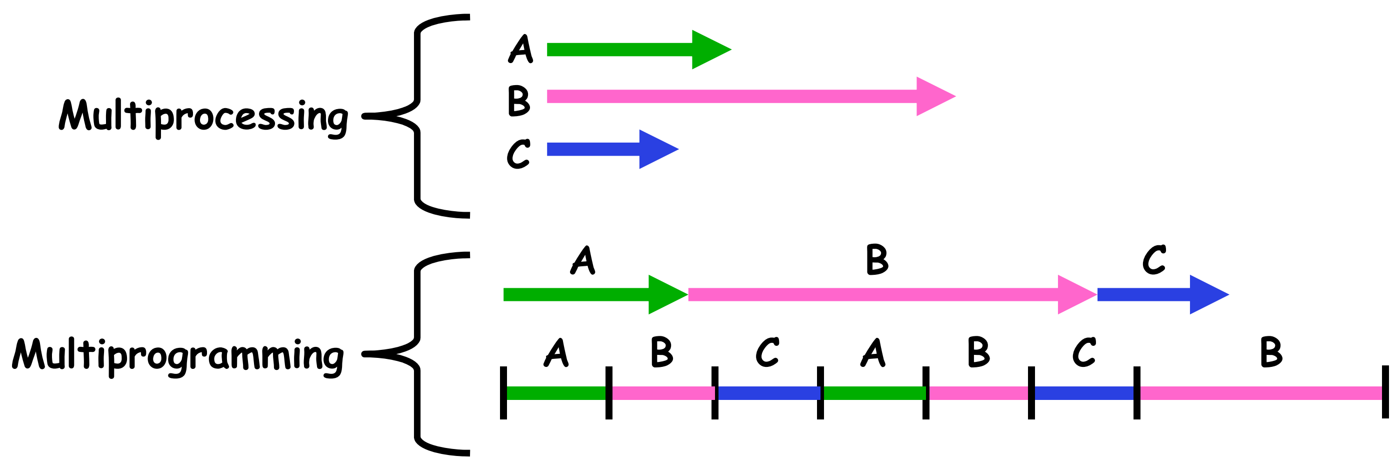
Threads vs Address Spaces: Options

# 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

Multiprocessing vs Multiprogramming

- Some Definitions:
  – Multiprocessing ≡ Multiple CPUs
  – Multiprogramming ≡ Multiple Jobs or Processes
  – Multithreading ≡ Multiple threads per Process
- What does it mean to run two threads "concurrently"?
  – Scheduler is free to run threads in any order and interleaving: FIFO, Random, …
  – Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks



Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
  - Can you test for this?
  - How can you know if your program works?
- Independent Threads:
  - No state shared with other threads
  - Deterministic $\Rightarrow$ Input state determines results
  - Reproducible $\Rightarrow$ Can recreate Starting Conditions, I/O
  - Scheduling order doesn't matter (if `switch()` works!!!)
- Cooperating Threads:
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible
- Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called "Heisenbugs"

