Operating Systems (Honor Track)

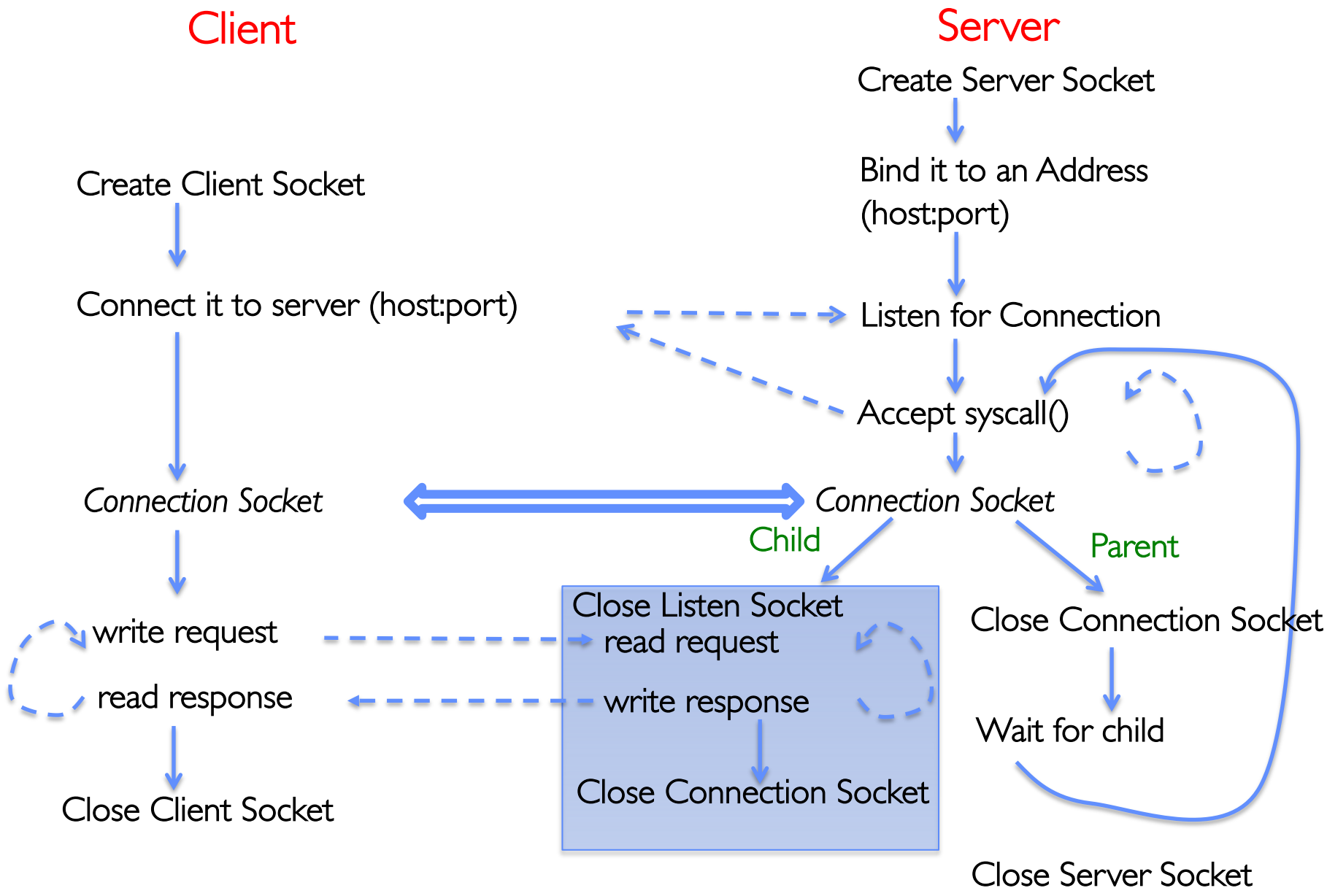
Synchronization 2: Concurrency (cont'd), Lock Implementation

Xin Jin

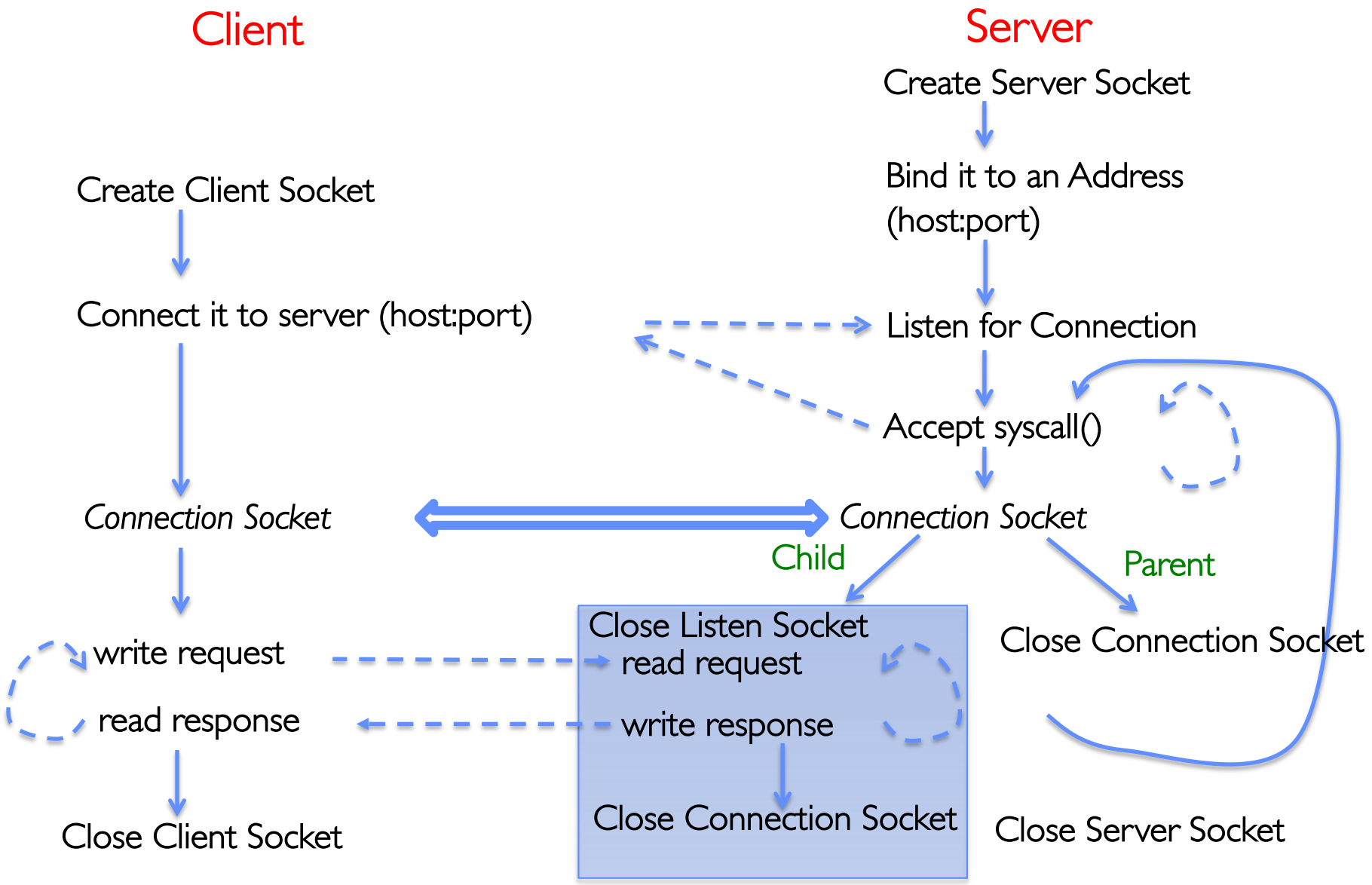
Spring 2022

Acknowledgments: Ion Stoica, Berkeley CS 162

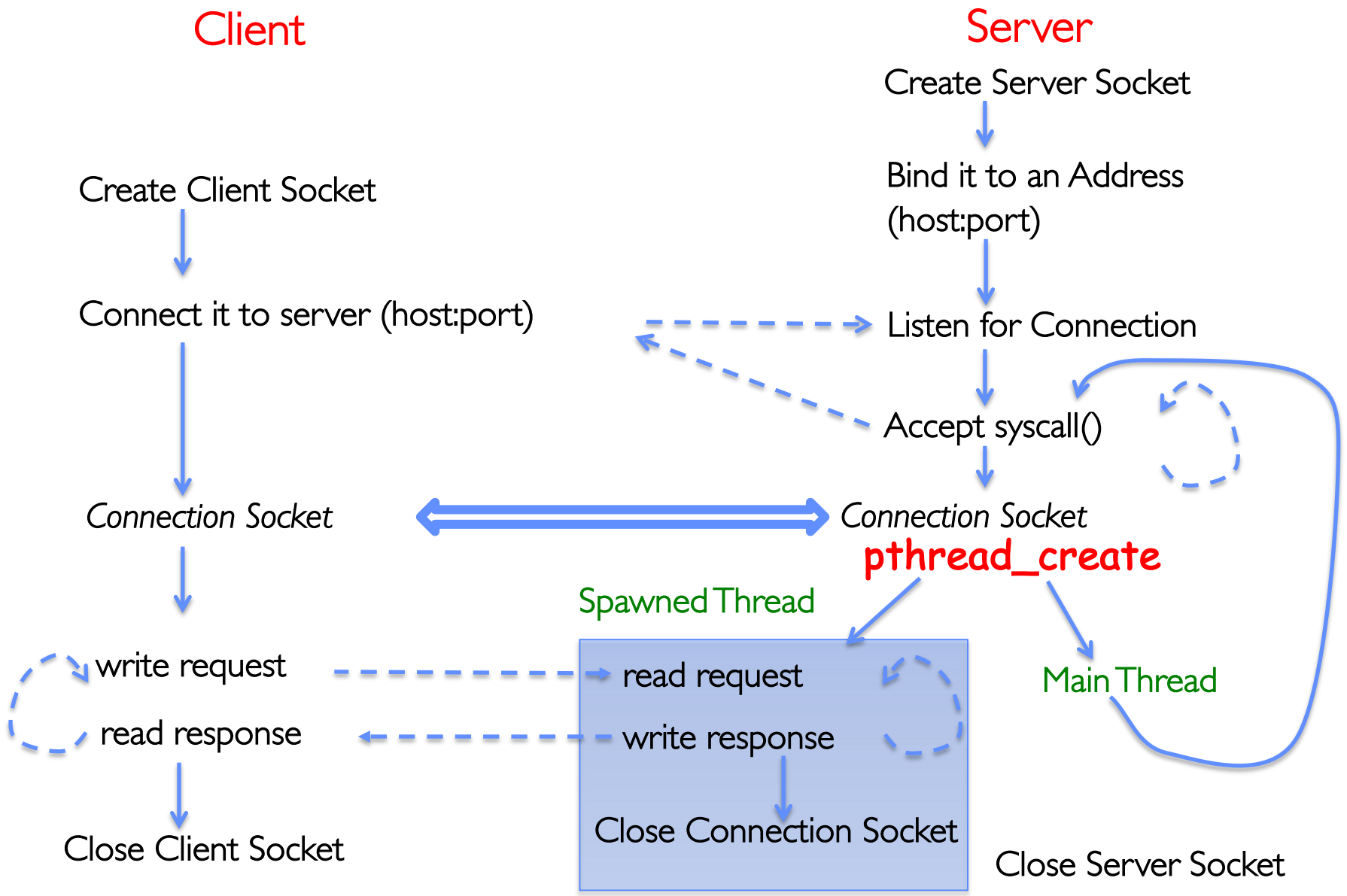
Recap: Sockets With Protection (each connection has own process)



