Synchronization 3: Lock Implementation

Sam Kumar

CS 162: Operating Systems and System Programming

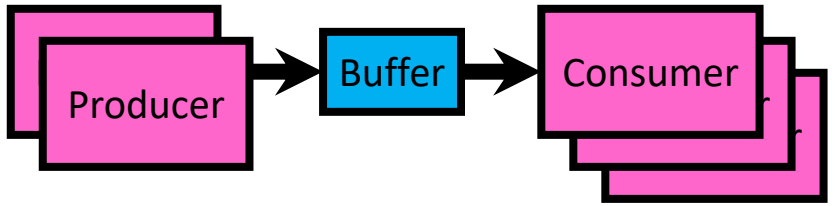
Lecture 10

https://inst.eecs.berkeley.edu/~cs162/su20

Read: A&D 5.1-3, 5.7

Recall: Producer-Consumer

- Problem Definition
  - Producers puts things into a shared buffer
  - Consumers takes them out



- Don’t want producers and consumers to have to work in lockstep, so put a buffer (bounded) between them
  - Need synchronization to maintain integrity of the data structure and coordinate producers/consumers
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty

Recall: Producer-Consumer (Semaphores)

```
Semaphore usedSlots = 0; // No slots used
Semaphore freeSlots = bufSize; // All slots free
Lock mutex = <initially unlocked>; // Nobody in critical sec.
```

```
Producer(item) {
  freeSlots.P();
  mutex.acquire();
  Enqueue(item);
  mutex.release();
  usedSlots.V();
}
```

```
Consumer() {
  usedSlots.P();
  mutex.acquire();
  item = Dequeue();
  mutex.release();
  freeSlots.V();
  return item;
}
```

Recall: Problems with Semaphores

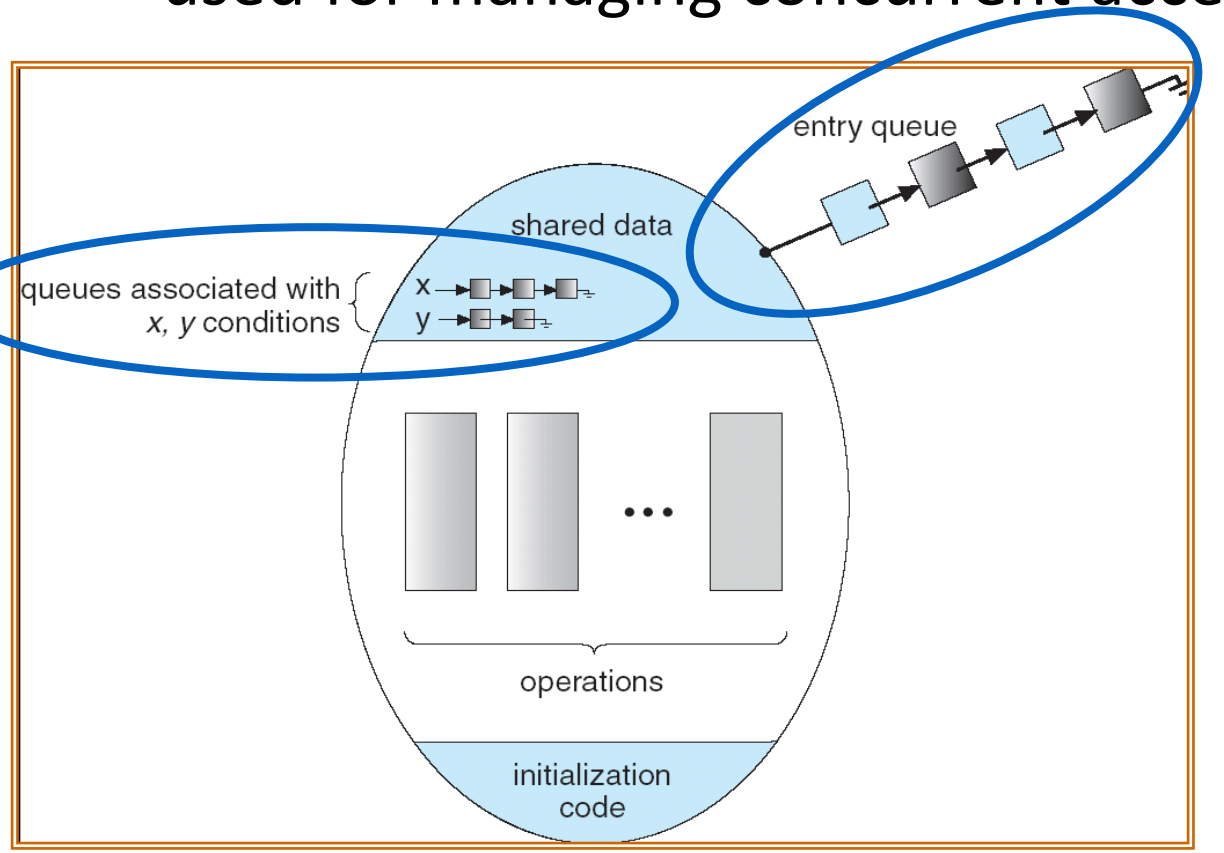
- More powerful (and primitive) than locks
- Argument: Clearer to have separate constructs for
  - Mutual Exclusion: One thread can do something at a time
  - Waiting for a condition to become true
- Need to make sure a thread calls P() for every V()
  - Other tools are more flexible than this