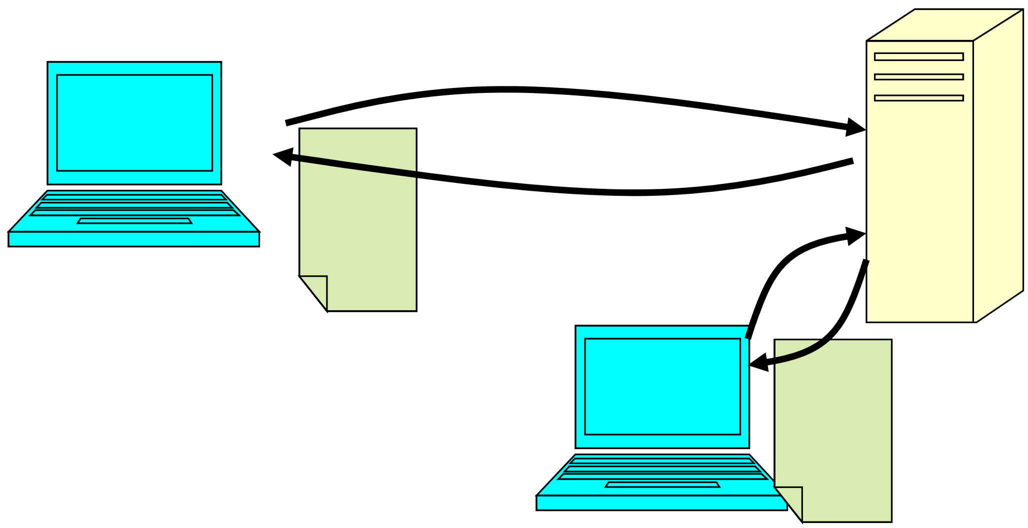
Interactions Complicate Debugging

- Is any program truly independent?
  - Every process shares the file system, OS resources, network, etc
  - Extreme example: buggy device driver causes thread A to crash "independent thread" B
- You probably don't realize how much you depend on reproducibility:
  - Example: Evil C compiler
    - Modifies files behind your back by inserting errors into C program unless you insert debugging code
  - Example: Debugging statements can overrun stack
- Non-deterministic errors are really difficult to find
  - Example: Memory layout of kernel+user programs
    - depends on scheduling, which depends on timer/other things
    - Original UNIX had a bunch of non-deterministic errors
  - Example: Something which does interesting I/O
    - User typing of letters used to help generate secure keys

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
    - » Many different file systems do read-ahead
  - Multiprocessors – chop up program into parallel pieces
- Advantage 3: Modularity
  - More important than you might think
  - Chop large problem up into simpler pieces
    - » To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
    - » Makes system easier to extend

Recall: High-level Example: Web Server



- Server must handle many requests
- Non-cooperating version:

```
serverLoop() {
    con = AcceptCon();
    ProcessFork(ServiceWebPage(),con);
}
```

- What are some disadvantages of this technique?

Recall: Threaded Web Server

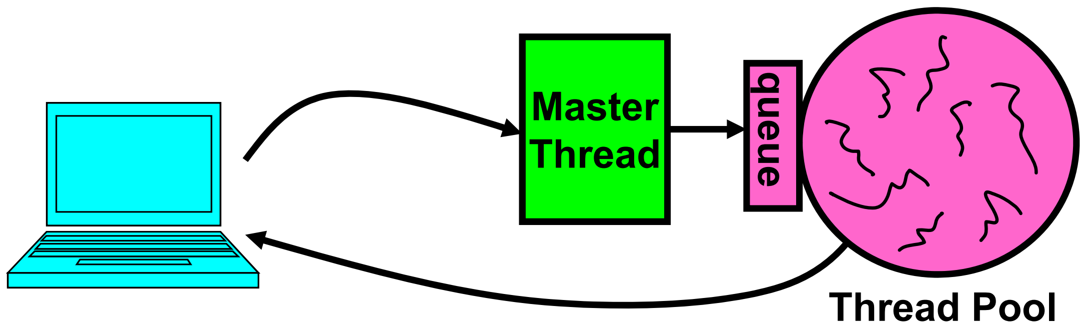
- Now, use a single process
- Multithreaded (cooperating) version:

```
serverLoop() {
     connection = AcceptCon();
     ThreadFork(ServiceWebPage(),connection);
  }
```

- Looks almost the same, but has many advantages:
  – Can share file caches kept in memory, results of CGI scripts, other things
  – Threads are *much* cheaper to create than processes, so this has a lower per-request overhead
- Question: would a user-level (say one-to-many) thread package make sense here?
  – When one request blocks on disk, all block…
- What about Denial of Service attacks or digg / Slash-dot effects?