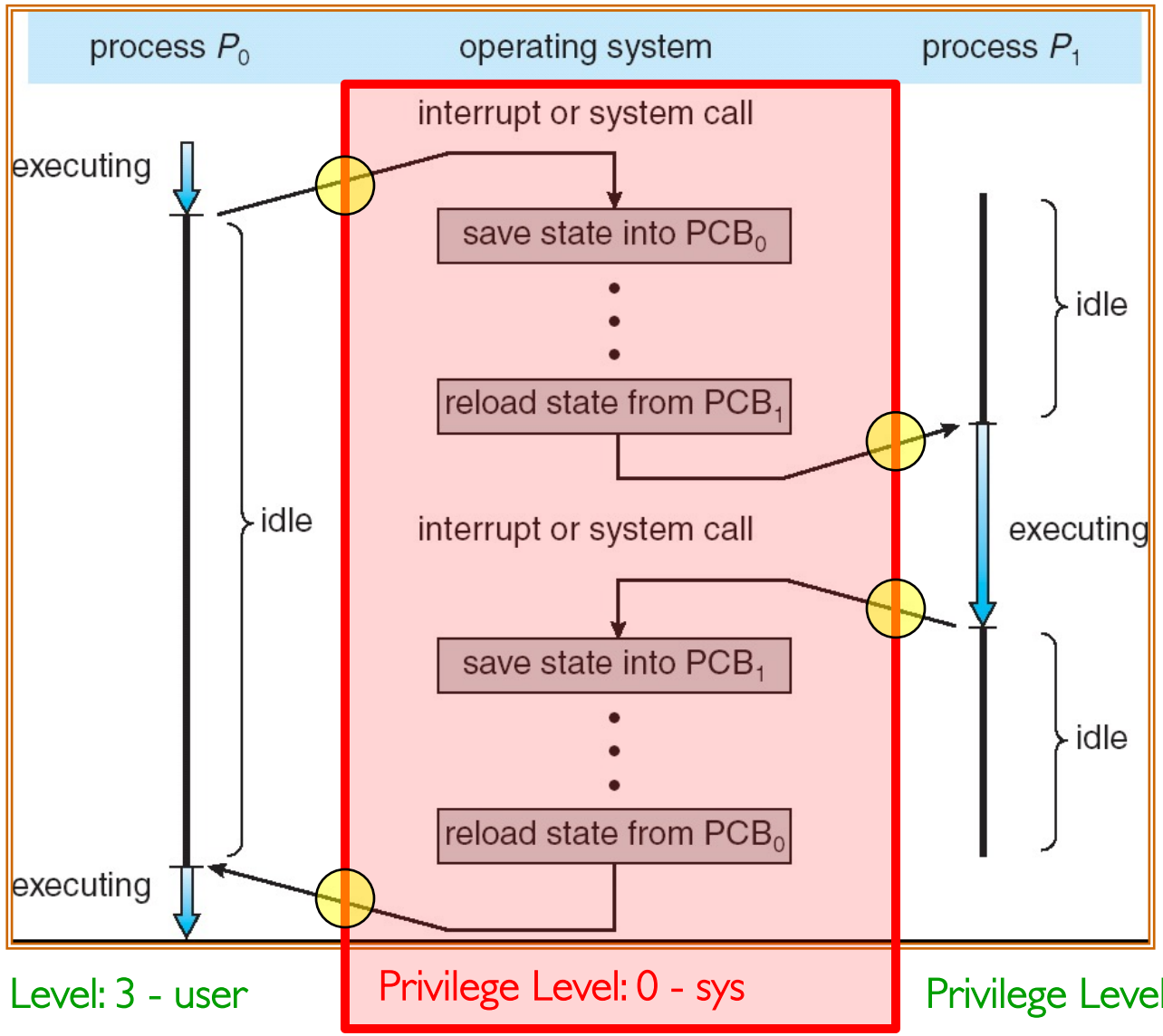
Recap: Sockets With Protection and Concurrency



Recap: Sockets with Concurrency, without Protection



Recap: Context Switch



Privilege Level: 3 - user

Privilege Level: 0 - sys

Privilege Level: 3 - user

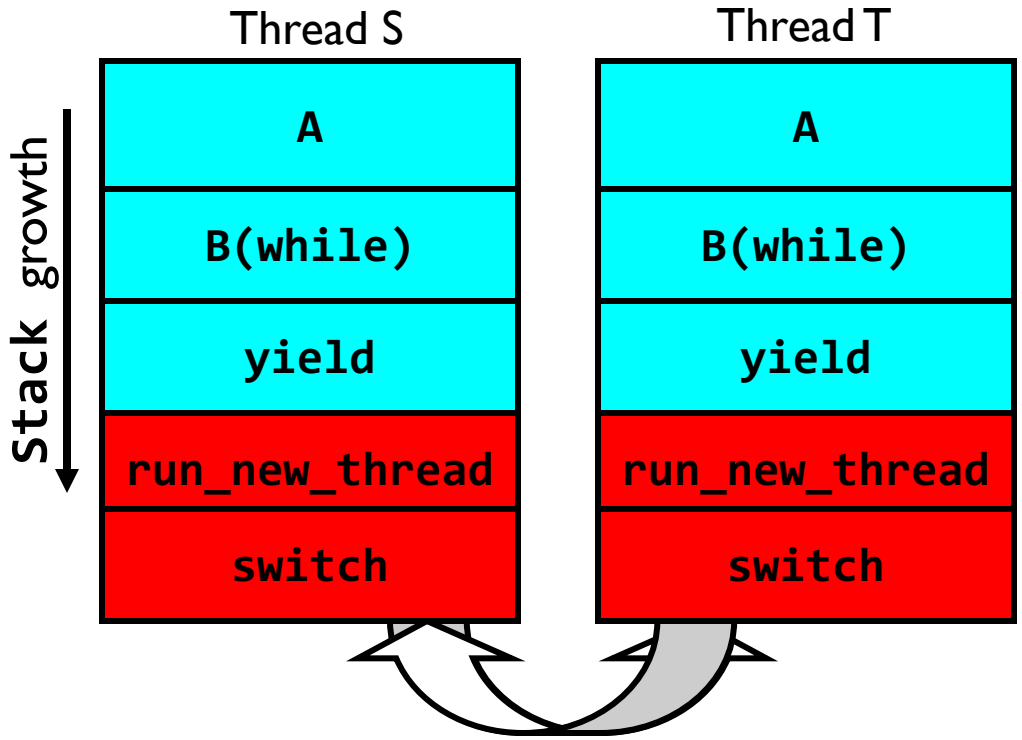
Recap: What Do the Stacks Look Like?

- 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?
  - 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

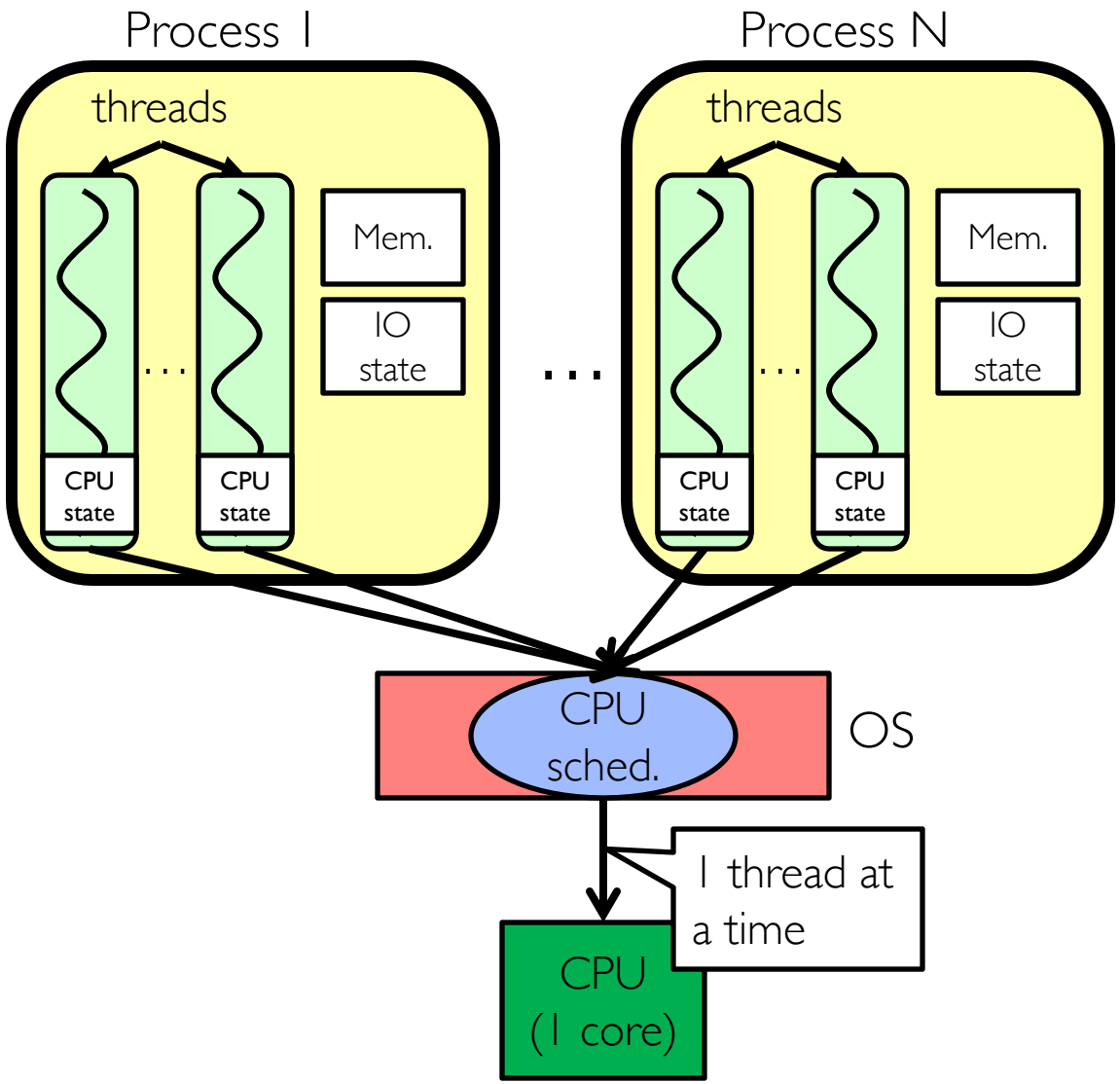
Aren't we still switching contexts?

