Operating Systems (Honor Track)

Synchronization 1: Concurrency

Xin Jin

Spring 2023

Acknowledgments: Ion Stoica, Berkeley CS 162

Recap: One example of this pattern: POSIX/Unix PIPE

```
write(wfd, wbuf, wlen);
```


```
n = read(rfd, rbuf, rmax);
```

- Memory Buffer is finite:
  - If producer (A) tries to write when buffer full, it *blocks* (Put sleep until space)
  - If consumer (B) tries to read when buffer empty, it *blocks* (Put to sleep until data)

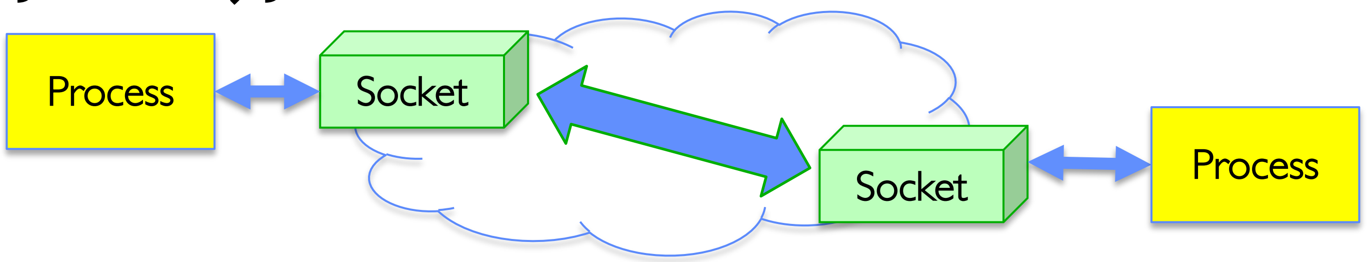
```
int pipe(int fileds[2]);
```

- Allocates two new file descriptors in the process
- Writes to `fileds[1]` read from `fileds[0]`
- Implemented as a fixed-size queue

Recap: The Socket Abstraction: Endpoint for Communication

- **Key Idea:** Communication across the world looks like File I/O

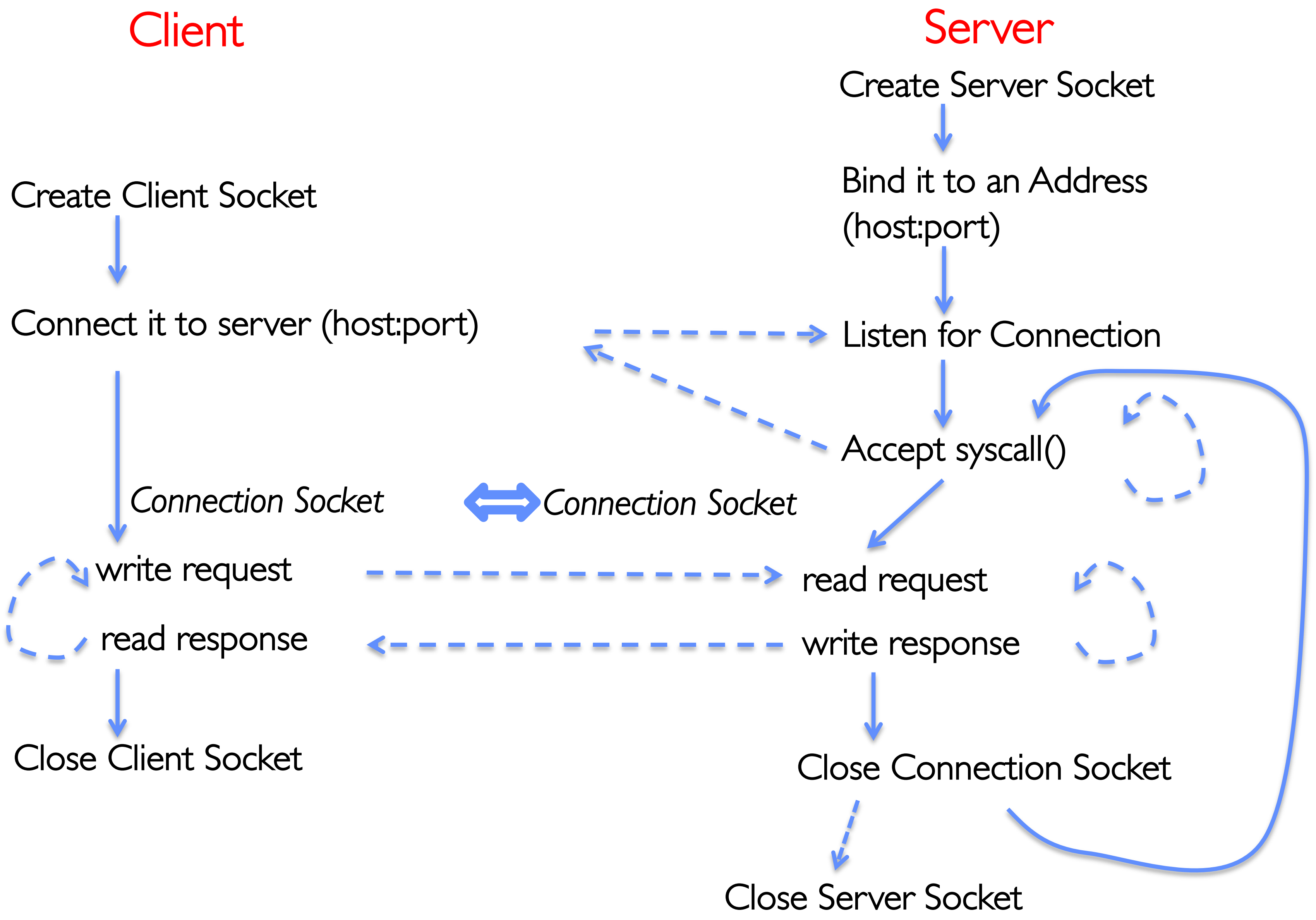
```
write(wfd, wbuf, wlen);
```



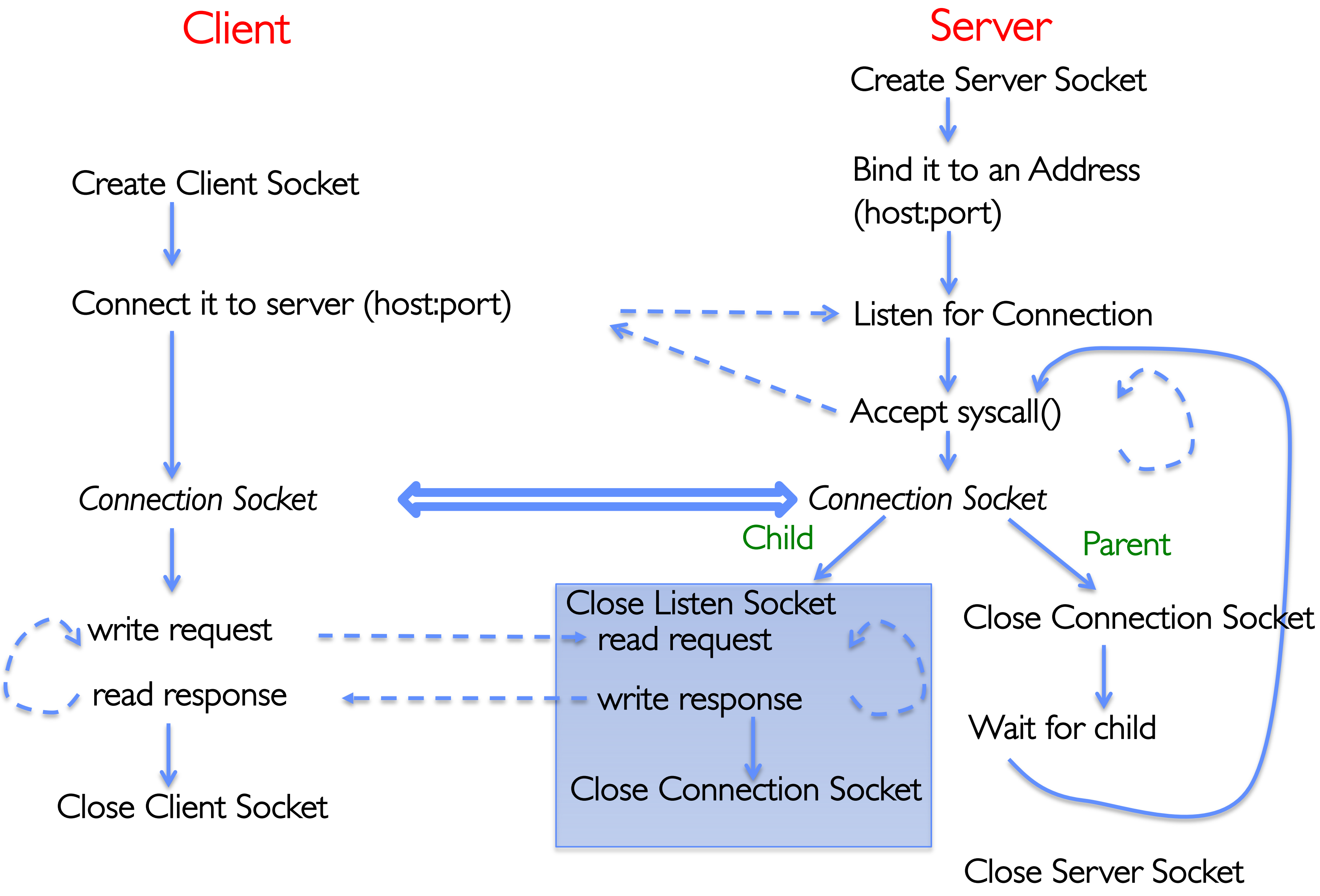

```
n = read(rfd, rbuf, rmax);
```

- Sockets: Endpoint for Communication
  - Queues to temporarily hold results
- Connection: Two Sockets Connected Over the network ⇒ IPC over network!
  - How to **open()**?
  - What is the namespace?
  - How are they connected in time?