Recall: Thread Pools: Bounded Concurrency

- Problem with previous version: Unbounded Threads
  – When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded "pool" of worker threads, representing the maximum level of multiprogramming



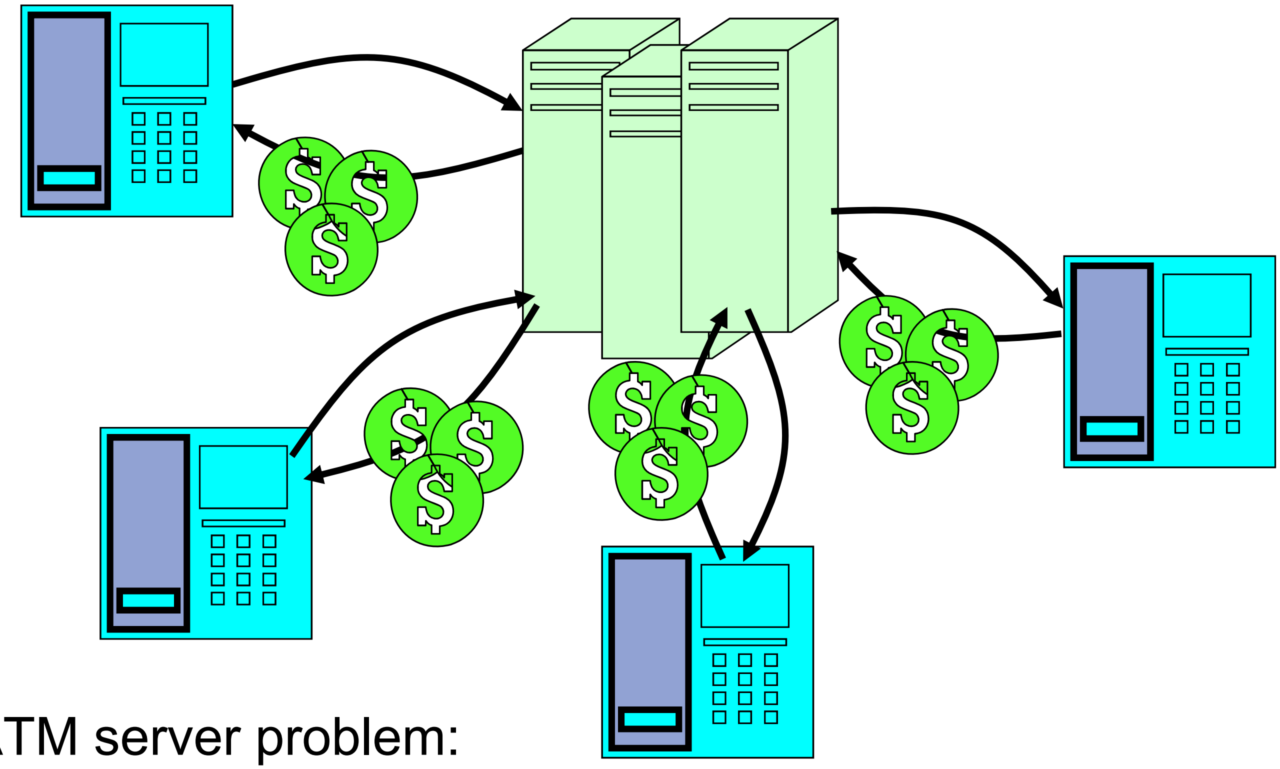

```
master() {
   allocThreads(worker,queue);
   while(TRUE) {
      con=AcceptCon();
      Enqueue(queue,con);
      wakeUp(queue);
   }
}
```

```
worker(queue) {
   while(TRUE) {
      con=Dequeue(queue);
      if (con==null)
         sleepOn(queue);
      else
         ServiceWebPage(con);
   }
}
```

Correctness with Concurrent Threads?