- 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

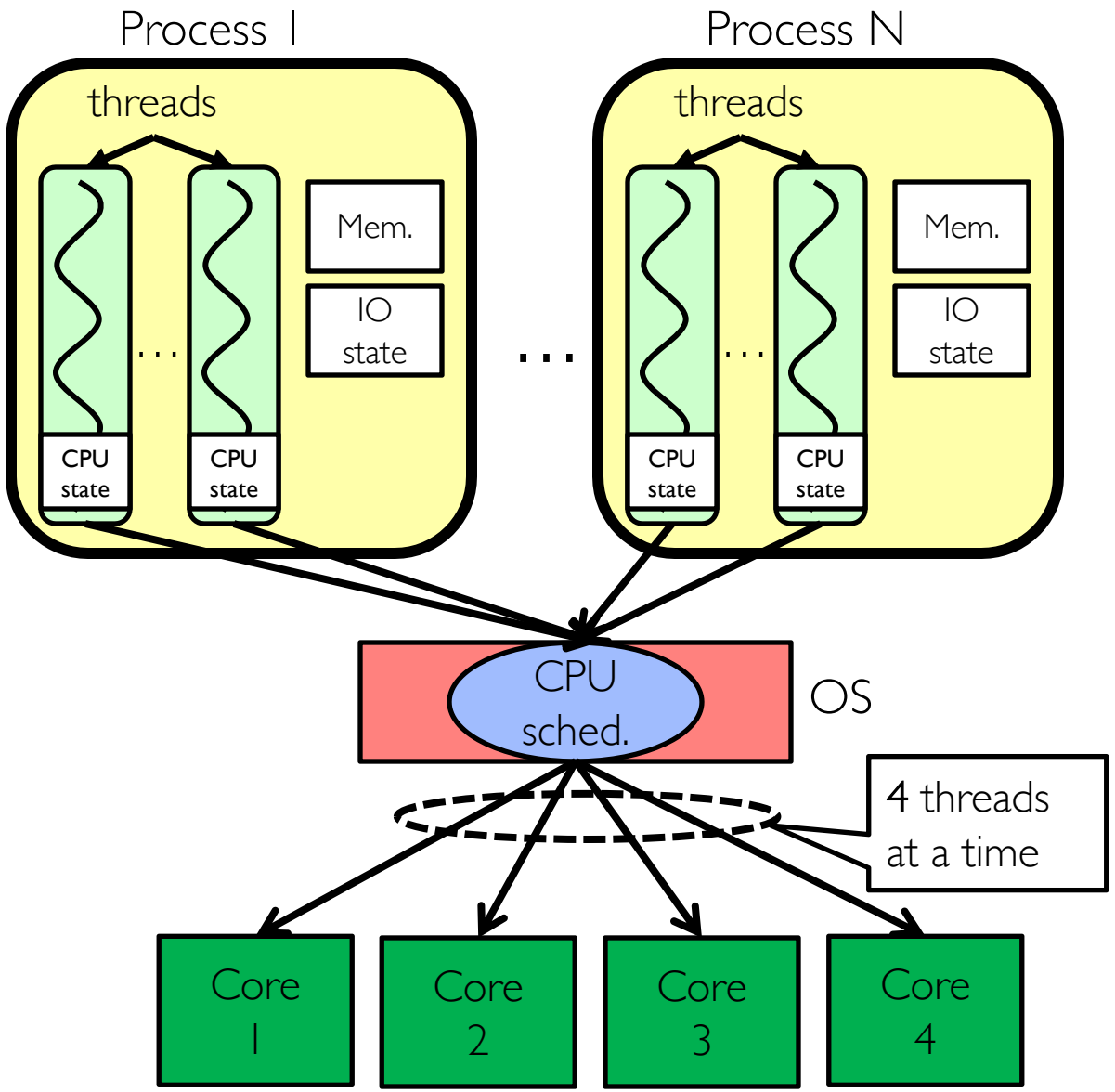
- Even cheaper: switch threads (using “yield”) in user-space!

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**

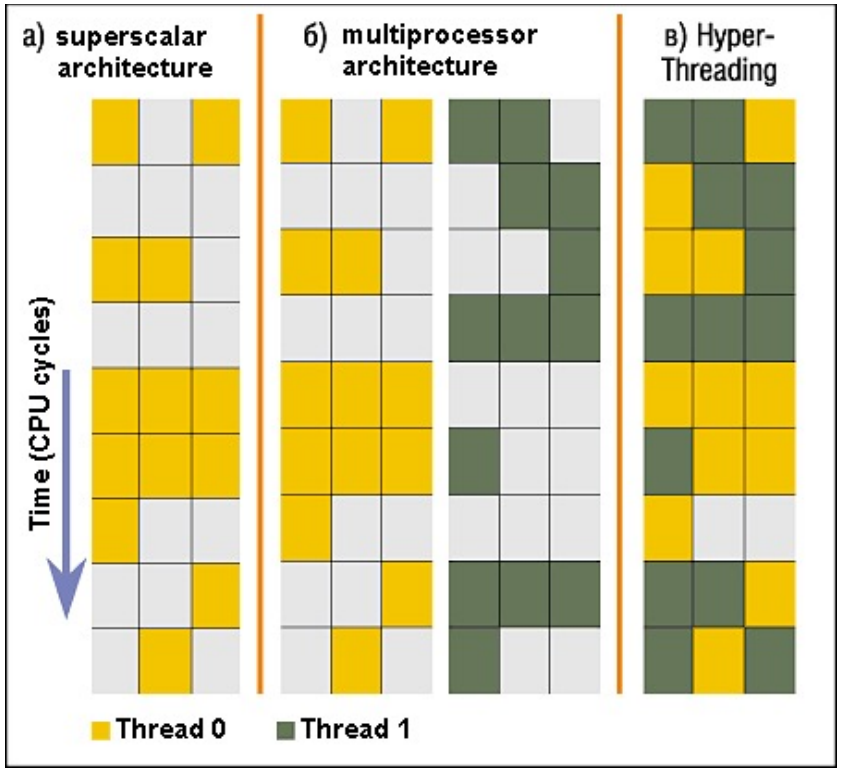
Processes vs. Threads



- 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

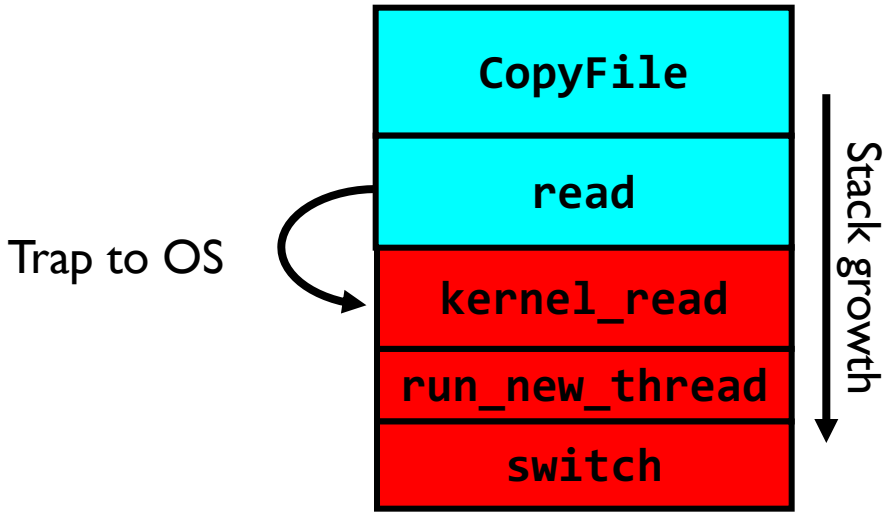
- Hardware scheduling technique
  - Superscalar processors can execute multiple instructions that are independent.
  - Hyperthreading duplicates register state to make a second "thread," allowing more instructions to run.
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!



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

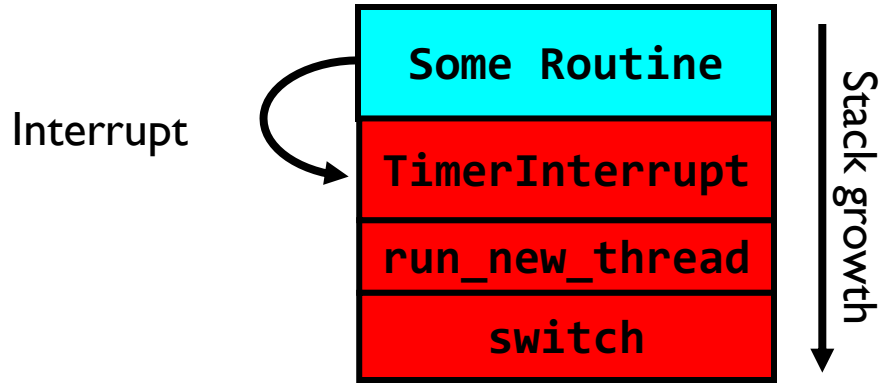
- What happens if thread never does any I/O, never waits, and never yields control?
  - Could the `ComputePI` program grab all resources and never release the processor?
    - » What if it didn't print to console?
  - Must find way that dispatcher can regain control!

- 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:

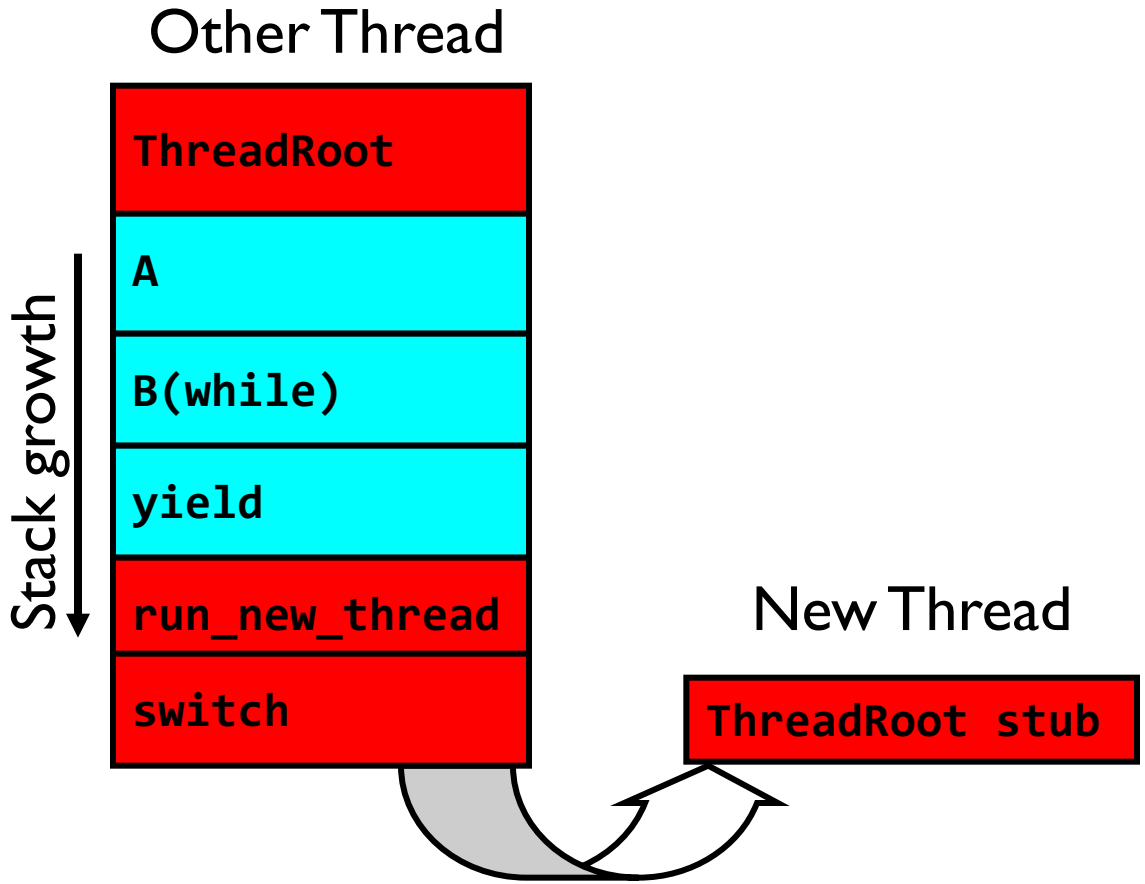
```
TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```

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

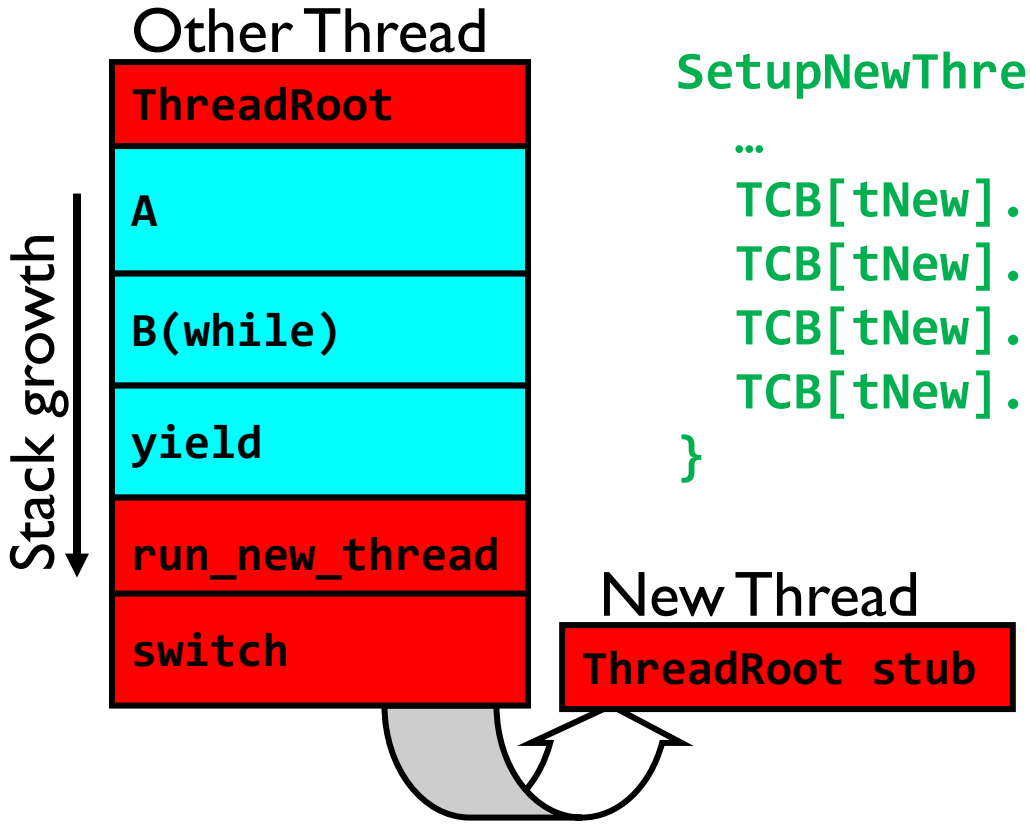
Initial Stack

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?



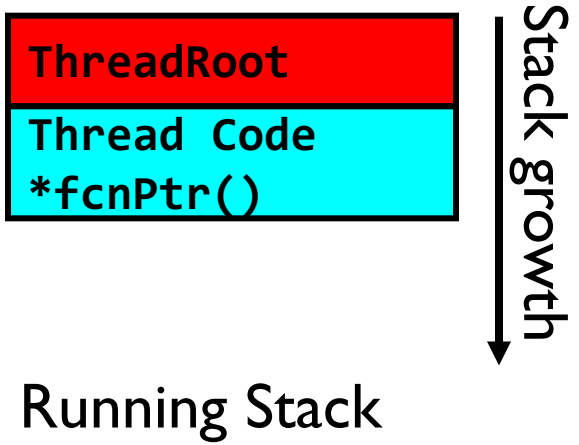