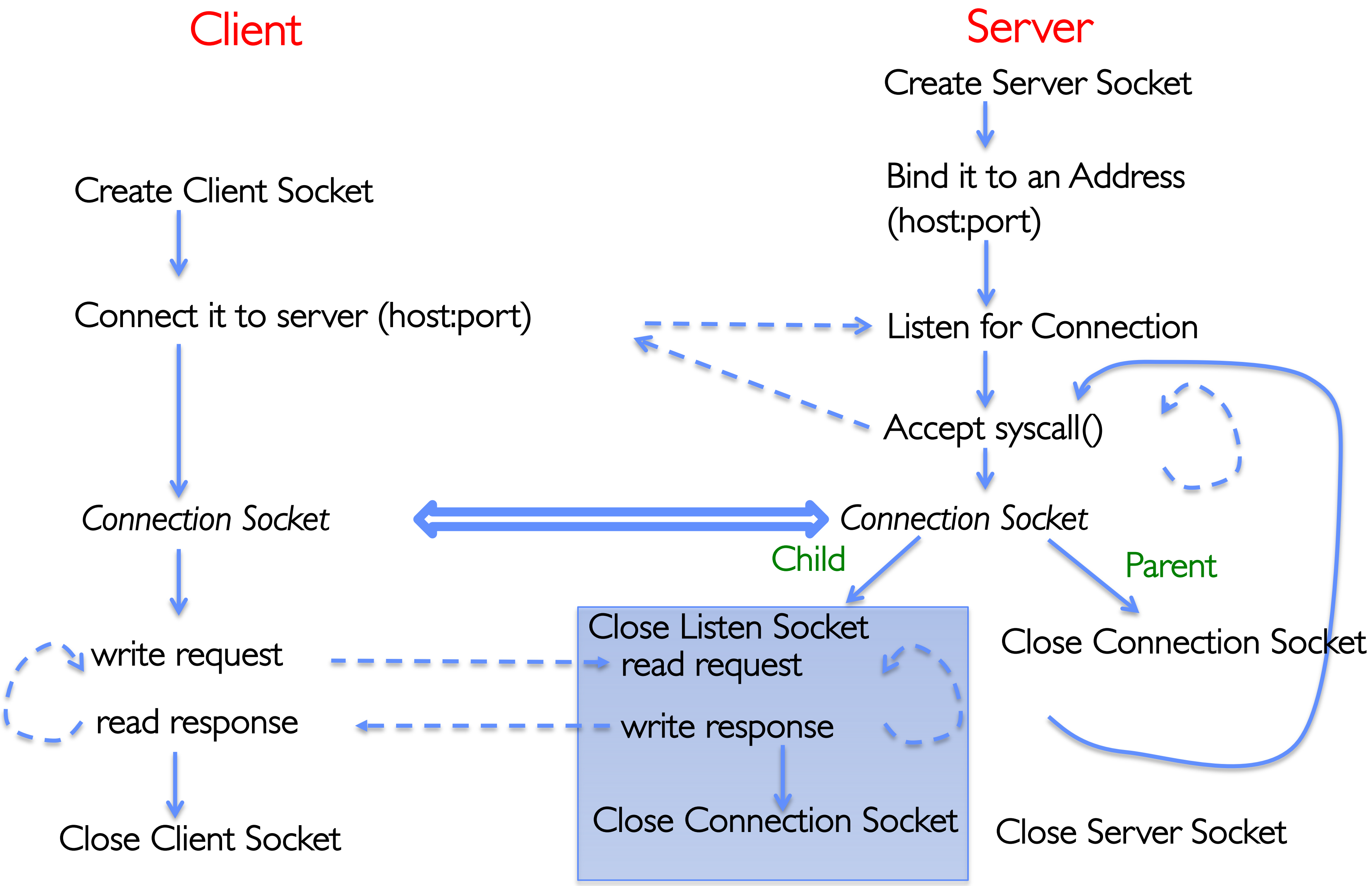
Recap: Sockets in concept



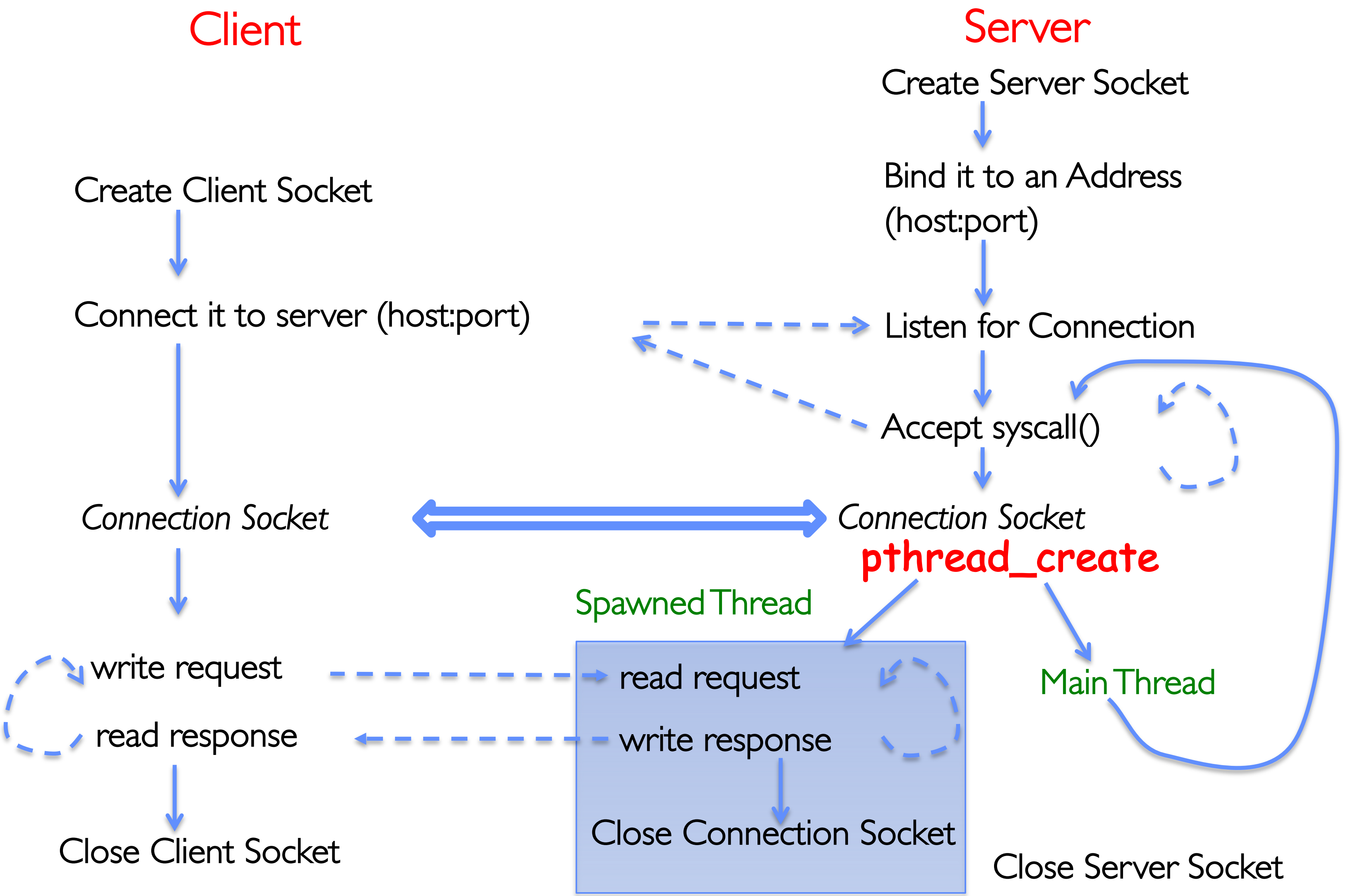
Recap: Sockets With Protection (each connection has own process)



Recap: Sockets With Protection and Concurrency



Recap: Sockets with Concurrency, without Protection



Group Discussion

- Topic: Pipes vs. Sockets
  - What is a pipe? What is a socket?
  - What are similar between pipes and sockets?
  - What are different between pipes and sockets?

- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

How to Read

You May Think You Already Know
How To READ, But…

You Spend a Lot of Time Reading

- Reading for undergrad/grad classes
- Reviewing conference submissions
- Giving colleagues feedback
- Keeping up with your field
- Staying broadly educated
- Transitioning into a new area
- Learning how to write better papers

It is worthwhile to learn to read *effectively*

Keshav's Three-Pass Approach: Step 1

- A ten-minute scan to get the general idea
  - Title, abstract, and introduction
  - Section and subsection titles
  - Conclusion and bibliography
- What to learn: the five C's
  - Category: What type of paper is it?
  - Context: What body of work does it relate to?
  - Correctness: Do the assumptions seem valid?
  - Contributions: What are the main research contributions?
  - Clarity: Is the paper well-written?
- Decide whether to read further…

Keshav's Three-Pass Approach: Step 2

- A more careful, one-hour reading
  - Read with greater care, but ignore details like proofs
  - Figures, diagrams, and illustrations
  - Mark relevant references for later reading
- Grasp the content of the paper
  - Be able to summarize the main idea
  - Identify whether you can (or should) fully understand
- Decide whether to
  - Abandon reading in greater depth
  - Read background material before proceeding further
  - Persevere and continue for a third pass

Keshav’s Three-Pass Approach: Step 3

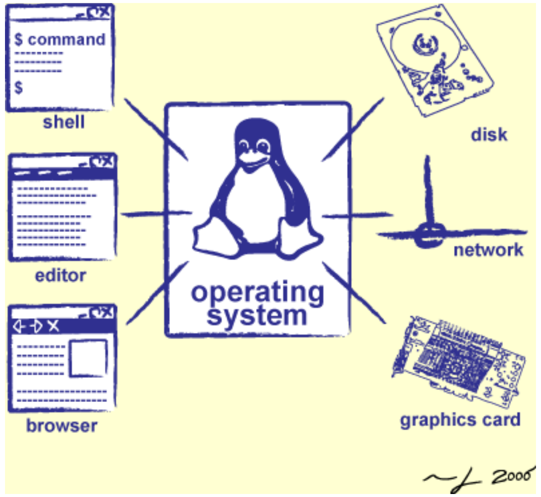
- Several-hour virtual re-implementation of the work
  - Making the same assumptions, recreate the work
  - Identify the paper’s innovations and its failings
  - Identify and challenge every assumption
  - Think how you would present the ideas yourself
  - Jot down ideas for future work
- When should you read this carefully?
  - Reviewing for a conference or journal
  - Giving colleagues feedback on a paper
  - Understanding a paper closely related to your research
  - Deeply understanding a classic paper in the field

Other Tips for Reading Papers

- Read at the right level for what you need
  - “Work smarter, not harder”
- Read at the right time of day
  - When you are fresh, not sleepy
- Read in the right place
  - Where you are not distracted, and have enough time
- Read actively
  - With a purpose (what is your goal?)
  - With a pen or computer to take notes
- Read critically
  - Think, question, challenge, critique, …

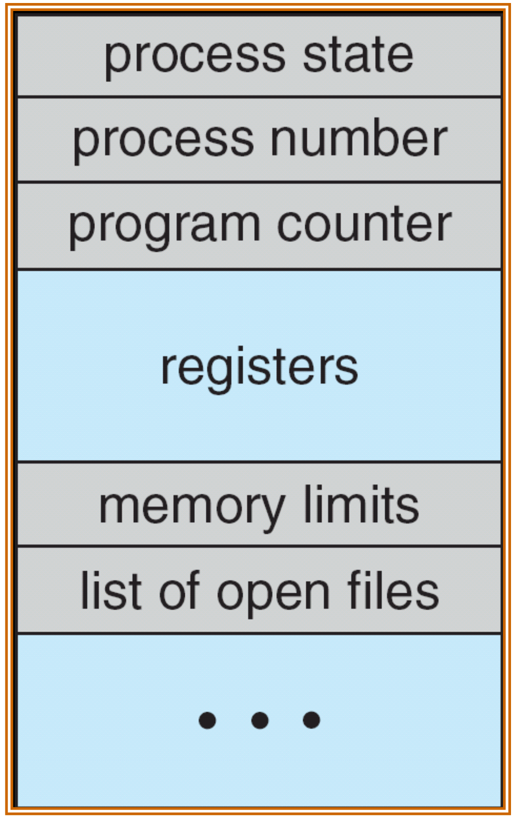
Agenda: Synchronization

- How does an OS provide concurrency through threads?
  - Brief discussion of process/thread states and scheduling
  - High-level discussion of how stacks contribute to concurrency
- Introduce needs for synchronization
- Discussion of Locks and Semaphores



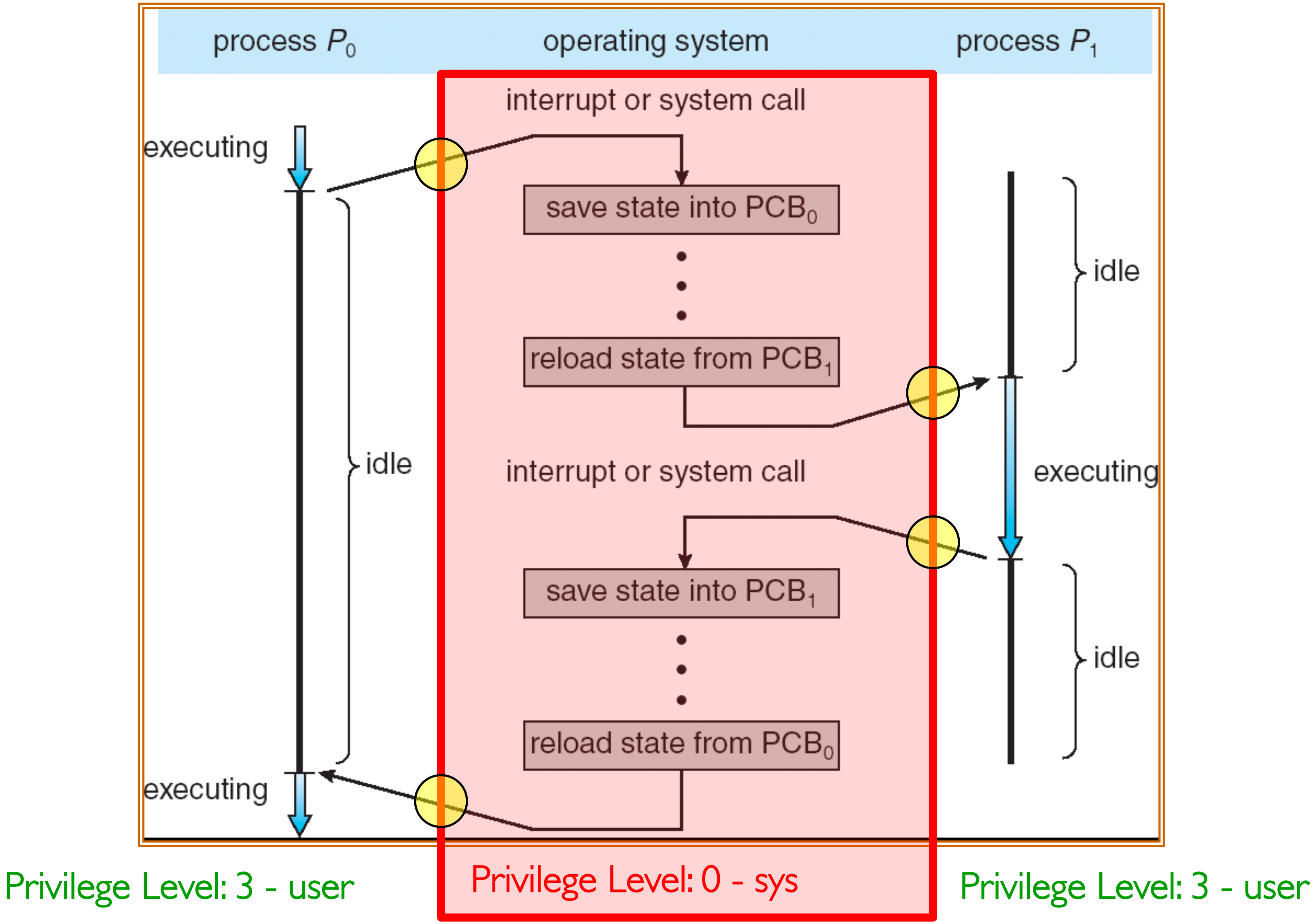
Multiplexing Processes: The Process Control Block