Recall: Condition Variables

- **Queue of threads waiting *inside* a critical section**
  - **Typically, waiting until a condition on some variables becomes true**
  - **Variables typically are protected by a mutex**
- Operations:
  - **`wait(&lock)`:** Atomically release lock and go to sleep until condition variable is signaled. Re-acquire the lock before returning.
  - **`signal()`:** Wake up one waiting thread (if there is one)
  - **`broadcast()`:** Wake up all waiting threads
- **Rule:** Hold lock when using a condition variable

Recall: Monitors

- A monitor consists of a lock and zero or more condition variables used for managing concurrent access to shared data



- Lock: the lock provides mutual exclusion to shared data
- Condition Variable: a queue of threads waiting for something *inside* a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep

Recall: Why the `while` Loop?

- When a thread is woken up by `signal()`, it is simply marked as eligible to run

- It may or may not reacquire the lock immediately!
  - Another thread could be scheduled and "sneak in" make the condition it's waiting for no longer true
  - Need a loop to re-check condition on wakeup

- This is called Mesa Scheduling (Mesa-style Monitors)
- Most operating systems use Mesa-style Monitors!

Recall: Mesa Monitors vs. Hoare Monitors

Mesa Monitor

```
while (buffer empty) {
  cond_wait(&not_empty, &buf_lock);
}
```

Hoare Monitor

```
if (buffer empty) {
  cond_wait(&not_empty, &buf_lock);
}
```

- In practice, almost all OSes implement Mesa monitors

Recall: Java Support for Monitors

- Along with a lock, every object has a **single** condition variable associated with it

- To wait inside a synchronized method:
  - **`void wait();`**
  - **`void wait(long timeout);`**

- To signal while in a synchronized method:
  - **`void notify();`**
  - **`void notifyAll();`**

Recall: Go Channels

- Semantics similar to pipes, with the following differences:
  - Used within a single process (not across processes)
  - Carries language objects/structs, not bytes (no marshalling/unmarshalling)

```
var x chan int = make(chan int, 5)
x <- 162
y := <- x
fmt.Println(y) // Prints 162
```

Today: How to implement synchronization primitives?

For now, just consider *locks inside the kernel.*

Recall: Race Conditions

- What are the possible values of x below?
- Initially `x == 0`

Thread A	Thread B
`x += 1;`	`x += 1;`

- 1 or 2 (non-deterministic)

Thread A

```
ld r1, &x
add r1, r1, 1
st r1, &x
```

Thread B

```
ld r1, &x
add r1, r1, 1
st r1, &x
```

Recall: Race Conditions

- What are the possible values of x below?
- Initially `x == 0`

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

- 1 or 2 (non-deterministic)
- Maybe even 3 for serial processors (!)

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
- 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

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 (reading on course site)
- 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

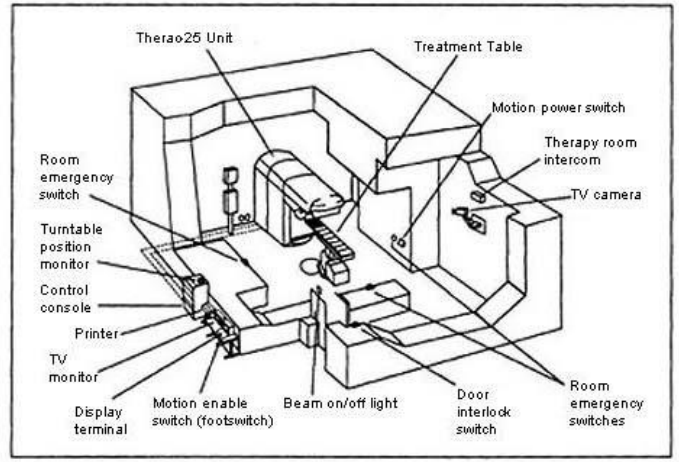


Figure 1. Typical Therac-25 facility

Motivating Example: "Too Much Milk"