```
SetupNewThread(tNew) {
  …
  TCB[tNew].regs.sp = newStackPtr;
  TCB[tNew].regs.retpc = &ThreadRoot;
  TCB[tNew].regs.r0 = fcnPtr
  TCB[tNew].regs.r1 = fcnArgPtr
}
```

- 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

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 areas
- 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, …

Shinjuku: Preemptive Scheduling for Microsecond-Scale Tail Latency

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

NSDI'19

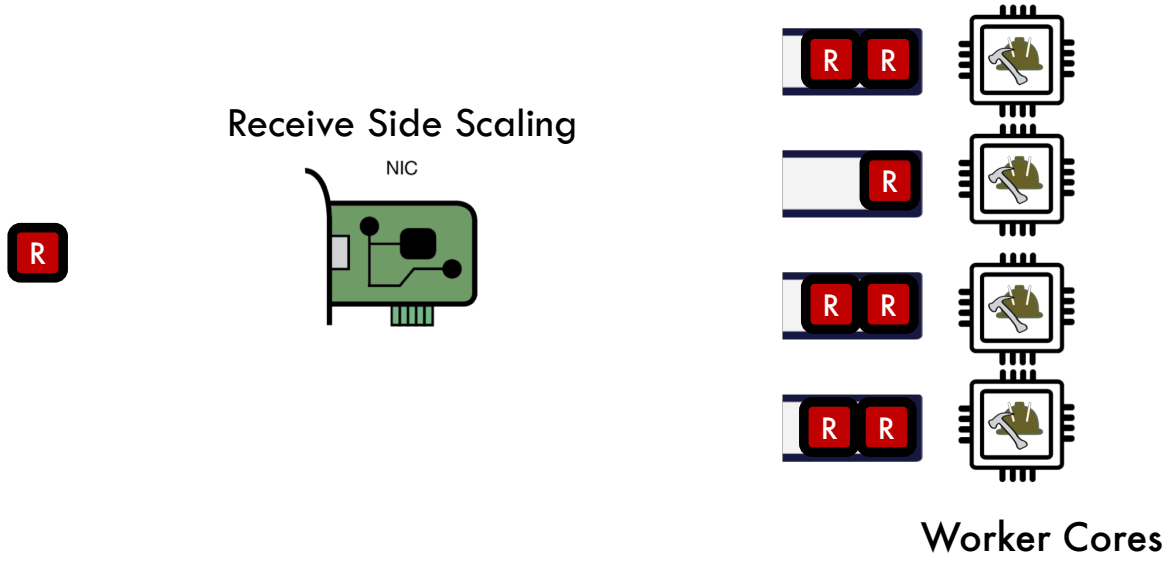
Achieving low tail latency at microsecond scale is hard

Problem: High OS overheads

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

Distributed Queues + First Come First Serve scheduling

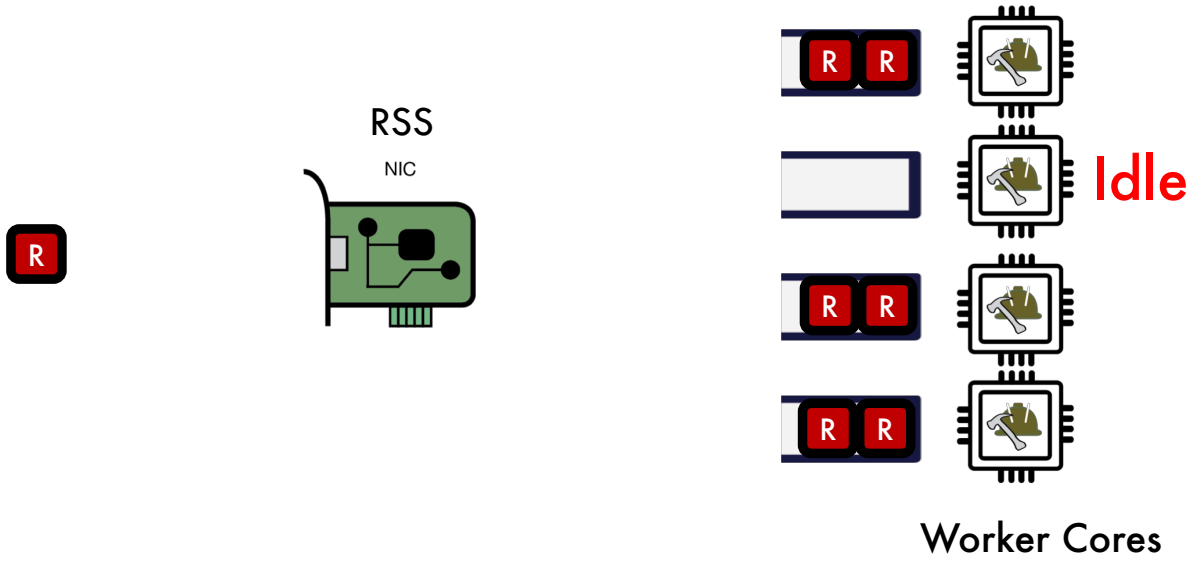
d-FCFS (DPDK, IX, Arrakis)



Acknowledgments: Kostis Kaffes

Achieving low tail latency at microsecond scale is hard

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

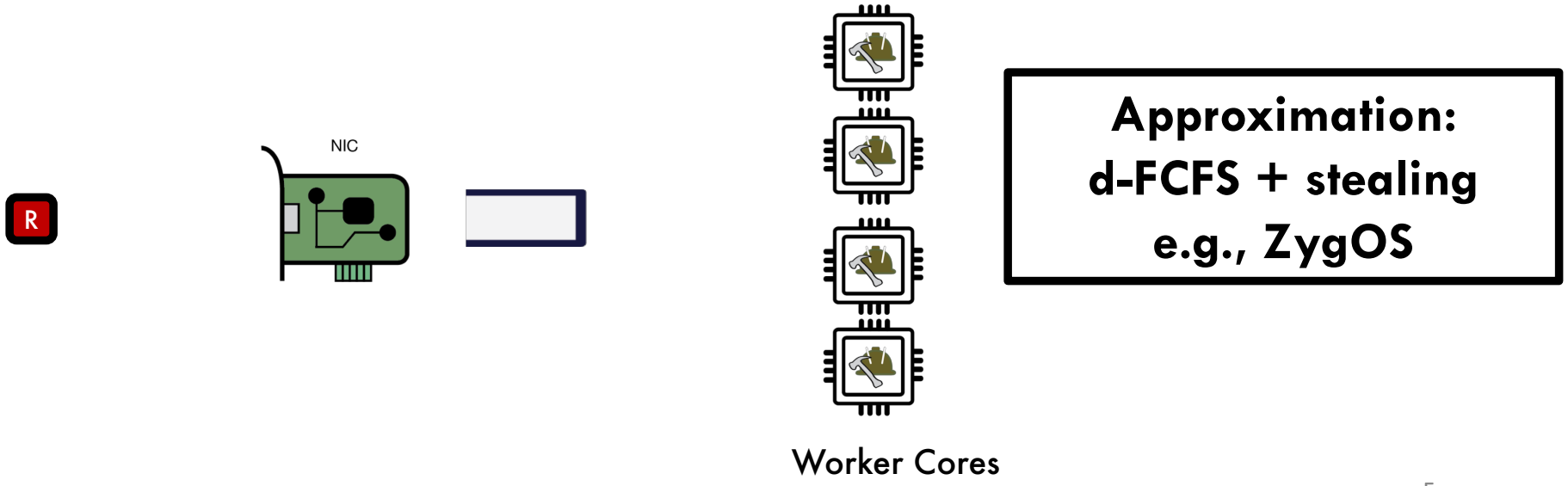


Acknowledgments: Kostis Kaffes

Achieving low tail latency at microsecond scale is hard

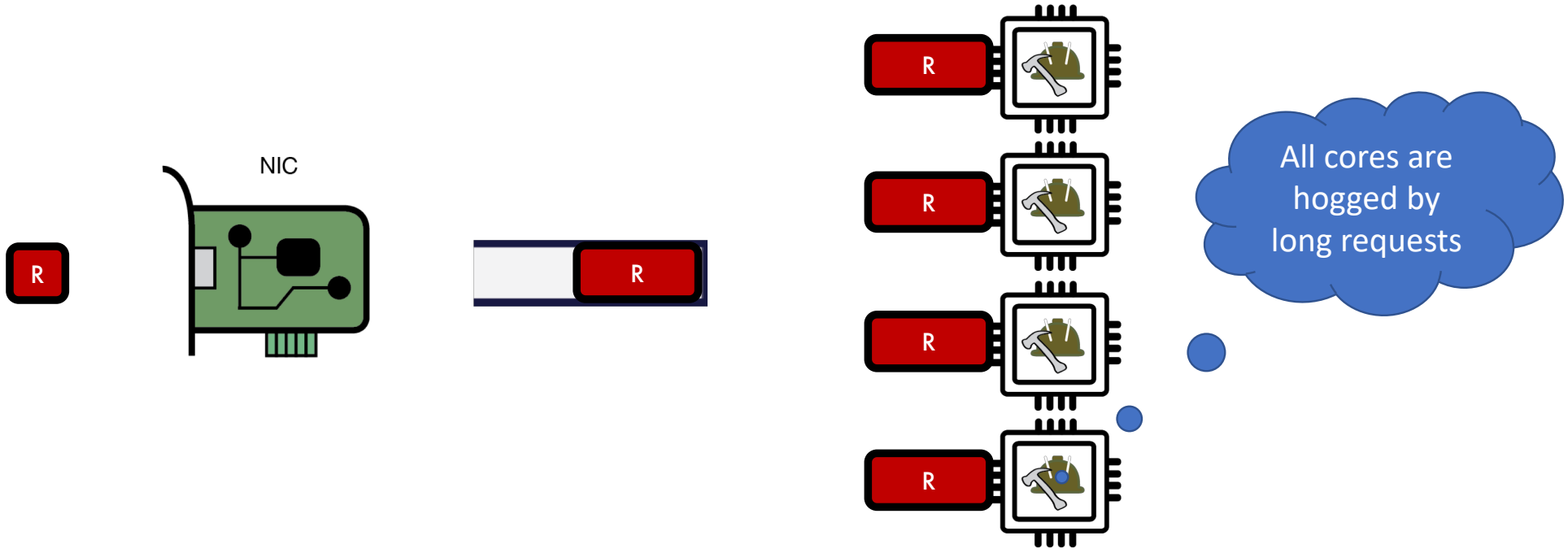
Problem: Queue imbalance because d-FCFS is not work conserving

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



Acknowledgments: Kostis Kaffes

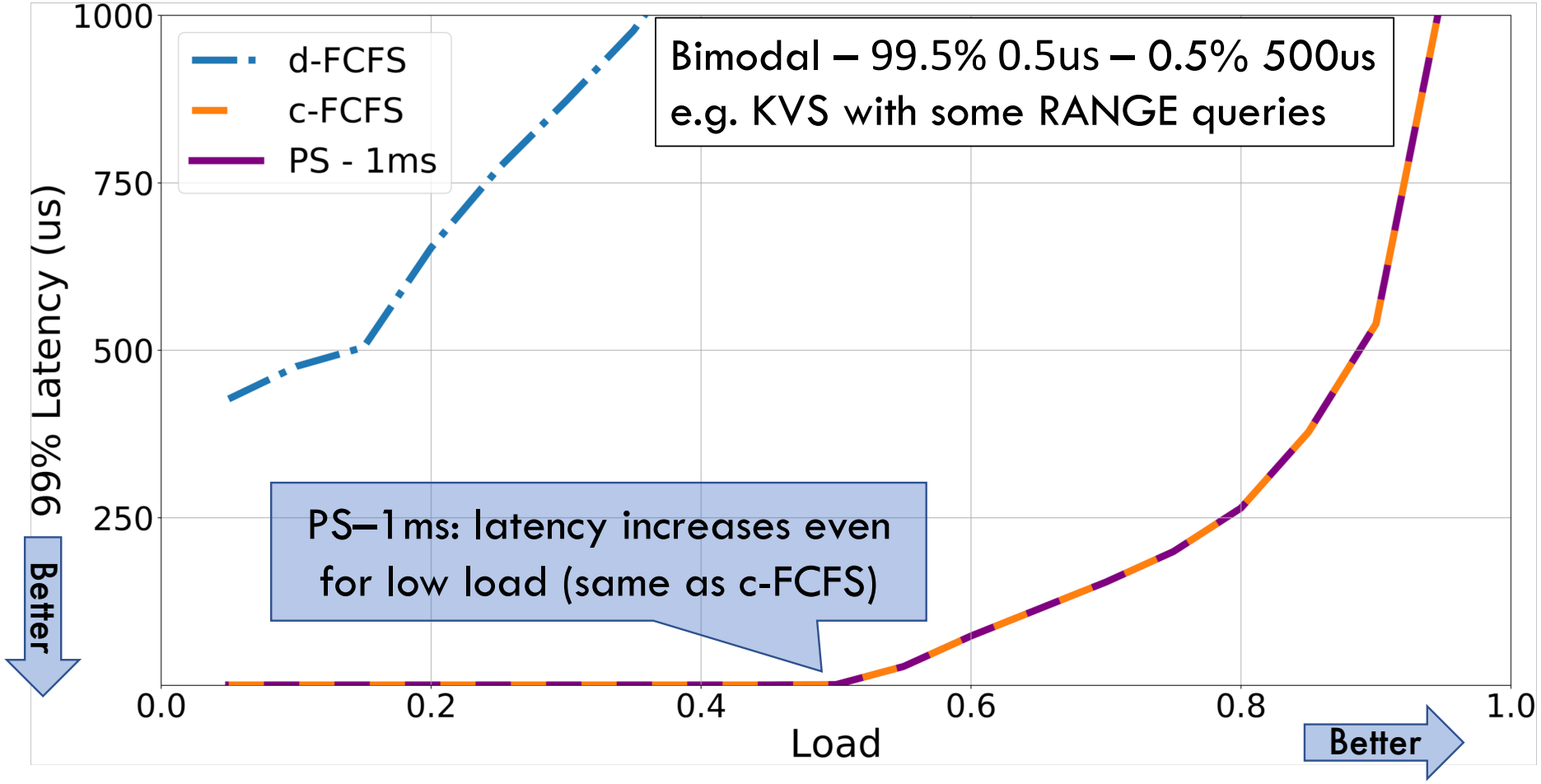
Problem: Short requests get stuck behind long ones