- 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

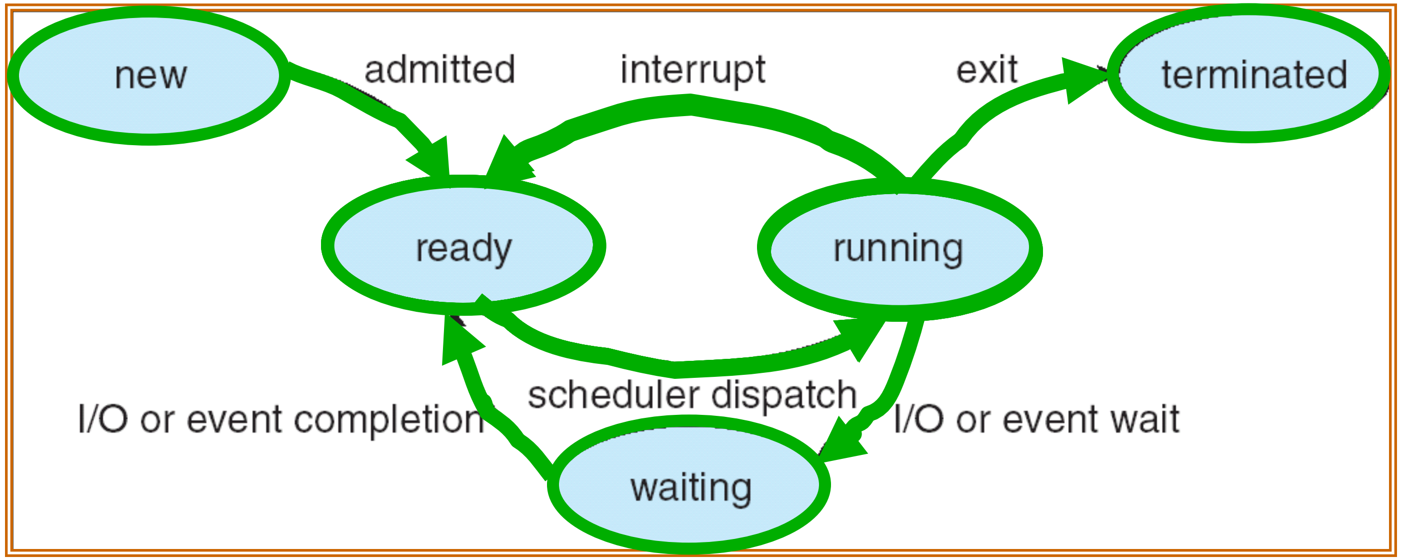


Process Control Block

Context Switch

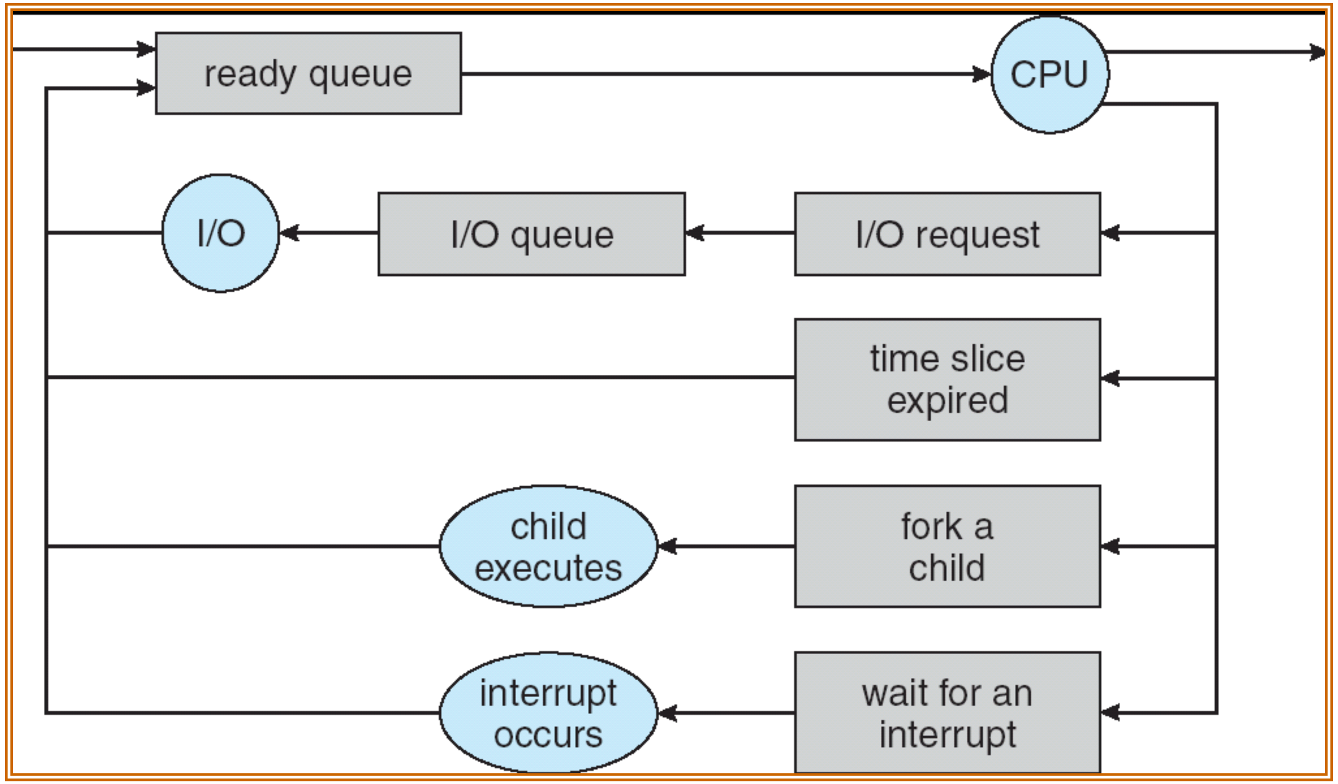


Lifecycle of a Process or Thread



- As a process executes, it changes state:
  - new: The process/thread 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

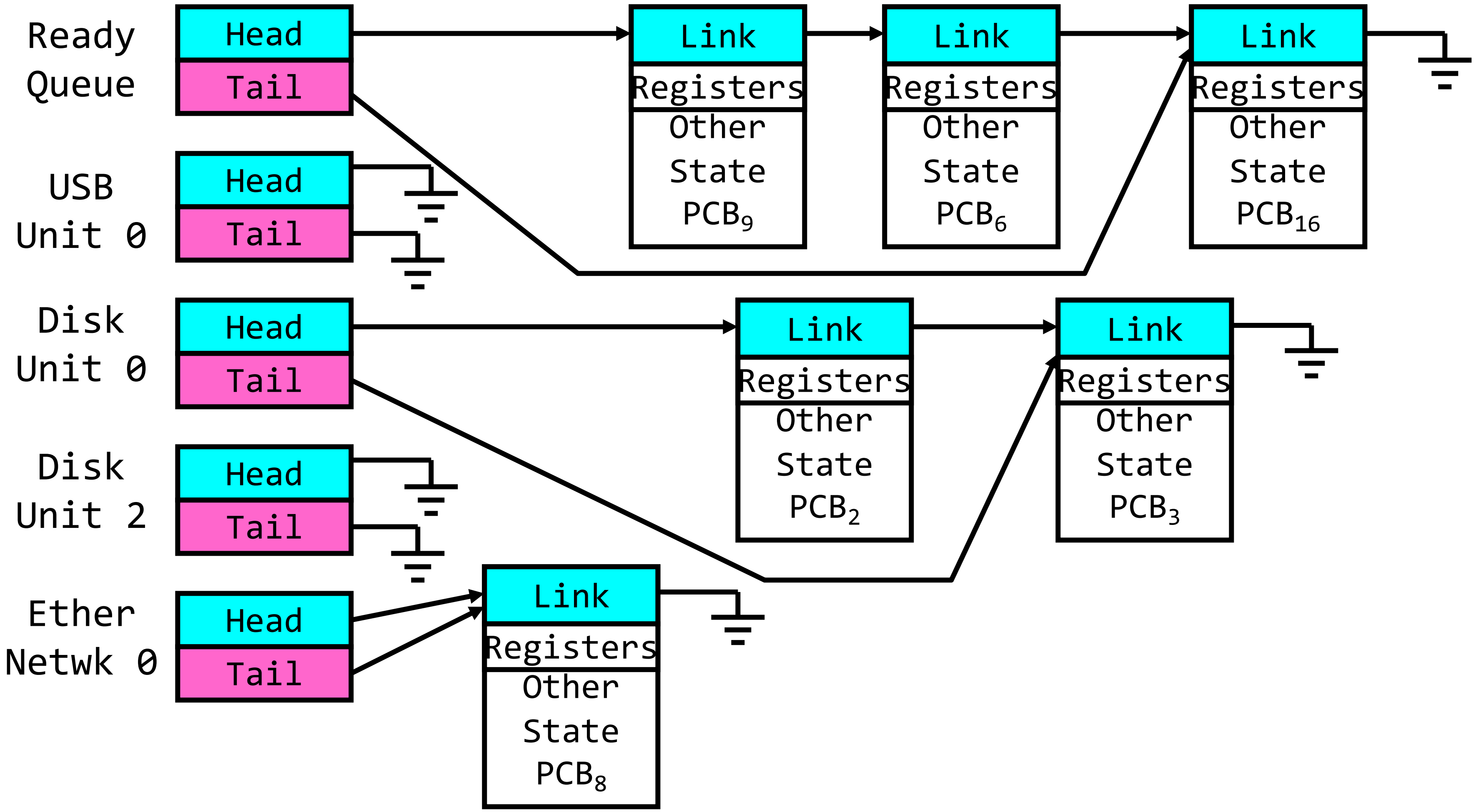
Scheduling: All About Queues



- PCBs move from queue to queue
- **Scheduling:** which order to remove from queue
  - Much more on this soon