- Analogy between problems in OS and problems in real life
- 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

Too Much Milk: Correctness

1. Safety: At most one person buys milk.

2. Liveness: If milk is needed, at least one person buys it.

Attempt #1

- Leave a note
  - Place on fridge before buying
  - Remove after buying
  - Don’t go to store if there’s already a note

- Leaving/checking a note is atomic (word load/store)

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

Attempt #1 in Action

Thread A

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


    leave Note;
    buy milk;
    remove Note;
  }
}
```

Thread B

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


    leave Note;
    buy milk;
    remove note;
  }
}
```

Achieves liveness but not safety

Attempt #1.5

- Idea: leave note, then check for milk

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

But there's always a note – you just left one!

Attempt #2: Use Named Notes

Thread A

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

Thread B

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

Attempt #2 in Action

Thread A

```
leave note A
if (noMilk) {

  if (noNote B) {
    buy milk
  }
}

remove note A
```

Thread B

```
leave note B

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

Achieves safety but not liveness

Attempt #3: Wait

This is a correct solution!

Thread A

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

Thread B

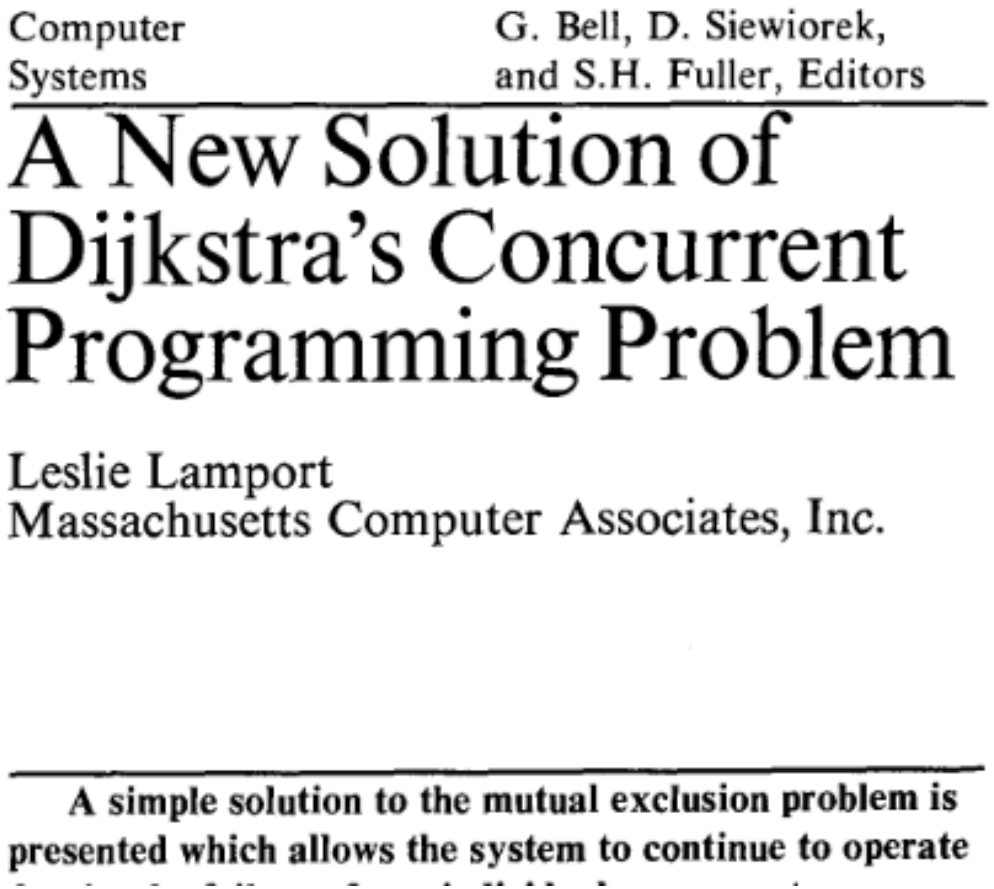
```
leave note B
if (noNote A) {
  if (noMilk) {
    buy milk
  }
}
remove note B
```

This Generalizes to n Threads…

- Leslie Lamport's "Bakery Algorithm" (1974)

- Allows one to protect a critical section like:

```
if (noMilk) {
    buy milk;
}
```



Computer Systems

G. Bell, D. Siewiorek, and S.H. Fuller, Editors

A New Solution of Dijkstra's Concurrent Programming Problem

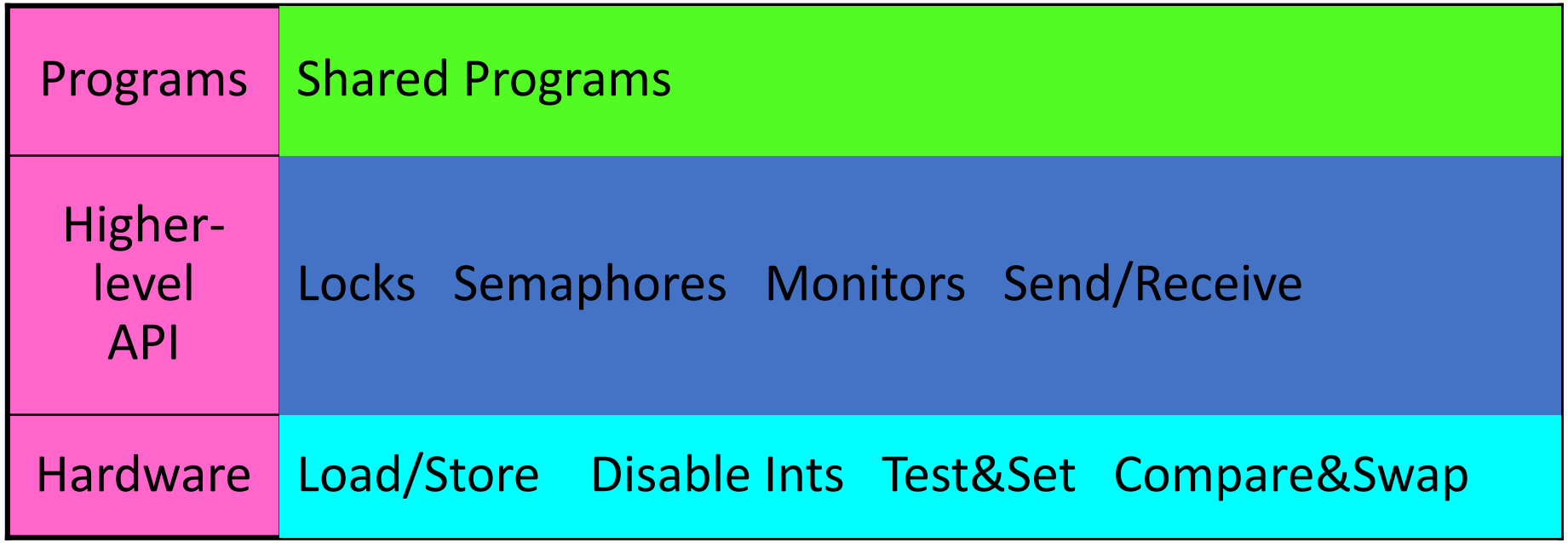
Leslie Lamport
Massachusetts Computer Associates, Inc.

A simple solution to the mutual exclusion problem is presented which allows the system to continue to operate

Solution #3 Discussion

- Solution #3 works, but it's not great
  - Really complex – even for this simple an example
    - Hard to convince yourself that this really works
  - While A is waiting, it is consuming CPU time
    - This is called "busy-waiting"
- There's a better way
  - Have hardware provide higher-level primitives than atomic load & store
  - Build even higher-level programming abstractions on this hardware support
  - *Make sure the OS scheduler never allows another thread to enter the critical section*
    - *The other thread becomes blocked if it tries to enter*

Where are we going with Synchronization?



- Building an efficient, easy-to-use API

Announcements

- Homework 3 is released
- Project 1 design reviews are today
- Project 1 code is due on Tuesday, July 14

Implementing Locks: Single Core

- How can we make lock.Acquire() and lock.Release() appear **atomic** to other threads?

- Idea: A context switch can only happen (assuming threads don't yield) if there's an **interrupt**
- "Solution": **Disable interrupts** while holding lock

- x86 has `cli` and `sti` instructions that only operate in system mode (PL=0)
  - Interrupts enabled bit in FLAGS register

Naïve Interrupt Enable/Disable

```
Acquire() {
  disable interrupts;
}
```

```
Release() {
  enable interrupts;
}
```

- Problem: can stall the entire system

```
Lock.Acquire()
While (1) {}
```

- Problem: What if we want to do I/O?