8

Acknowledgments: Kostis Kaffes

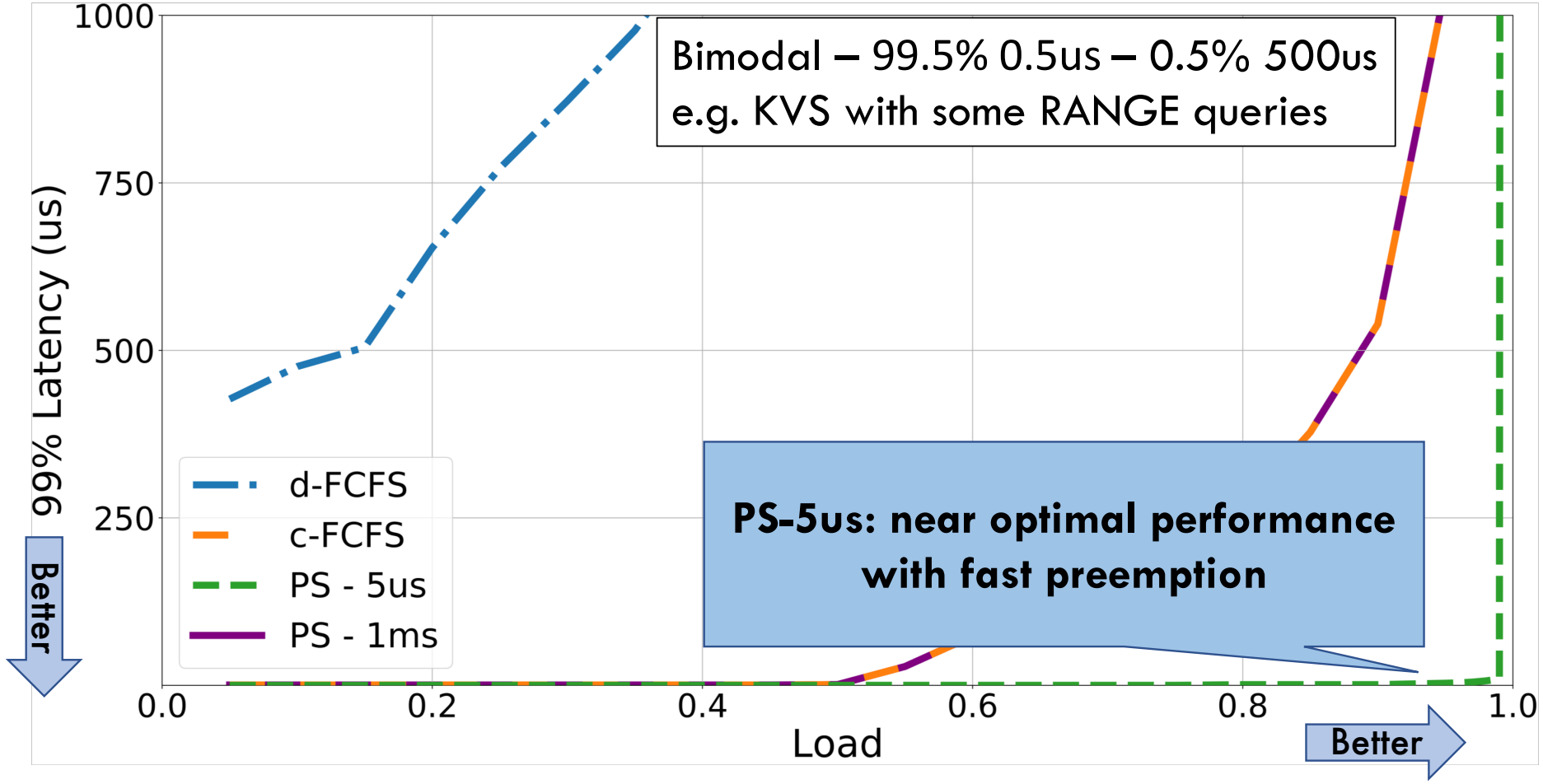
What if we could use the same preemptive scheduling as Linux?



9

Acknowledgments: Kostis Kaffes

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



10

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

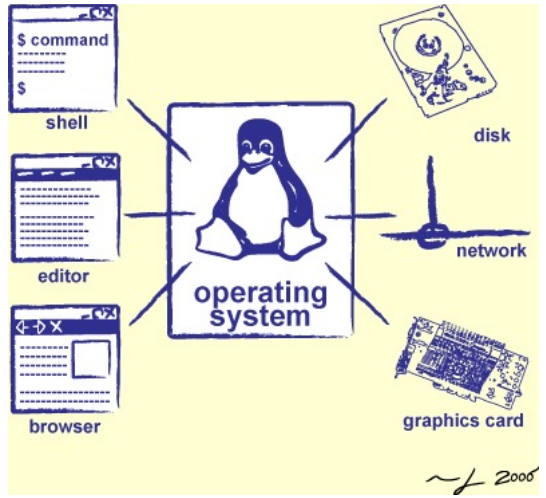
Key Features: **Preemption as often as 5us**

- 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

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

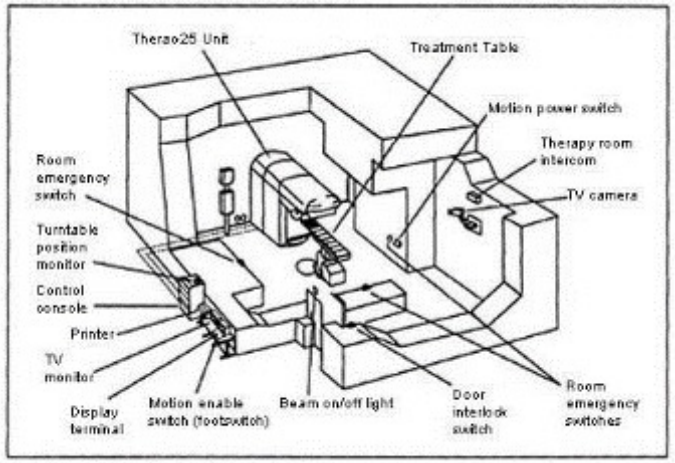
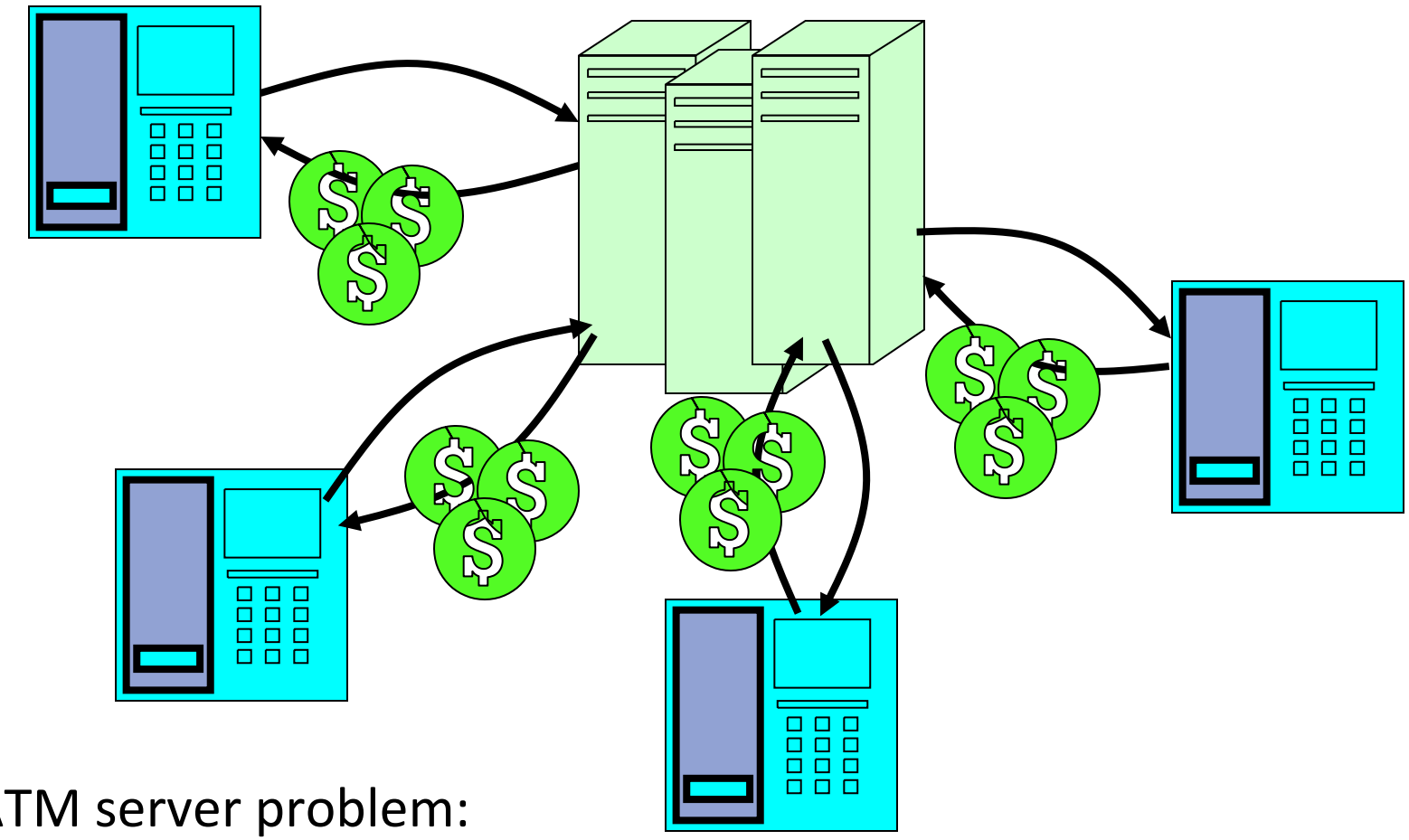


Figure 1. Typical Therac-25 facility

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

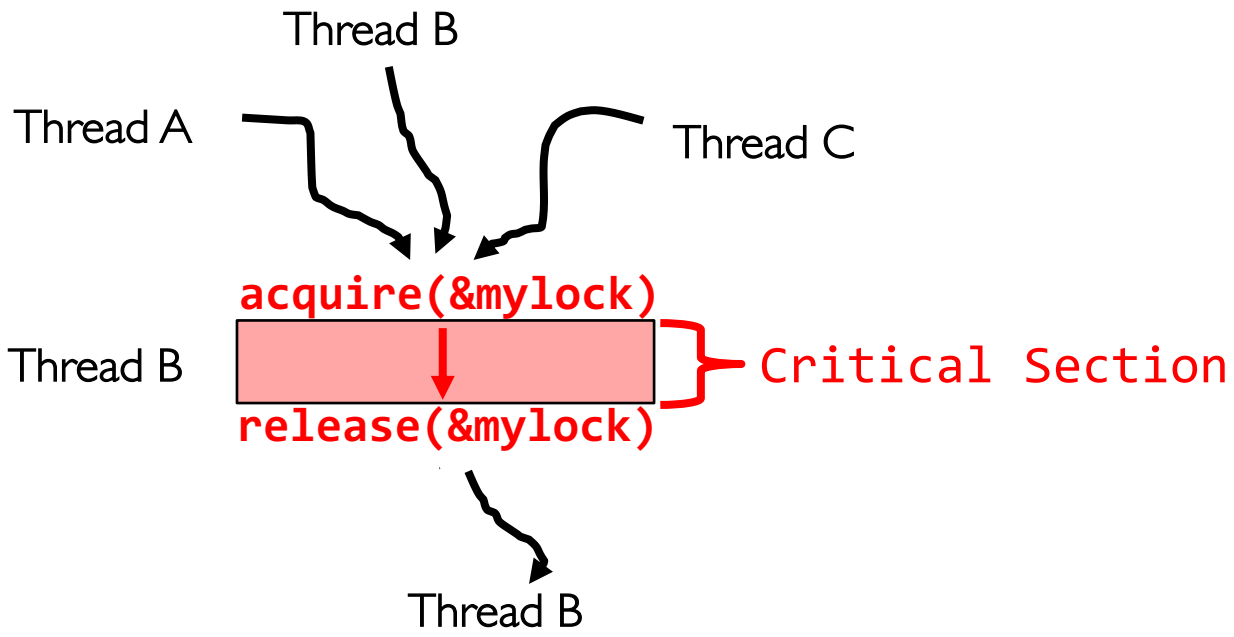
- To understand a concurrent program, we need to know what the underlying indivisible operations are!
- Atomic Operation: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle
 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

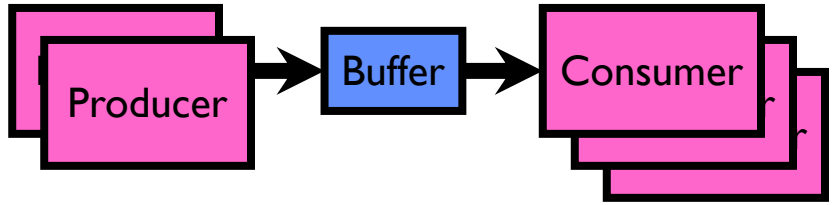
Critical Section



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



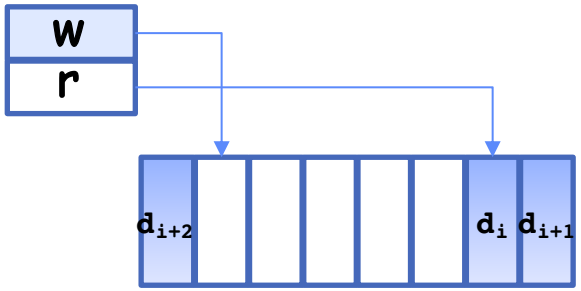
- Problem Definition
  - Producer(s) put things into a shared buffer
  - Consumer(s) take them out
  - Need synchronization to coordinate producer/consumer

- 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?*

Group Discussion