Ready Queue And Various I/O Device Queues

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

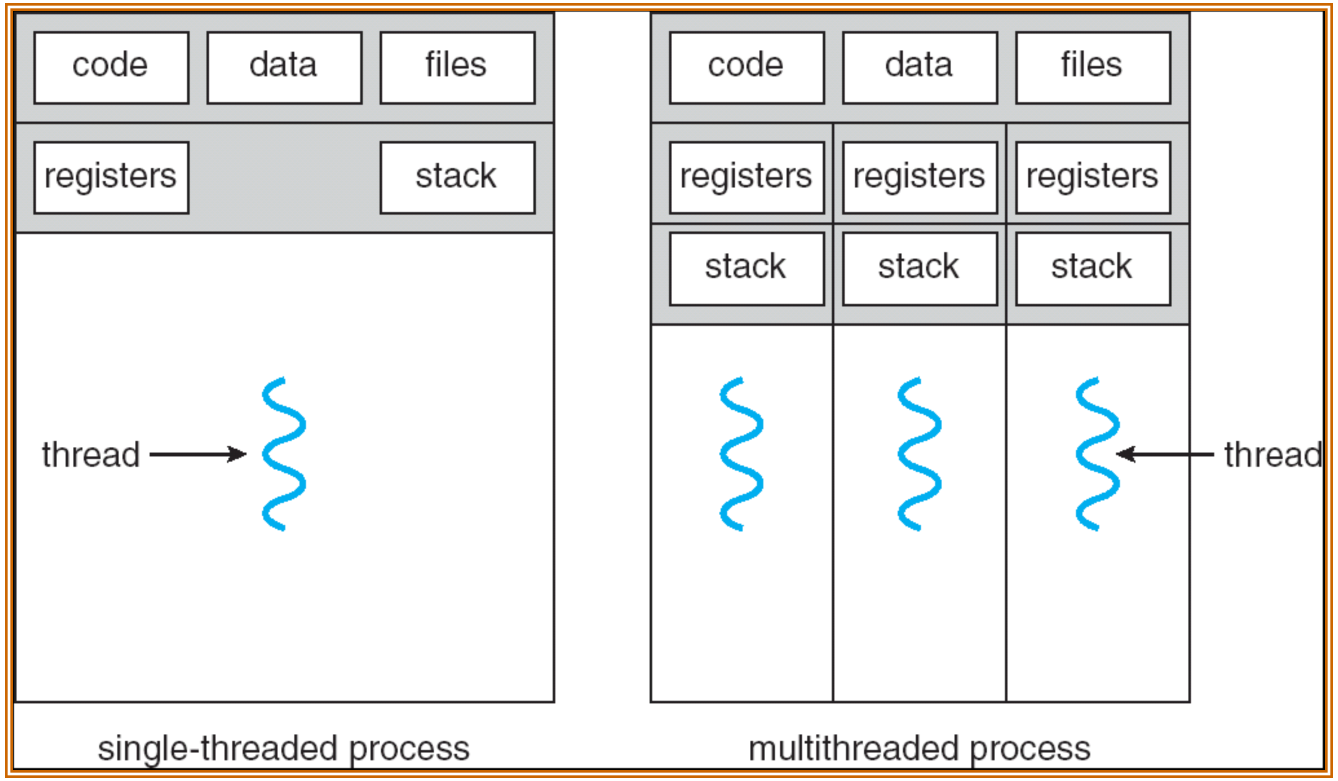


Scheduler

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

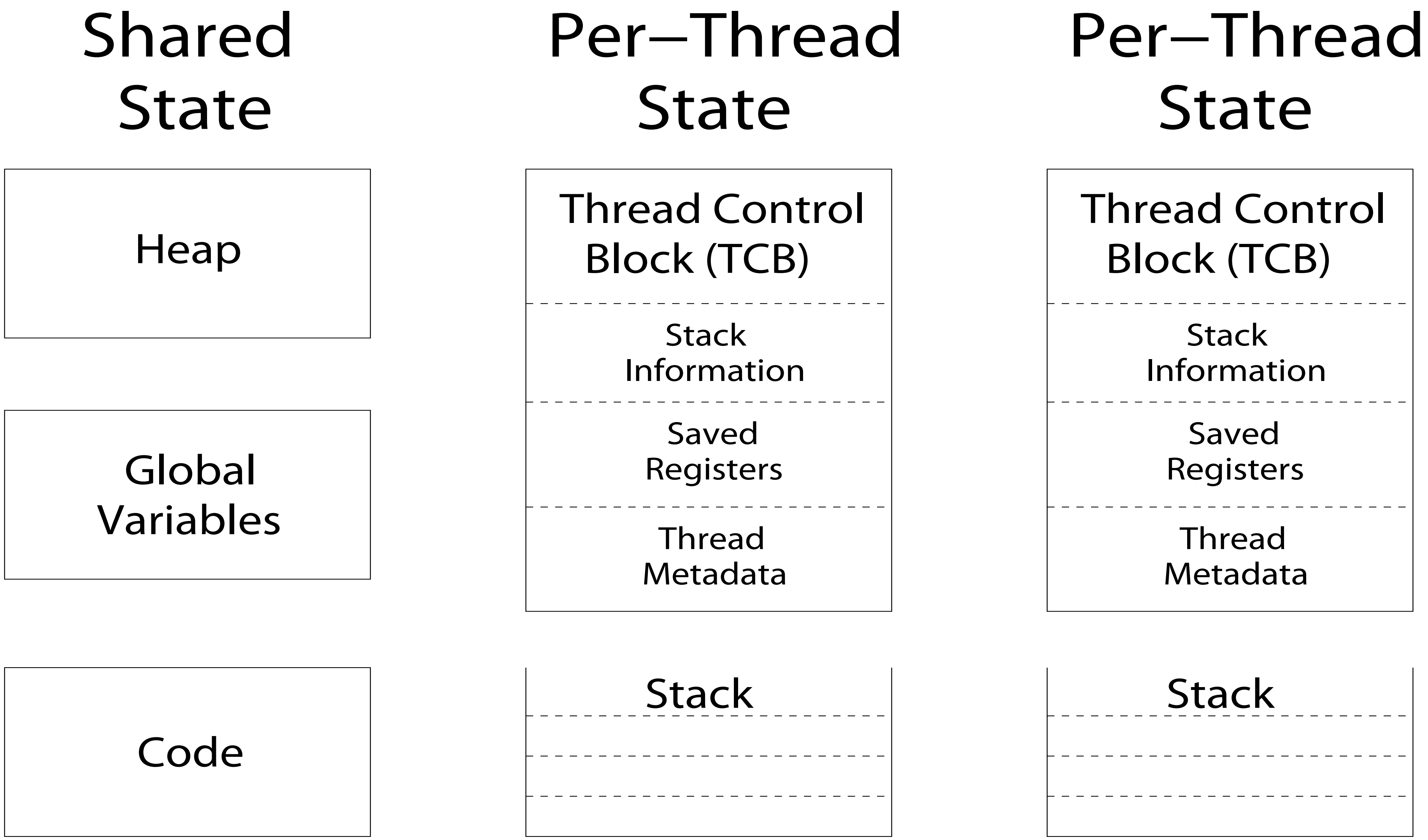
- Scheduling: Mechanism for deciding which processes/threads receive the CPU
- Lots of different scheduling policies provide …
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or …

Recall: Single and Multithreaded Processes



- Threads encapsulate concurrency
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

Shared vs. Per-Thread State



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

Running a thread

Consider first portion: `RunThread()`

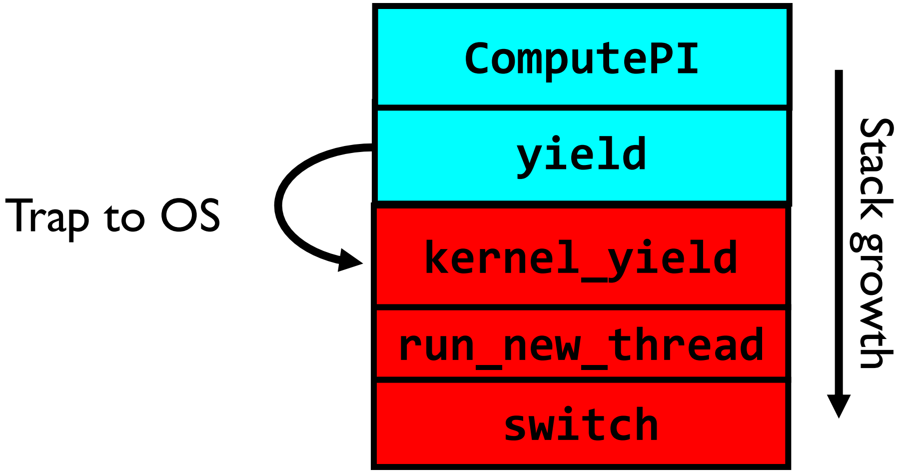
- How do I run a thread?
  - Load its state (registers, PC, stack pointer) into CPU
  - Load environment (virtual memory space, etc)
  - Jump to the PC
- How does the dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets *preempted*

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

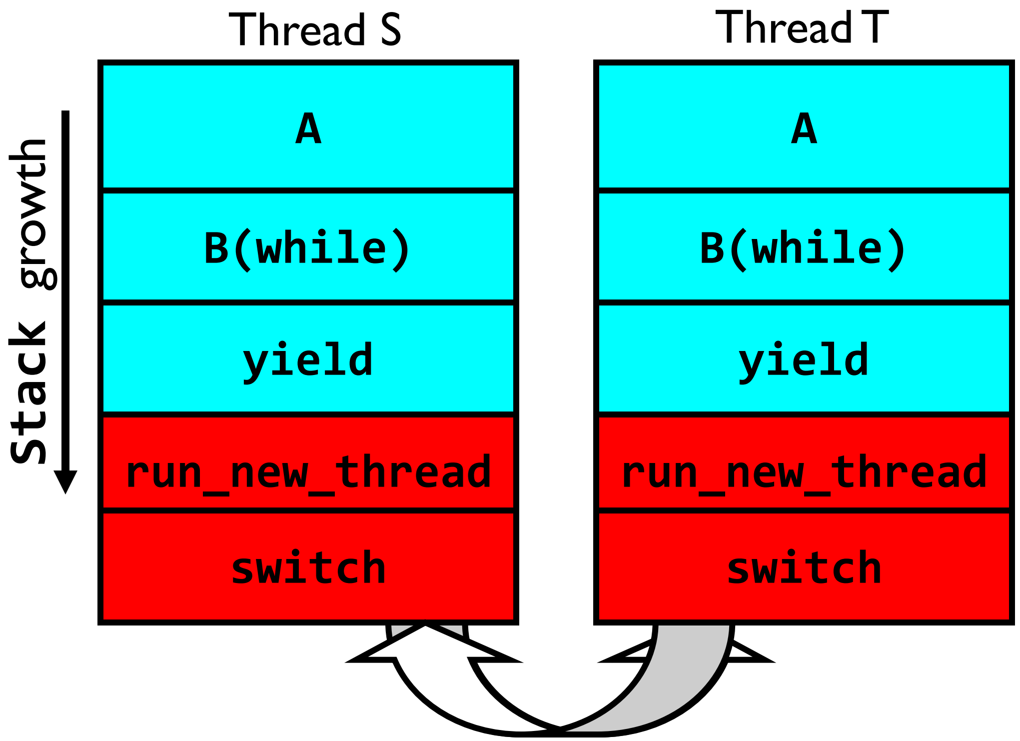
What Do the Stacks Look Like?