- Non-determinism:
  - Scheduler can run threads in **any order**
  - Scheduler can switch threads **at any time**
  - This can make testing very difficult
- *Independent Threads*
  - No state shared with other threads
  - Deterministic, reproducible conditions
- *Cooperating Threads*
  - Shared state between multiple threads
- **Goal: Correctness by Design**

ATM Bank Server



- ATM server problem:
  - Service a set of requests
  - Do so without corrupting database
  - Don't hand out too much money

ATM bank server example

- Suppose we wanted to implement a server process to handle requests from an ATM network:

```
BankServer() {
   while (TRUE) {
      ReceiveRequest(&op, &acctId, &amount);
      ProcessRequest(op, acctId, amount);
   }
}
ProcessRequest(op, acctId, amount) {
   if (op == deposit) Deposit(acctId, amount);
   else if …
}
Deposit(acctId, amount) {
   acct = GetAccount(acctId); /* may use disk I/O */
   acct->balance += amount;
   StoreAccount(acct); /* Involves disk I/O */
}
```

- How could we speed this up?
  - More than one request being processed at once
  - Event driven (overlap computation and I/O)
  - Multiple threads (multi-proc, or overlap comp and I/O)

Event Driven Version of ATM server

- Suppose we only had one CPU
  - Still like to overlap I/O with computation
  - Without threads, we would have to rewrite in event-driven style
- Example

```
BankServer() {
   while(TRUE) {
      event = WaitForNextEvent();
      if (event == ATMRequest)
         StartOnRequest();
      else if (event == AcctAvail)
         ContinueRequest();
      else if (event == AcctStored)
         FinishRequest();
   }
}
```

  - This technique is used for graphical programming
- Complication:
  - What if we missed a blocking I/O step?
  - What if we have to split code into hundreds of pieces which could be blocking?

Can Threads Make This Easier?

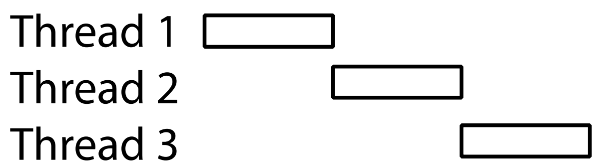
- Threads yield overlapped I/O and computation without "deconstructing" code into non-blocking fragments
  - One thread per request
- Requests proceeds to completion, blocking as required:

```
Deposit(acctId, amount) {
  acct = GetAccount(actId); /* May use disk I/O */
  acct->balance += amount;
  StoreAccount(acct);      /* Involves disk I/O */
}
```

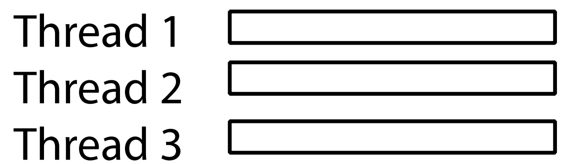
- Unfortunately, shared state can get corrupted:

Thread 1	Thread 2
`load r1, acct->balance`	
	`load r1, acct->balance`
	`add r1, amount2`
	`store r1, acct->balance`
`add r1, amount1`	
`store r1, acct->balance`	

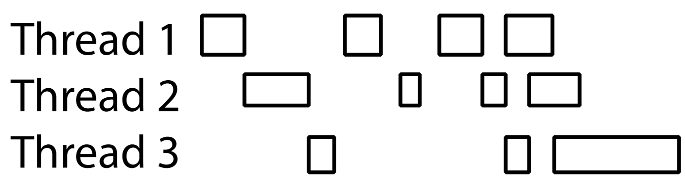
Recall: Possible Executions



a) One execution



b) Another execution



c) Another execution

Problem is at the Lowest Level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, what about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  – What are the possible values of x?
- Or, what are the possible values of x below?

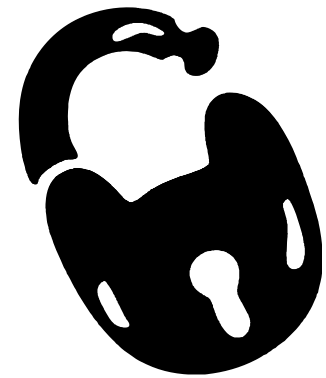
Thread A	Thread B
x = 1;	x = 2;

  – X could be 1 or 2 (non-deterministic!)
  – Could even be 3 for serial processors:
    » Thread A writes 0001, B writes 0010 → scheduling order ABABABBA yields 3!