```
Producer(item) {
  enqueue(item);
}

Consumer() {
  item = dequeue();
  return item;
}
```

- Topic: Circular Buffer
  - How to implement it with locks?
  - How to implement it with semaphores?
  - What are the pros and cons of each solution?

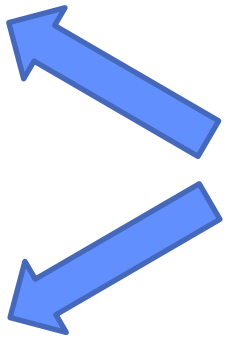
- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

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;
}
```



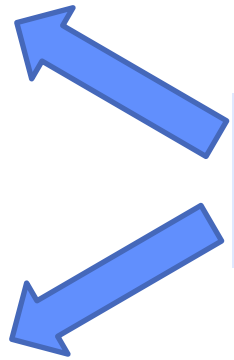
Will we ever come out of the wait loop?

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?

Recall: Semaphores

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - `Down()` or `P()`: an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - » Think of this as the wait() operation
  - `Up()` or `V()`: an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - » This of this as the signal() operation
  - Note that `P()` stands for "*proberen*" (to test) and `V()` stands for "*verhogen*" (to increment) in Dutch

Revisit Bounded Buffer: Correctness constraints for solution

- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine
- General rule of thumb: Use a separate semaphore for each constraint
  - `Semaphore fullBuffers; // consumer's constraint`
  - `Semaphore emptyBuffers;// producer's constraint`
  - `Semaphore mutex;       // mutual exclusion`

Full Solution to Bounded Buffer (coke machine)


```
Semaphore fullSlots = 0;      // Initially, no coke
Semaphore emptySlots = bufSize;
                              // Initially, num empty slots
Semaphore mutex = 1;          // No one using machine

Producer(item) {
    semaP(&emptySlots);       // Wait until space
    semaP(&mutex);            // Wait until machine free
    Enqueue(item);
    semaV(&mutex);
    semaV(&fullSlots);        // Tell consumers there is
                              // more coke
}
Consumer() {
    semaP(&fullSlots);        // Check if there's a coke
    semaP(&mutex);            // Wait until machine free
    item = Dequeue();
    semaV(&mutex);
    semaV(&emptySlots);       // tell producer need more
    return item;
}
```

fullSlots signals coke

emptySlots signals space

Critical sections using mutex protect integrity of the queue

Discussion about Solution

- Why asymmetry?
  - Producer does: **semaP(&emptyBuffer), semaV(&fullBuffer)**
  - Consumer does: **semaP(&fullBuffer), semaV(&emptyBuffer)**

Decrease # of empty slots

Increase # of occupied slots

Decrease # of occupied slots

Increase # of empty slots

- Is order of P's important?
- Is order of V's important?
- What if we have 2 producers or 2 consumers?

```
Producer(item) {
   semaP(&mutex);
   semaP(&emptySlots);
   Enqueue(item);
   semaV(&mutex);
   semaV(&fullSlots);
}
Consumer() {
   semaP(&fullSlots);
   semaP(&mutex);
   item = Dequeue();
   semaV(&mutex);
   semaV(&emptySlots);
   return item;
}
```

Where are we going with synchronization?

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Compare&Swap

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

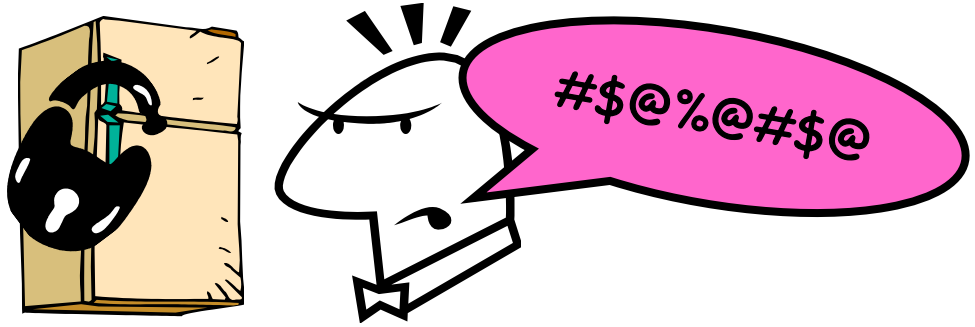
Motivating Example: "Too Much Milk"

- Great thing about OS's – analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people
- Example: People need to coordinate:

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Recall: What is a lock?

- 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
- For example: fix the milk problem by putting a key on the refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: roommate angry if only wants orange juice



- Of Course – We don't know how to make a lock yet
  - Let's see if we can answer this question!

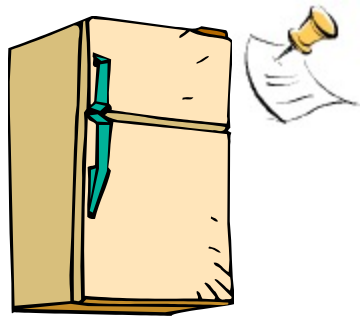
Too Much Milk: Correctness Properties

- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Impulse is to start coding first, then when it doesn't work, pull hair out
  - Instead, think first, then code
  - Always write down behavior first
- What are the correctness properties for the "Too much milk" problem???
  - Never more than one person buys
  - Someone buys if needed
- First attempt: Restrict ourselves to use only atomic load and store operations as building blocks

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
      remove note;
   }
}
```



Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

Thread A

```
if (noMilk) {


   if (noNote) {
     leave Note;
     buy Milk;
     remove Note;
   }
}
```

Thread B

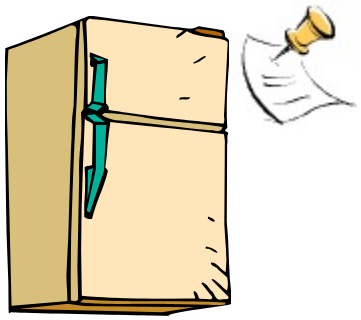
```
if (noMilk) {
   if (noNote) {


                leave Note;
                buy Milk;
                remove Note;
   }
}
```

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
      remove note;
   }
}
```



- Result?
  - Still too much milk but only occasionally!
  - Thread can get context switched after checking milk and note but before buying milk!
- Solution makes problem worse since fails intermittently
  - Makes it really hard to debug…
  - Must work despite what the dispatcher does!

Too Much Milk: Solution #1½

- Clearly the Note is not quite blocking enough
  - Let's try to fix this by placing note first
- Another try at previous solution:

```
leave Note;
if (noMilk) {
   if (noNote) {
      buy milk;
   }
}
remove Note;
```

- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

Too Much Milk Solution #2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:

Thread A

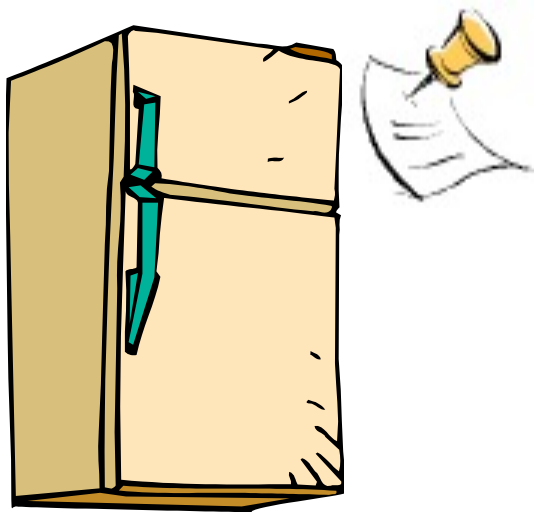
```
leave note A;
if (noNote B) {
   if (noMilk) {
      buy Milk;
   }
}
remove note A;
```

Thread B

```
leave note B;
if (noNoteA) {
   if (noMilk) {
      buy Milk;
   }
}
remove note B;
```

- Does this work?
- Possible for neither thread to buy milk
  - Context switches at exactly the wrong times can lead each to think that the other is going to buy
- Really insidious:
  - Extremely unlikely this would happen, but will at worse possible time
  - Probably something like this in UNIX

Too Much Milk Solution #2: problem!



- *I'm* not getting milk, *You're* getting milk
- This kind of lockup is called "starvation!"

Too Much Milk Solution #3

- Here is a possible two-note solution:

Thread A

```
leave note A;
while (note B) {\\X
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {\\Y
   if (noMilk) {
      buy milk;
   }
}
remove note B;
```

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At X:
  - If no note B, safe for A to buy,
  - Otherwise wait to find out what will happen
- At Y:
  - If no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

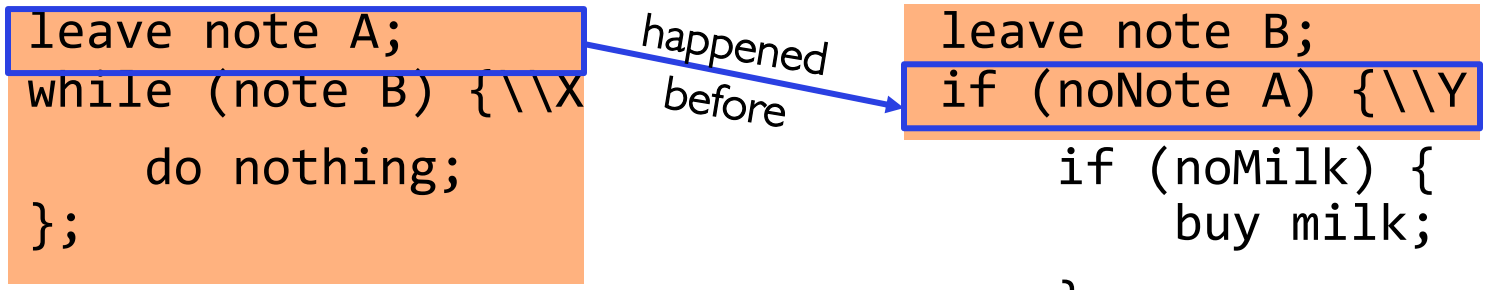
Case 1

- "`leave note A`" happens before "`if (noNote A)`"


```
if (noMilk) {
    buy milk;}
}
remove note A;
```