- Consider the following code blocks:

```
func A() {
    B();

}
func 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?
  - Very challenging! Too many combinations and interleavings
- 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

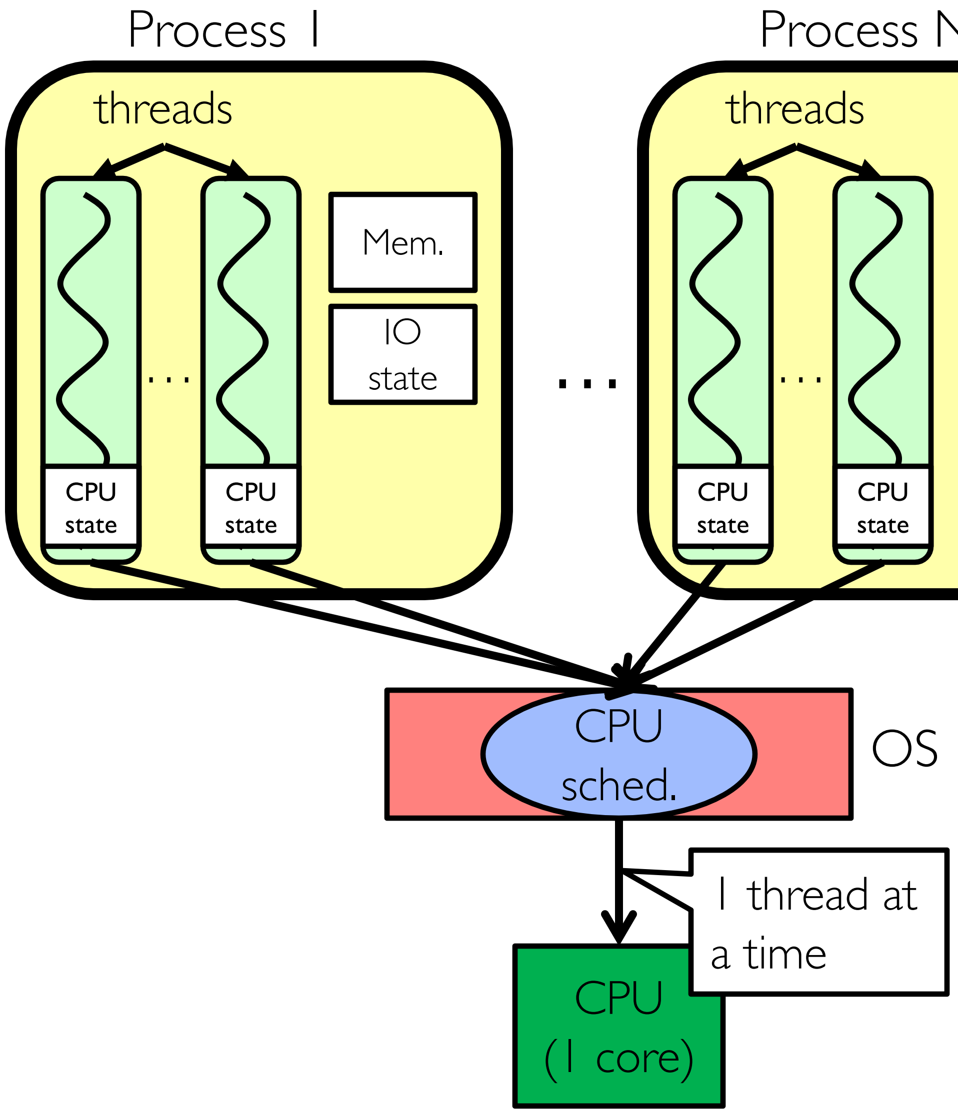
Aren't we still switching contexts?

- 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 μs
  - Switching between threads: 100 ns

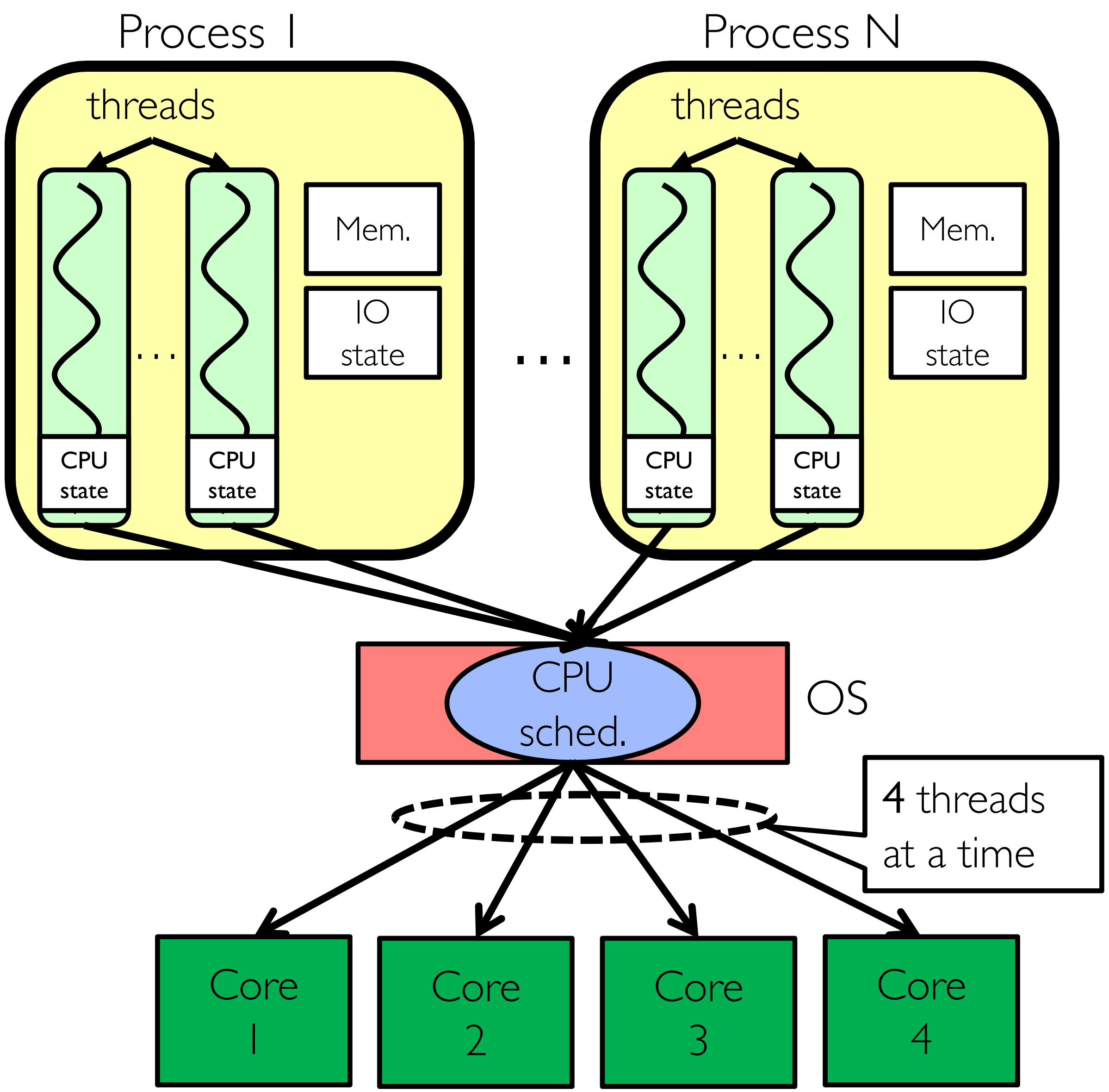
- Even cheaper: switch threads (using “yield”) in user-space!

Processes vs. Threads



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

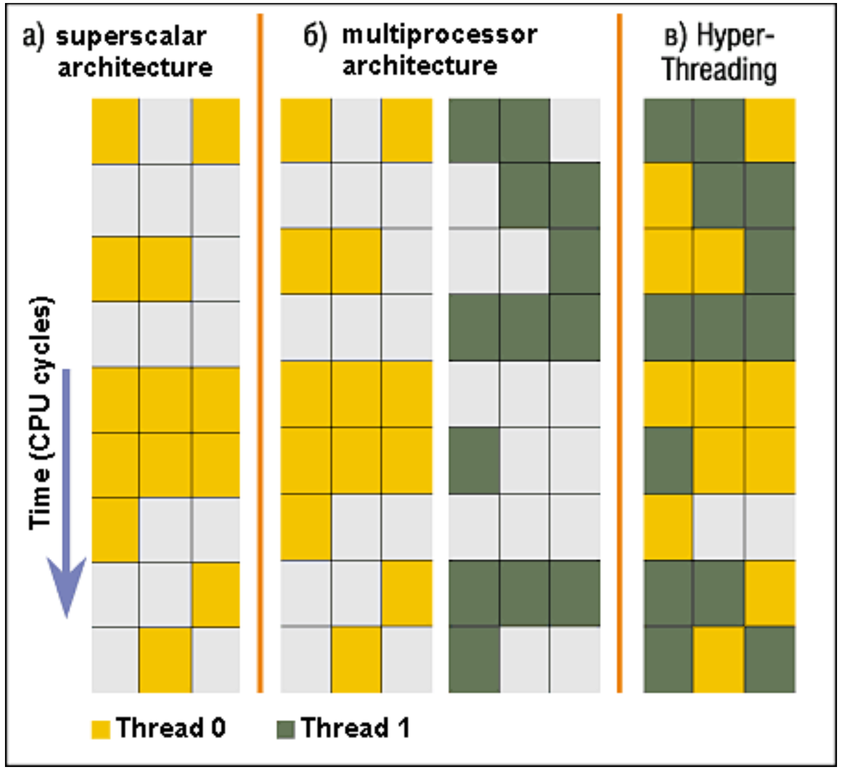
Processes vs. Threads



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

Simultaneous MultiThreading/Hyperthreading

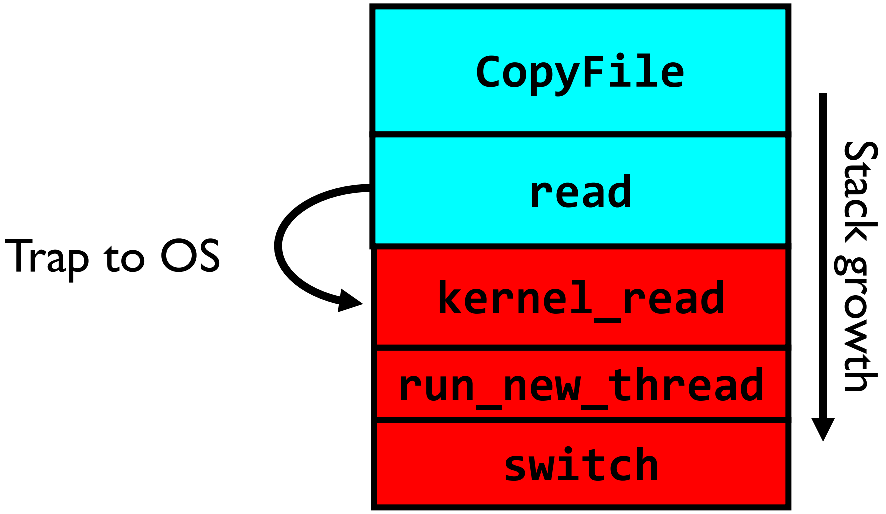
- Hardware scheduling technique
  - Superscalar processors can execute multiple instructions that are independent.
  - Hyperthreading duplicates register state to make a second "thread," allowing more instructions to run.
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!



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

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

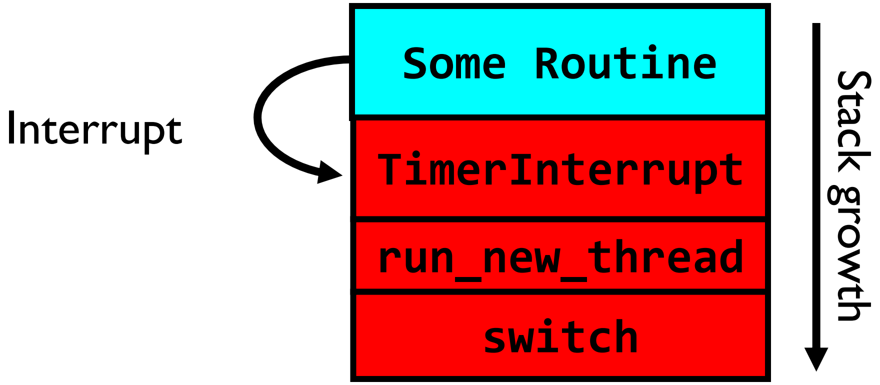
- What happens if thread never does any I/O, never waits, and never yields control?
  - Could the `ComputePI` program grab all resources and never release the processor?
    - » What if it didn't print to console?
  - Must find way that dispatcher can regain control!

- 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

Use of Timer Interrupt to Return Control

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



- Timer Interrupt routine:

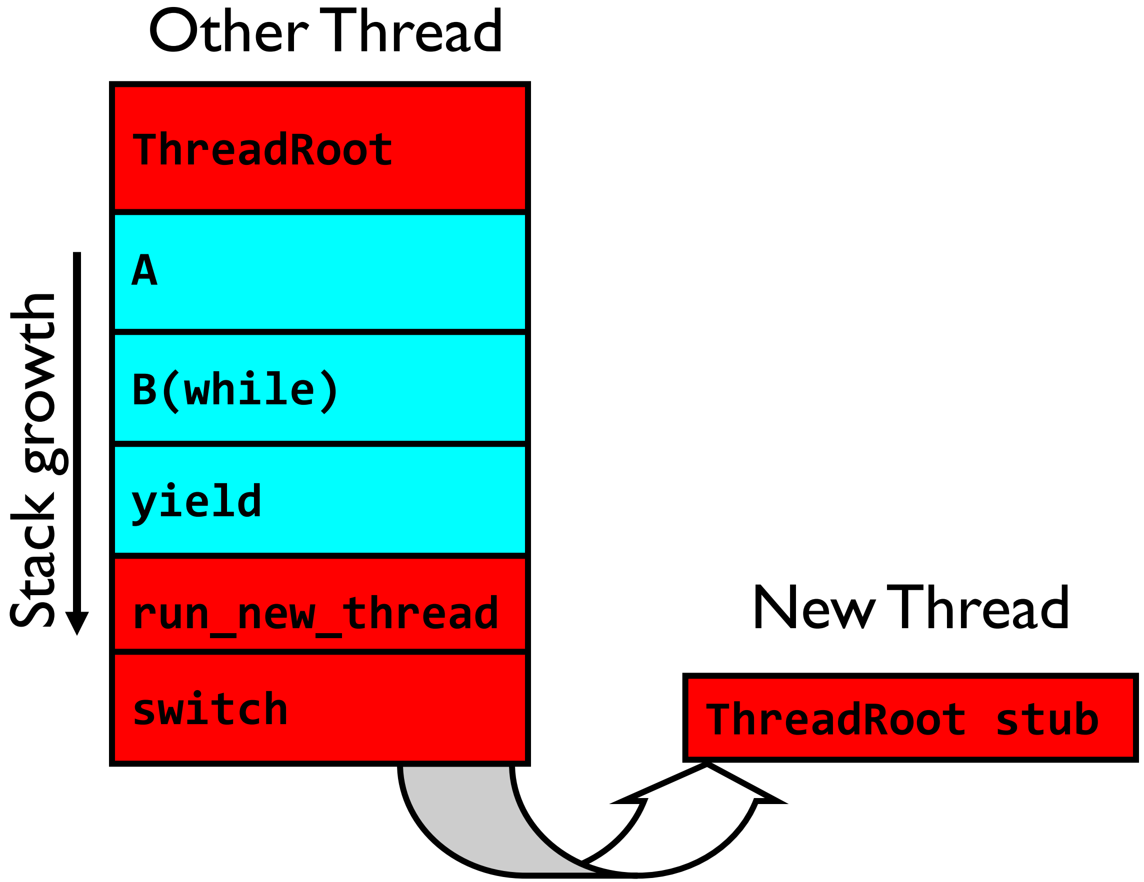
```
TimerInterrupt() {
   DoPeriodicHouseKeeping();
   run_new_thread();
}
```

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

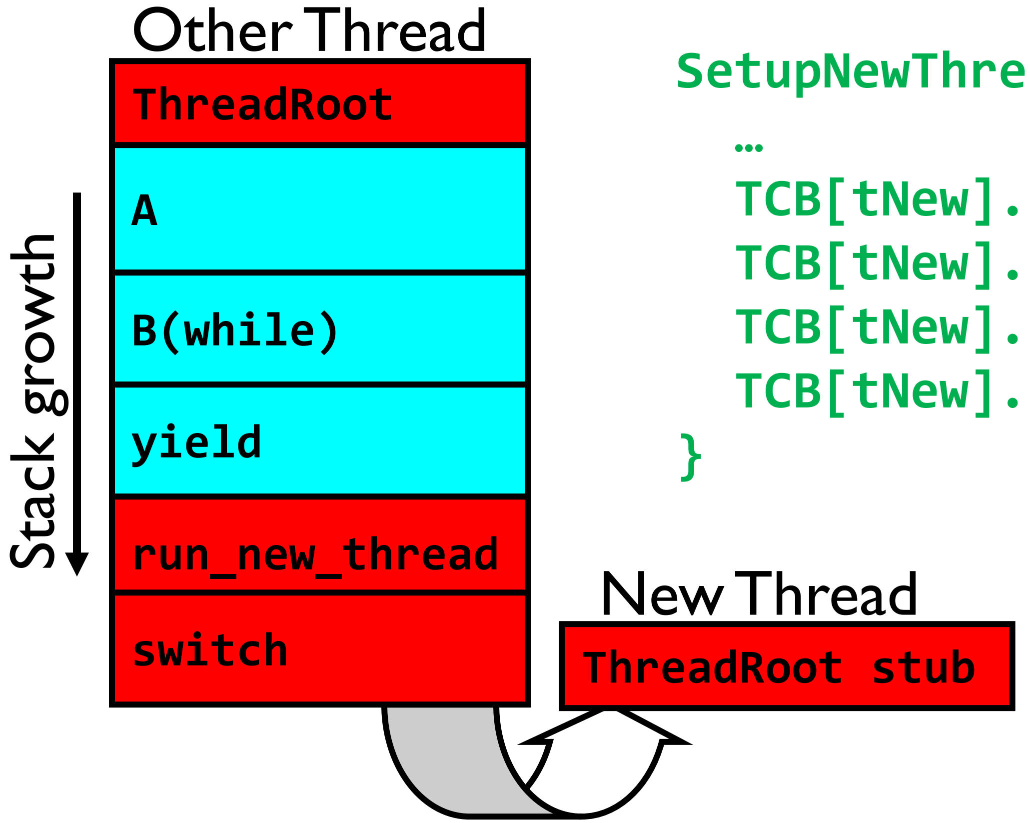
Initial Stack

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?



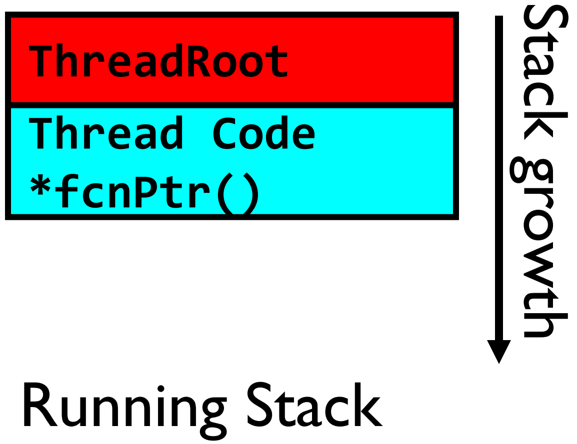