Atomic Operations

- To understand a concurrent program, we need to know what the underlying indivisible operations are!
- Atomic Operation: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
  - Consequently – weird example that produces “3” on previous slide can’t happen
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array

Locks



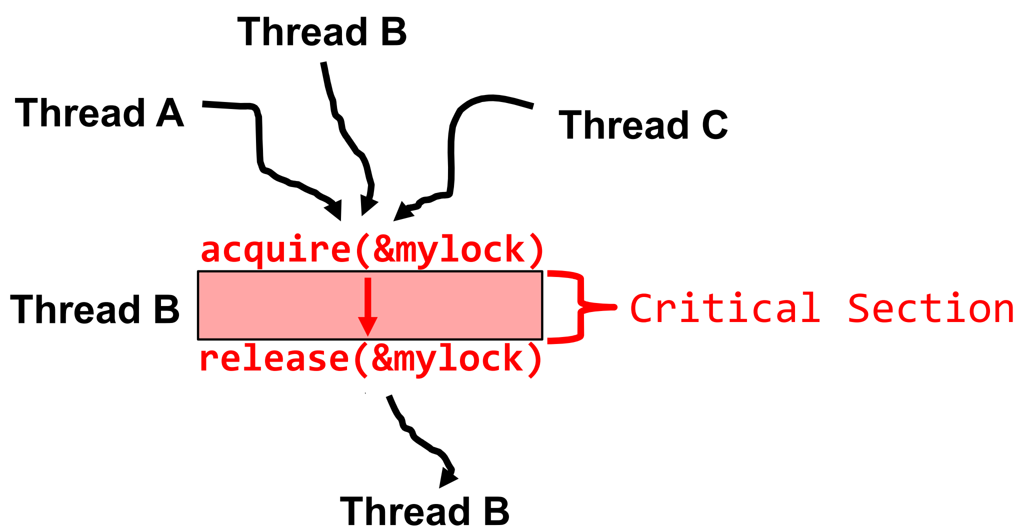
- Lock: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - » Important idea: all synchronization involves waiting
- Locks need to be allocated and initialized:
  - `structure Lock mylock` or `pthread_mutex_t mylock;`
  - `lock_init(&mylock)` or `mylock = PTHREAD_MUTEX_INITIALIZER;`
- Locks provide two **atomic** operations:
  - acquire(&mylock) – wait until lock is free; then mark it as busy
    - » After this returns, we say the calling thread *holds* the lock
  - release(&mylock) – mark lock as free
    - » Should only be called by a thread that currently holds the lock
    - » After this returns, the calling thread no longer holds the lock

Fix banking problem with Locks!

- Identify critical sections (atomic instruction sequences) and add locking:

```
Deposit(acctId, amount) {
  acquire(&mylock)          // Wait if someone else in critical section!
  acct = GetAccount(actId);
  acct->balance += amount;
  StoreAccount(acct);
  release(&mylock)          // Release someone into critical section
}
```

Critical Section



Threads serialized by lock through critical section. Only one thread at a time

- Must use SAME lock (`mylock`) with all of the methods (Withdraw, etc…)
  - Shared with all threads!

Conclusion

- Recall: **`open()`**, **`read()`**, **`write()`**, and **`close()`** used for wide variety of I/O:
  - – Files on disk
  - – Devices (terminals, printers, etc.)
  - – Regular files on disk
  - – Networking (sockets)
  - – Local interprocess communication (pipes, sockets)
- Processes have two parts
  - – Threads (Concurrency)
  - – Address Spaces (Protection)
- Stack is essential part of computation
  - – Every thread has two stacks: user-level (in address space) and kernel
  - – The kernel stack + support often called the “kernel thread”
- 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

Conclusion (cont)

- 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
- Upcoming: how to ensure correct behavior through synchronization