Case 1

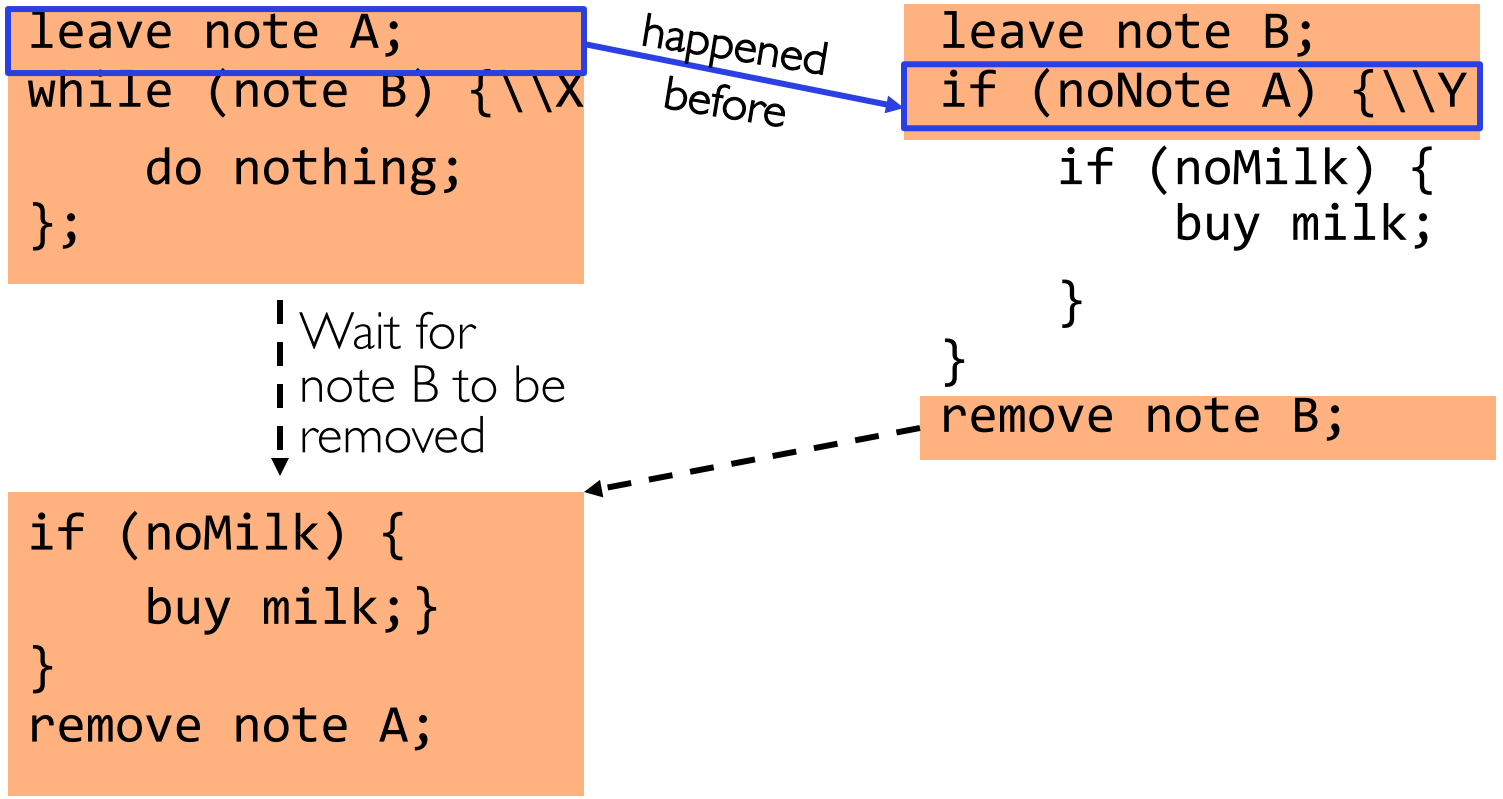
- "`leave note A`" happens before "`if (noNote A)`"




```
if (noMilk) {
    buy milk;}
}
remove note A;
```

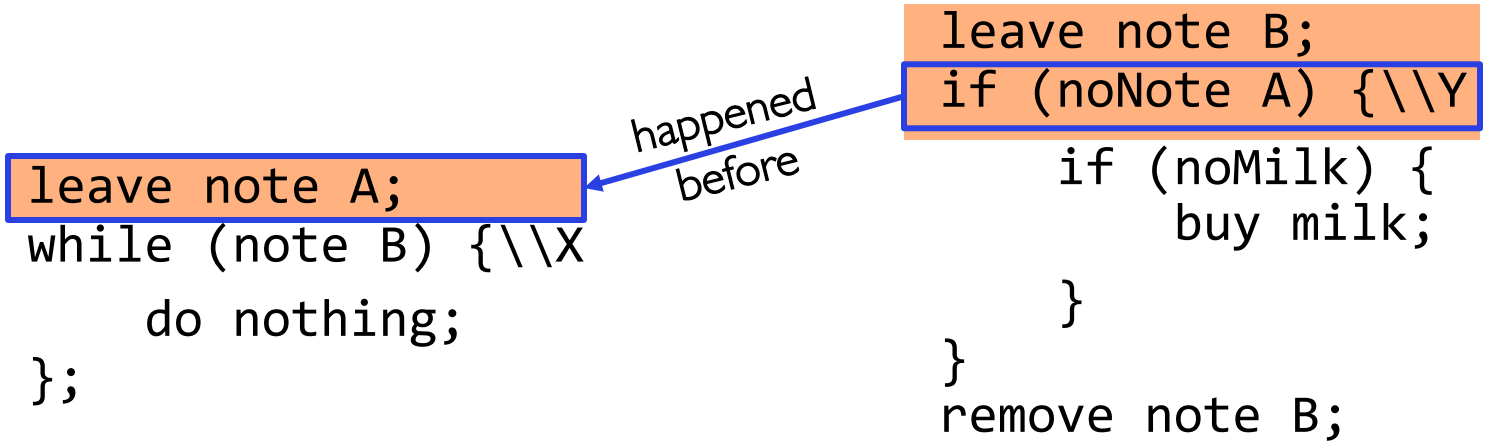
Case 1

- "`leave note A`" happens before "`if (noNote A)`"



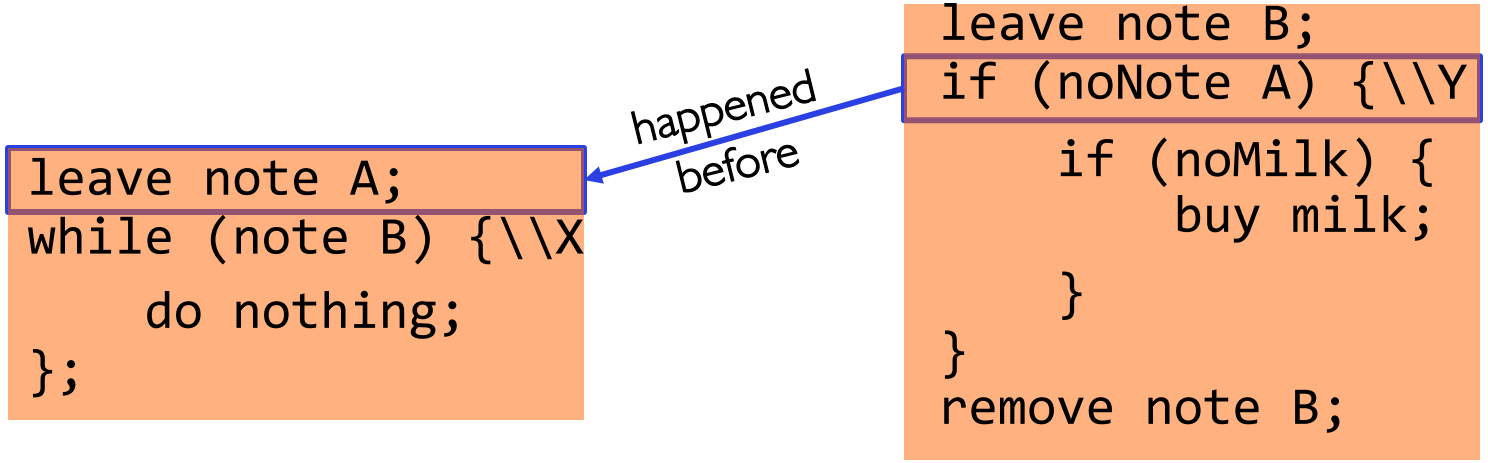
Case 2

- "`if (noNote A)`" happens before "`leave note A`"



Case 2

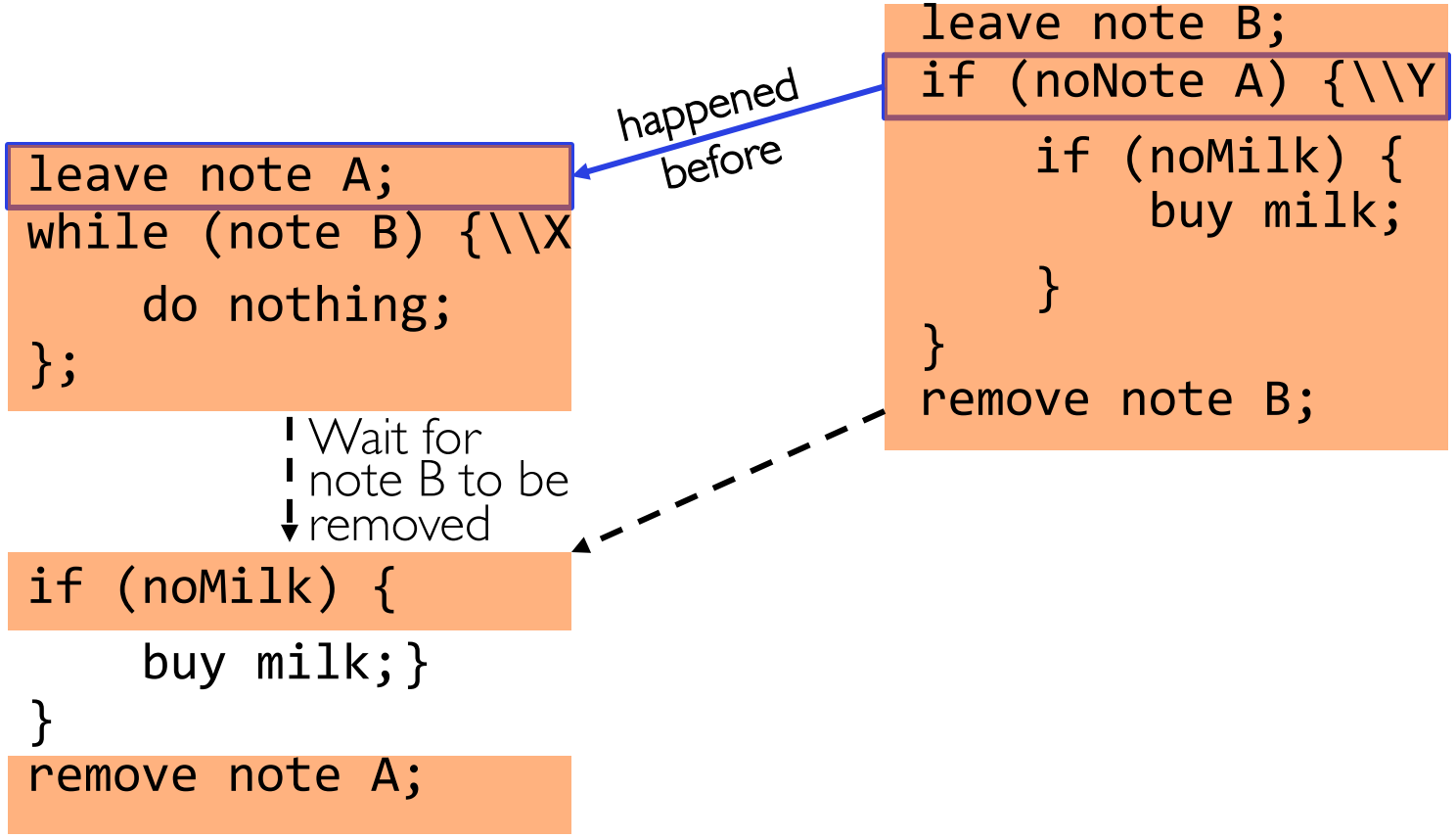
- "`if (noNote A)`" happens before "`leave note A`"




```
if (noMilk) {
    buy milk;}
}
remove note A;
```

Case 2

- "`if (noNote A)`" happens before "`leave note A`"



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
- Semaphores: synchronization mechanism for enforcing resource constraints