```
SetupNewThread(tNew) {
   …
   TCB[tNew].regs.sp = newStackPtr;
   TCB[tNew].regs.retpc = &ThreadRoot;
   TCB[tNew].regs.r0 = fcnPtr
   TCB[tNew].regs.r1 = fcnArgPtr
}
```

- 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
  - 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

Shinjuku: Preemptive Scheduling for Microsecond-Scale Tail Latency

Kostis Kaffes, Timothy Chong, Jack Tigar Humphries,
Adam Belay, David Mazières, Christos Kozyrakis

Stanford MAST

NSDI'19

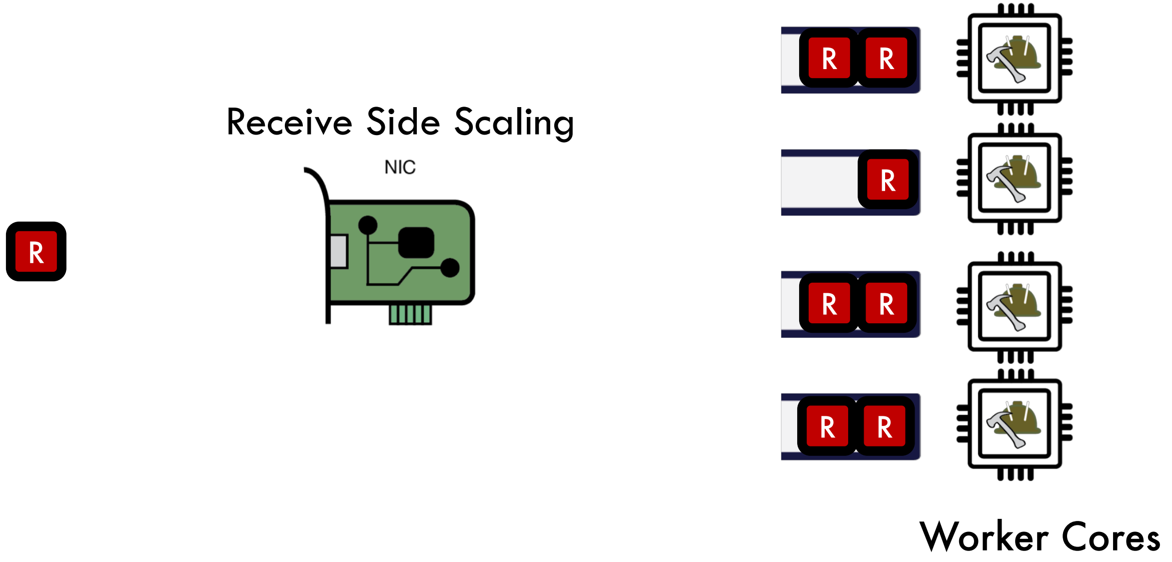
Achieving low tail latency at microsecond scale is hard

Problem: High OS overheads

Solution: OS Bypass, polling (no interrupts), run-to-completion (no scheduling)

Distributed Queues + First Come First Serve scheduling

d-FCFS (DPDK, IX, Arrakis)

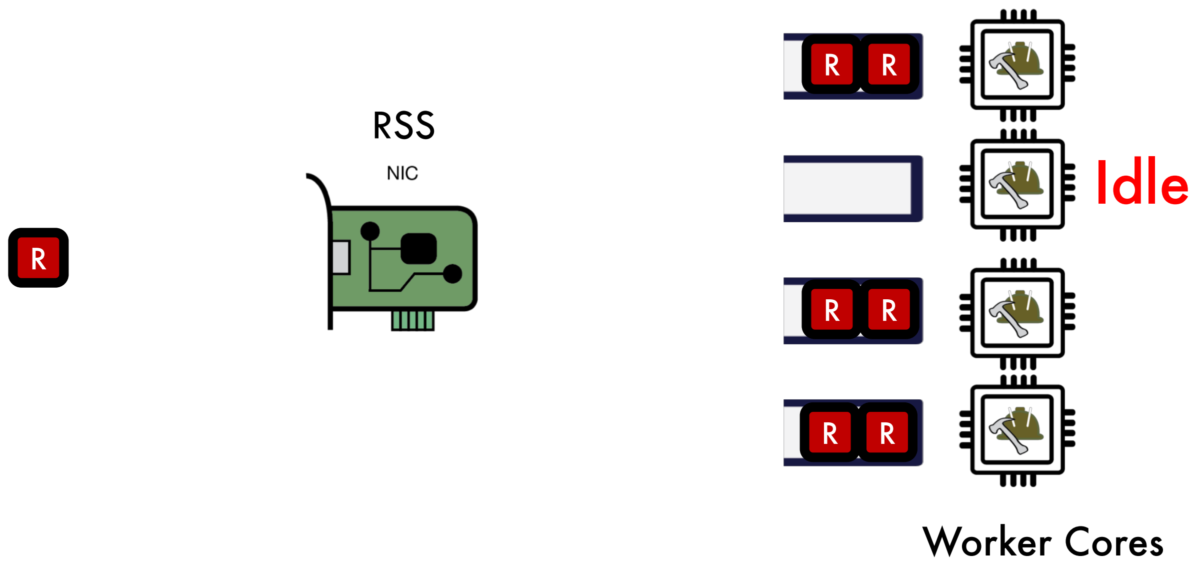


3

Acknowledgments: Kostis Kaffes

Achieving low tail latency at microsecond scale is hard

Problem: Queue imbalance because d-FCFS is not work conserving



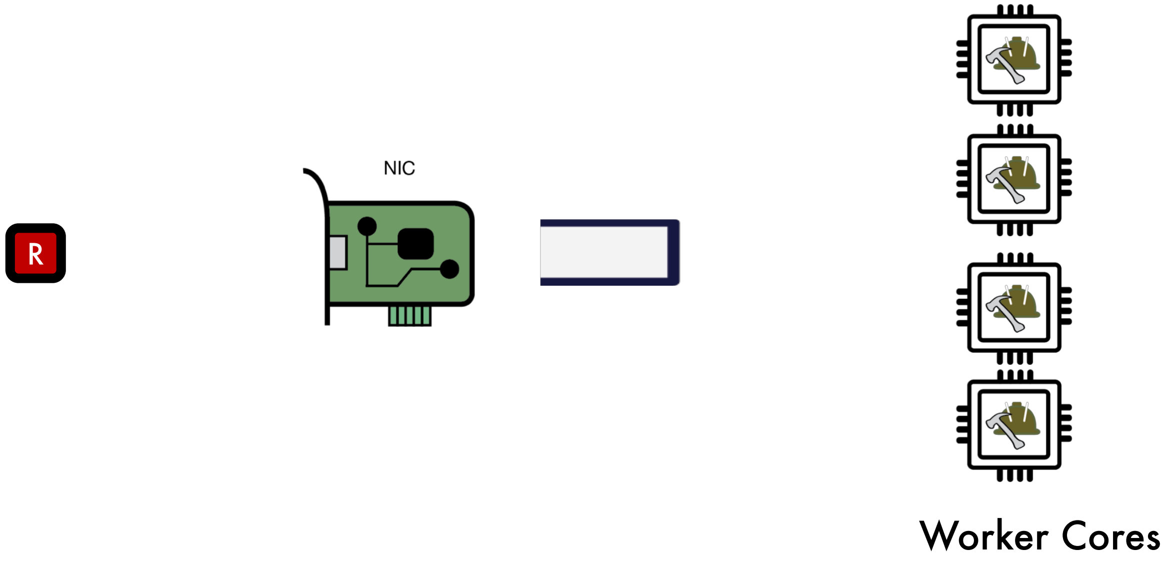
4

Acknowledgments: Kostis Kaffes

Achieving low tail latency at microsecond scale is hard

Problem: Queue imbalance because d-FCFS is not work conserving

Solution: Centralized queue - **c-FCFS**

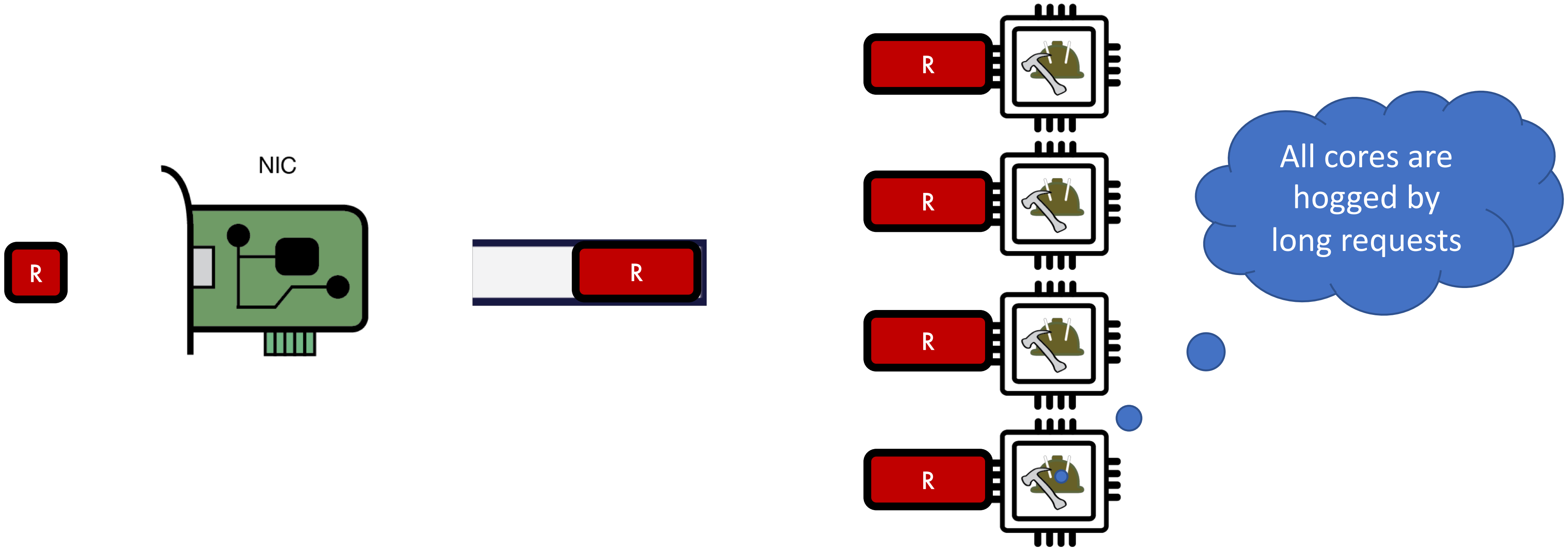


Approximation:
d-FCFS + stealing
e.g., ZygOS

5

Acknowledgments: Kostis Kaffes

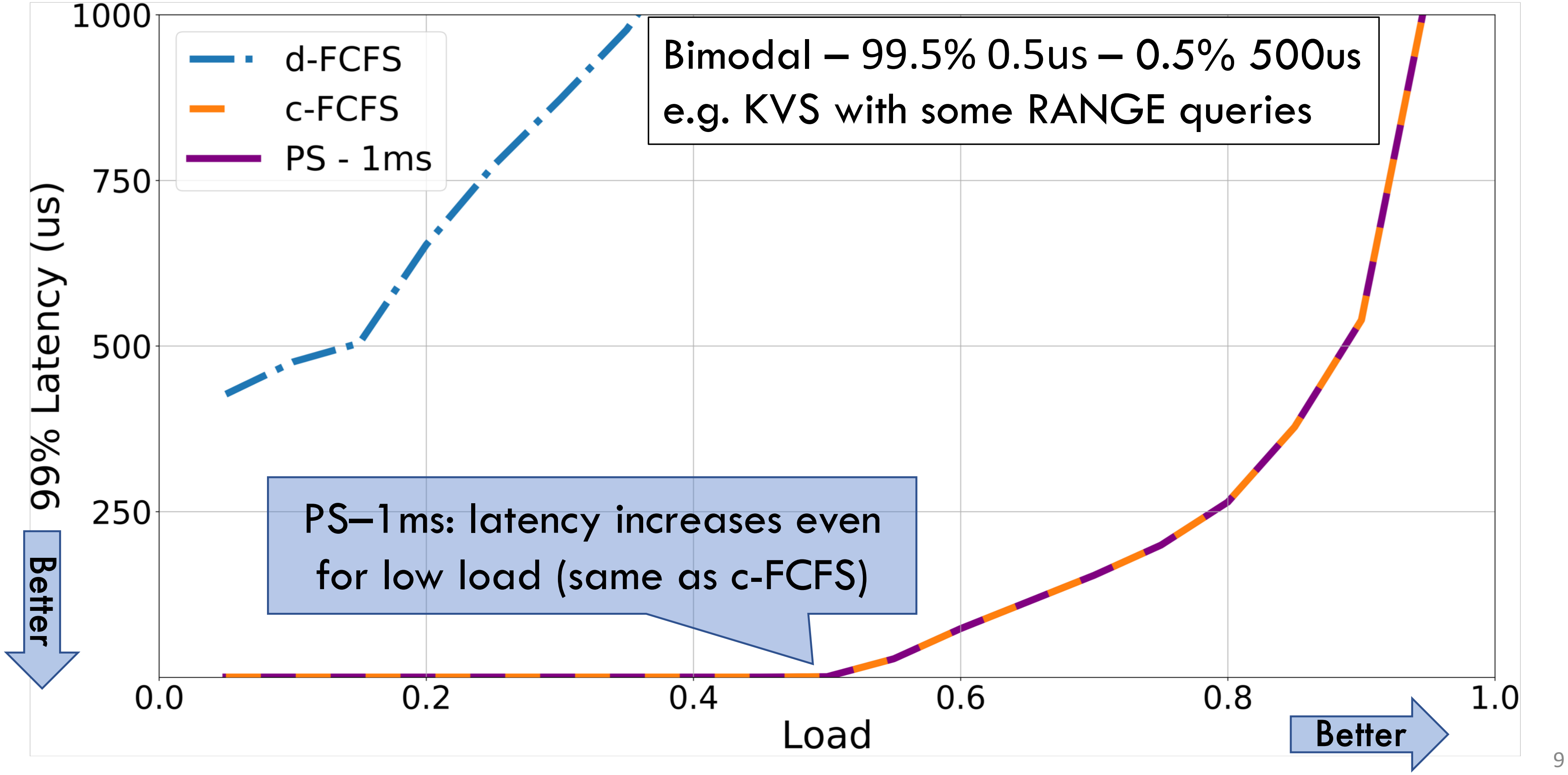
Problem: Short requests get stuck behind long ones



8

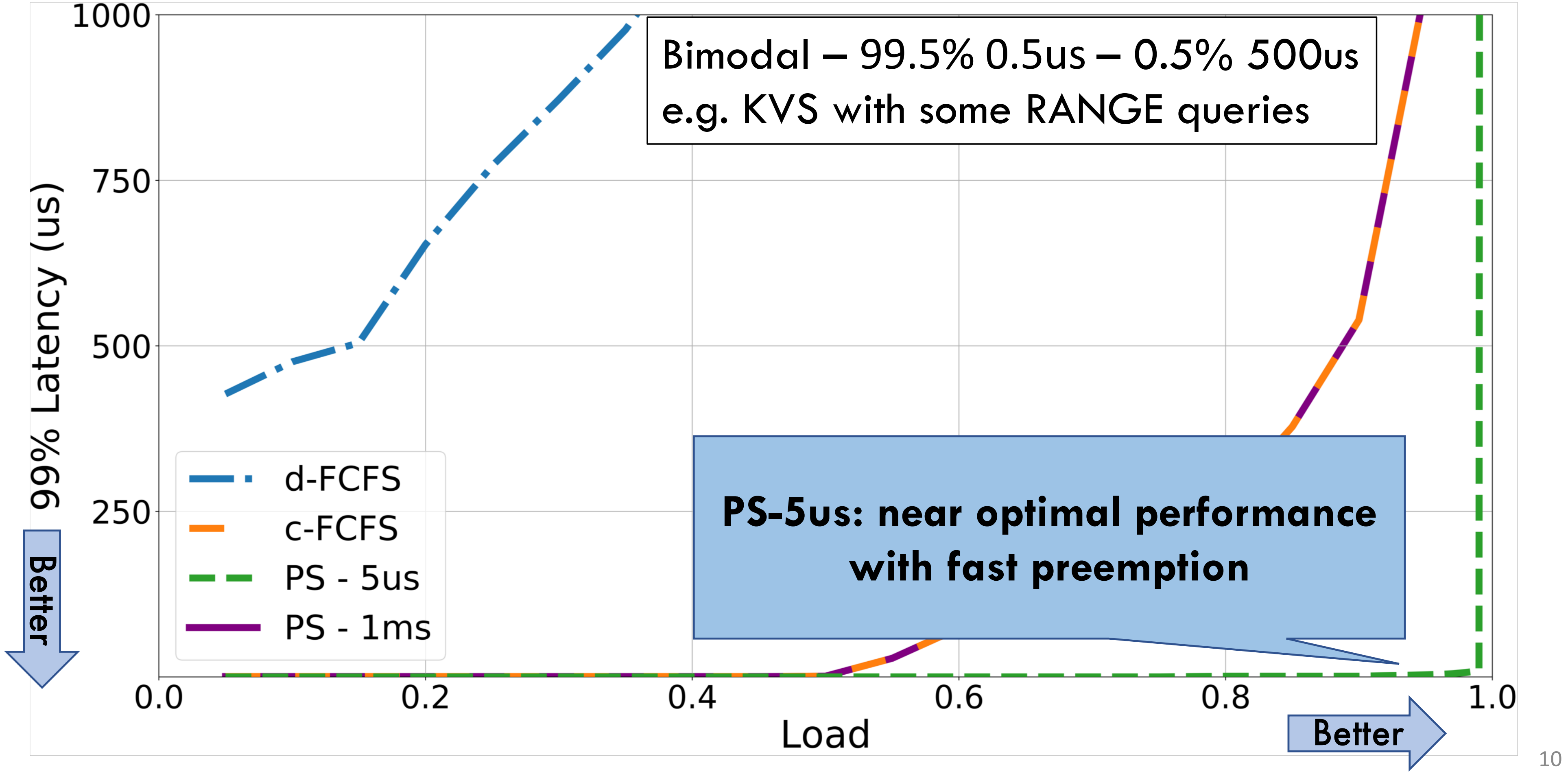
Acknowledgments: Kostis Kaffes

What if we could use the same preemptive scheduling as Linux?



Acknowledgments: Kostis Kaffes

Solution: What if we could use preemptive scheduling but at usec scale?



Acknowledgments: Kostis Kaffes

Solution: Shinjuku

A single address-space operating system that achieves microsecond-scale tail latency for all types of workloads regardless of variability in task duration

Key Features: **Preemption as often as 5us**

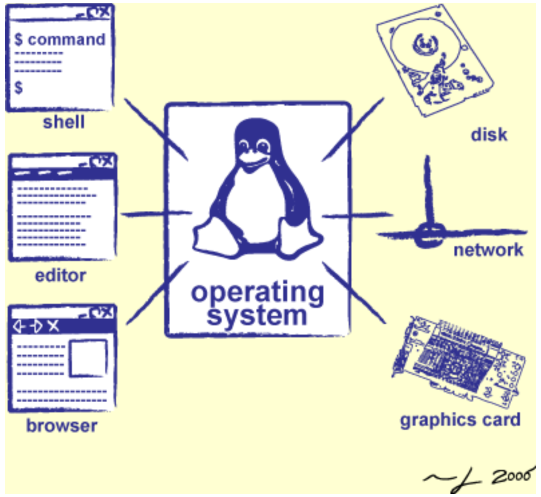
- Dedicated core for scheduling and queue management
- Leverage hardware support for virtualization for fast preemption
- Very fast context switching in user space
- Match scheduling policy to task distribution and target latency

12

Acknowledgments: Kostis Kaffes

Agenda: Synchronization

- How does an OS provide concurrency through threads?
  - Brief discussion of process/thread states and scheduling
  - High-level discussion of how stacks contribute to concurrency
- Introduce needs for synchronization
- Discussion of Locks and Semaphores



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**

Concurrency is Hard!

- Even for practicing engineers trying to write mission-critical, bulletproof code!
  - Threaded programs must work for all interleavings of thread instruction sequences
  - Cooperating threads inherently non-deterministic and non-reproducible
  - Really hard to debug unless carefully designed!
- Therac-25: Radiation Therapy Machine with Unintended Overdoses
  - Concurrency errors caused the death of a number of patients by misconfiguring the radiation production
  - Improper synchronization between input from operators and positioning software
- Mars Pathfinder Priority Inversion (JPL Account)
- Toyota Uncontrolled Acceleration (CMU Talk)
  - 256.6K Lines of C Code, ~9-11K global variables
  - Inconsistent mutual exclusion on reads/writes

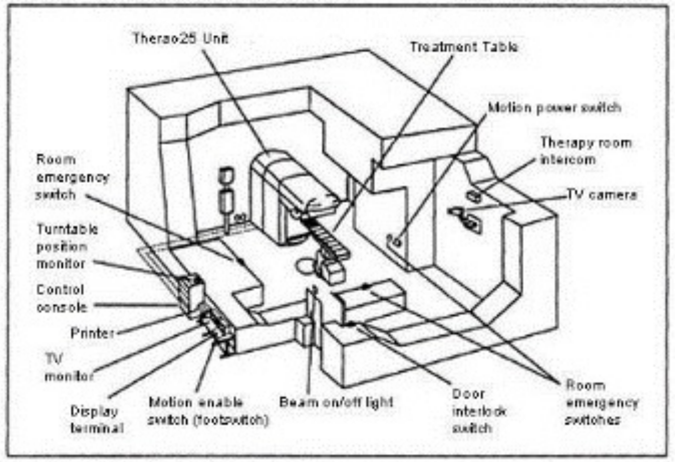
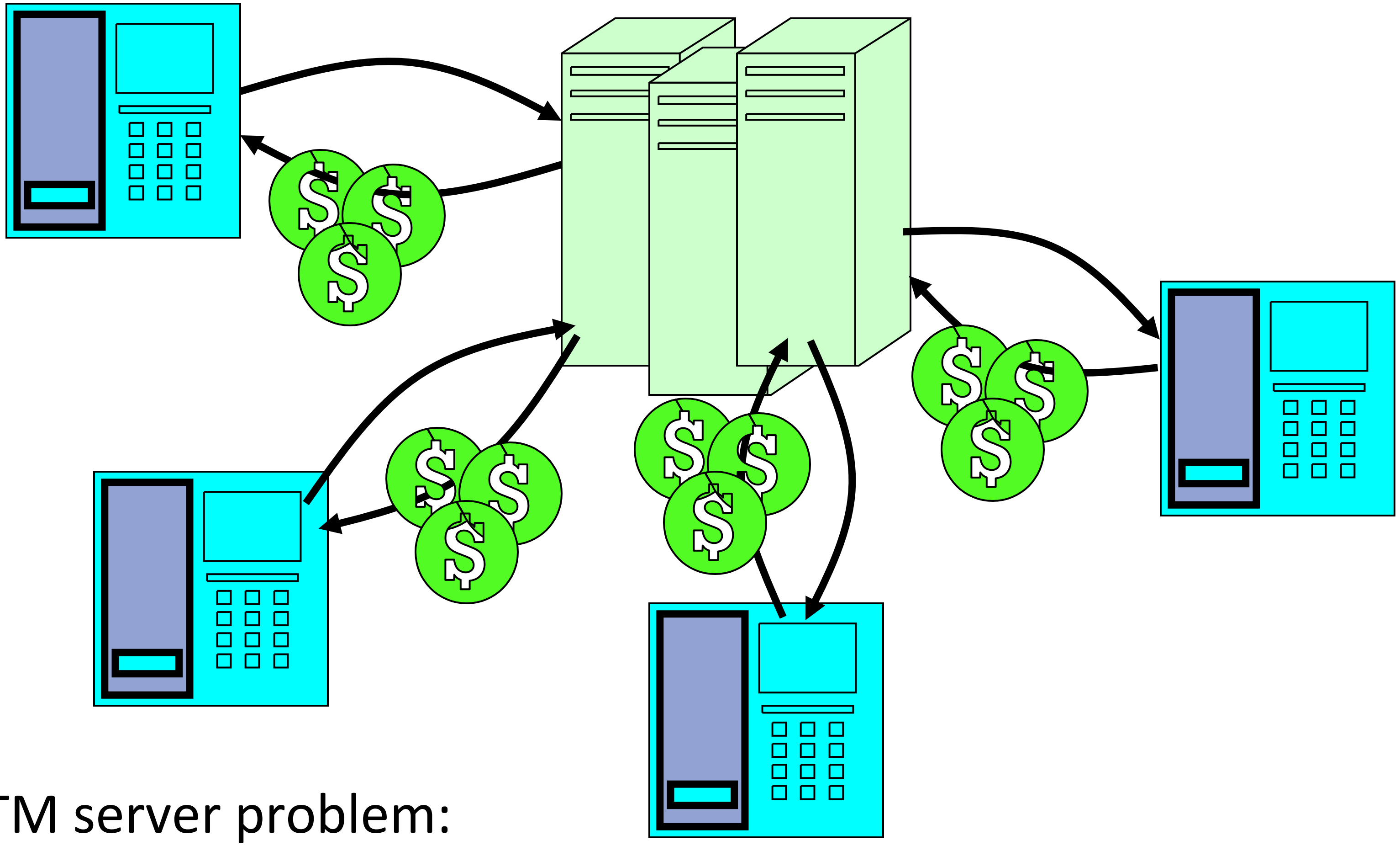


Figure 1. Typical Therac-25 facility

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-processing, or overlap computation 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();
   }
}
```

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

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`	

Atomic Operations

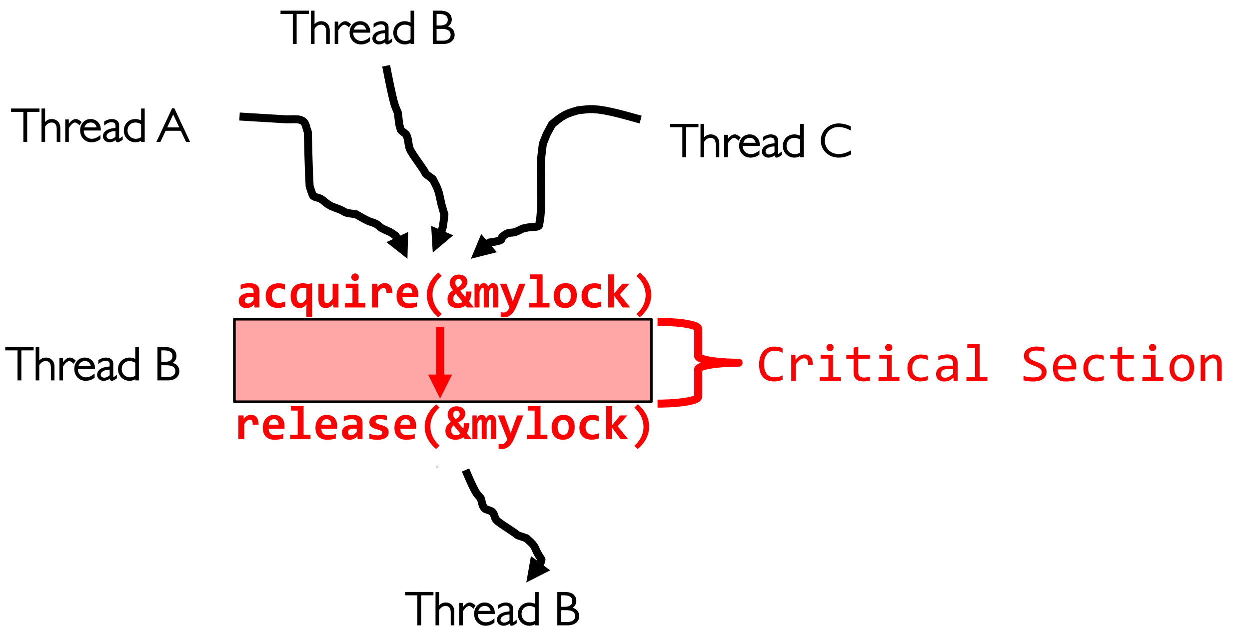
- To understand a concurrent program, we need to know what the underlying indivisible operations are!
- Atomic Operation: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - 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) are atomic
- 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

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
}
```

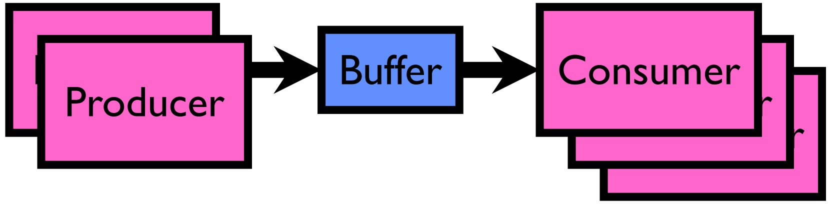
Critical Section



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!

Producer-Consumer with a Bounded Buffer

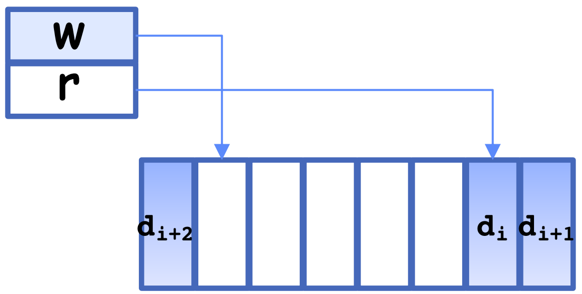


- Problem Definition
  - Producer(s) put things into a shared buffer
  - Consumer(s) take them out
  - Need synchronization to coordinate producer/consumer
- Don't want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty

- Example: Coke machine
  - Producer can put limited number of Cokes in machine
  - Consumer can't take Cokes out if machine is empty
- Others: Web servers, Routers, ….

Circular Buffer Data Structure (sequential case)

```
typedef struct buf {
  int write_index;
  int read_index;
  <type> *entries[BUFSIZE];
} buf_t;
```



- Insert: write & bump write ptr (enqueue)
- Remove: read & bump read ptr (dequeue)
- *How to tell if Full (on insert) Empty (on remove)?*
- *And what do you do if it is?*
- *What needs to be atomic?*

Circular Buffer – first cut

```
mutex buf_lock = <initially unlocked>

Producer(item) {
  acquire(&buf_lock);
  while (buffer full) {}; // Wait for a free slot
  enqueue(item);
  release(&buf_lock);
}


Consumer() {
  acquire(&buf_lock);
  while (buffer empty) {}; // Wait for arrival
  item = dequeue();
  release(&buf_lock);
  return item;
}
```

Will we ever come out of the wait loop?

Circular Buffer – 2nd cut

```
mutex buf_lock = <initially unlocked>

Producer(item) {
  acquire(&buf_lock);
  while (buffer full) {release(&buf_lock); acquire(&buf_lock);}
  enqueue(item);
  release(&buf_lock);
}

Consumer() {
  acquire(&buf_lock);
  while (buffer empty) {release(&buf_lock); acquire(&buf_lock);}
  item = dequeue();
  release(&buf_lock);
  return item;
}
```

What happens when one is waiting for the other?

Conclusion

- Concurrency accomplished by multiplexing CPU time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (yield(), I/O) or involuntary (interrupts)
- TCB + Stacks hold complete state of thread for restarting
- Atomic Operation: an operation that always runs to completion or not at all
- Synchronization: using atomic operations to ensure cooperation between threads
- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - One thread *excludes* the other while doing its task
- Critical Section: piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code
- Locks: synchronization mechanism for enforcing mutual exclusion on critical sections to construct atomic operations