```
Lock.Acquire()
Read from disk
/* OS waits for (disabled) interrupt! */
```

Implementing Locks: Single Core

- Key idea: maintain a lock variable (**`value`**) and disable interrupts only during operations on that variable

```
int value = FREE;

Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    run_new_thread();
    // Enable interrupts?
  } else {
    value = BUSY;
  }
  enable interrupts;
}
```

```
Release() {
  disable interrupts;
  if (anyone on wait queue) {
    take thread off wait queue;
    Place on ready queue;
  } else {
    value = FREE;
  }
  enable interrupts;
}
```

Discussion

- Why do we need to disable interrupts at all?
  - Avoid interruption between checking and setting lock value
  - Otherwise two threads could think that they both have lock

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      run_new_thread();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

Critical Section

- Disabling interrupts prevents preemption
- Locks disable interrupts to provide *another* critical section

- Unlike the naïve solution, interrupts are disabled for only a short time

Implementing Locks: Single Core

- Key idea: maintain a lock variable (**`value`**) and disable interrupts only during operations on that variable

```
int value = FREE;

Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    run_new_thread();
    // Enable interrupts?
  } else {
    value = BUSY;
  }
  enable interrupts;
}
```

```
Release() {
  disable interrupts;
  if (anyone on wait queue) {
    take thread off wait queue;
    Place on ready queue;
  } else {
    value = FREE;
  }
  enable interrupts;
}
```

Re-enabling Interrupts when Waiting

```
Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    run_new_thread()
  } else {
    value = BUSY;
  }
  enable interrupts;
}
```

enable interrupts

enable interrupts

- Before on the queue?
  - Release might not wake up this thread!
- After putting the thread on the queue?
  - Gets woken up, but immediately switches away

Re-enabling Interrupts when Waiting

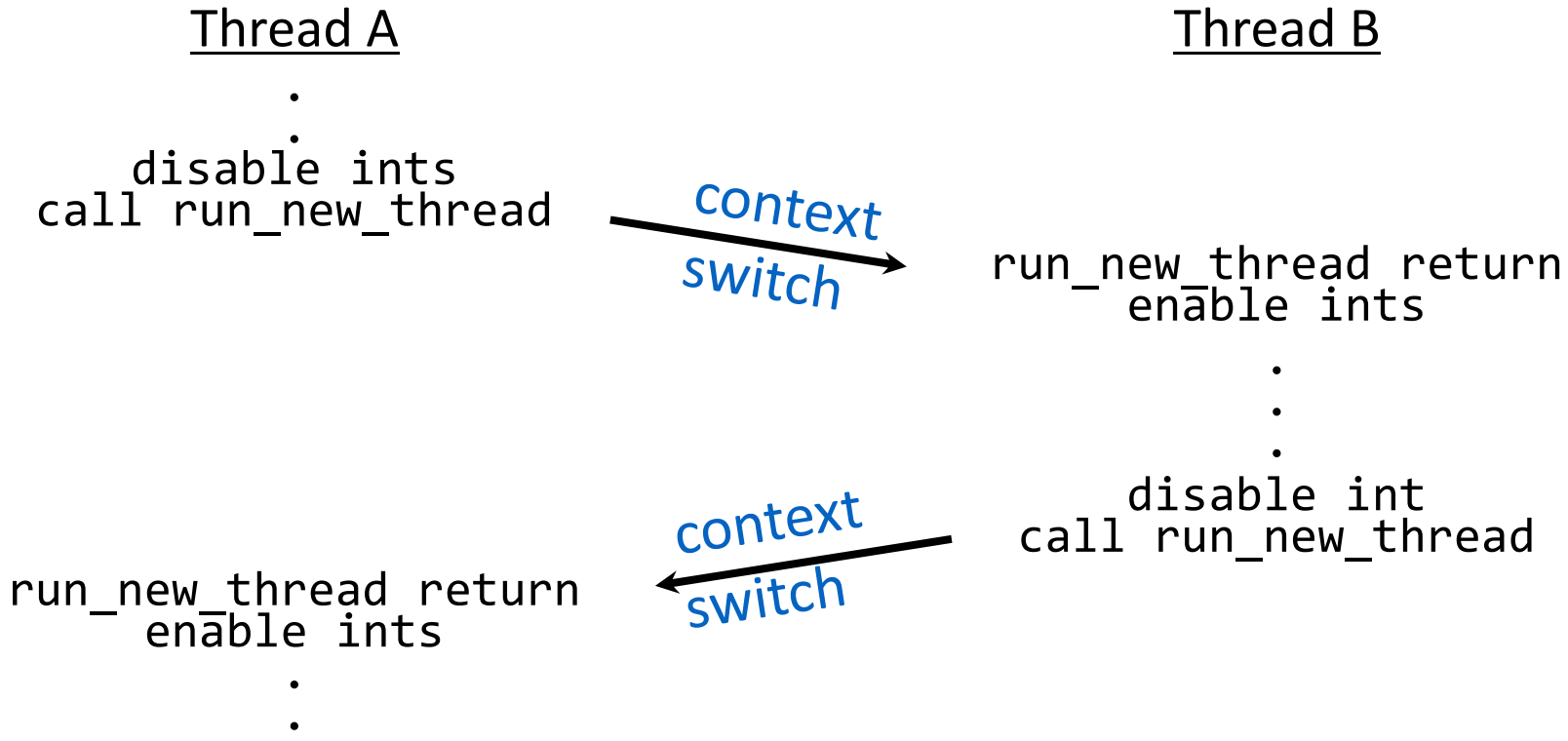
```
Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    run_new_thread()
  } else {
    value = BUSY;
  }
  enable interrupts;
}
```

enable interrupts

- Best solution: after the current thread suspends
- How?
  - `run_new_thread()` should do it!
  - Part of returning from `switch()`

How to Re-enable Interrupts when Waiting

- In scheduler, since interrupts are disabled when switching threads:
  - Responsibility of the next thread to re-enable interrupts
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts



Enabling Interrupts vs. Restoring Interrupts

- 99% of the time, you want to restore interrupts, not enable them
- We used “enable interrupts” in this lecture since we were assuming interrupts are enabled when acquiring the lock
- In Pintos:

```
enum intr_level state = intr_disable();
<code manipulating shared data>
intr_set_level(state);
```

When does this Lock Implementation Work?

- **Answer: For threads *in the kernel* on a *single-core* machine.**

Roadmap for today's lecture:

1. What about multi-core machines?
2. What about user threads?

Break

Multi-Core Machines

- How to synchronize with threads executing in parallel on other cores?
  - Disable interrupts on all cores?
  - Prevent other cores from making progress?

- Implement locks in hardware?
  - What's the interface between hardware lock and OS scheduler?

- Solution: Use hardware support for **atomic operations**

Atomic Operations

- Definition: **An operation runs to completion or not at all**
- Foundation for synchronization primitives

- Example: Loading or storing a word (on most modern architectures)

Atomic Read-Modify-Write Instructions

- These instructions read a value and write a new value atomically
- Hardware is responsible for implementing this correctly
  - on both uniprocessors (not too hard)
  - and multiprocessors (requires help from cache coherence protocol)
- Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors
- Natural extensions to user-level locking

Examples of Read-Modify Write

- ```
  test&set (&address) {              /* most architectures */
      result = M[address];           // return result from "address" and
      M[address] = 1;                // set value at "address" to 1
      return result;
  }
  ```
- ```
  swap (&address, register) {        /* x86 */
      temp = M[address];             // swap register's value to
      M[address] = register;         // value at "address"
      register = temp;
  }
  ```
- ```
  compare&swap (&address, reg1, reg2) { /* 68000 */
      if (reg1 == M[address]) {      // If memory still == reg1,
          M[address] = reg2;         // then  put reg2 => memory
          return success;
      } else {                       // Otherwise do not change memory
          return failure;
      }
  }
  ```
- ```
  load-linked&store-conditional(&address) { /* R4000, alpha */
      loop:
          ll r1, M[address];
          movi r2, 1;                // Can do arbitrary computation
          sc r2, M[address];
          beqz r2, loop;
  }
  ```

Implementing Locks with `test&set`

- Simple, but flawed, solution:

```
int value = 0; // Free

Acquire() {
   while (test&set(value)) {}; // spin while busy
}

Release() {
   value = 0;                  // atomic store
}
```

- Explanation:
  - If lock is free, `test&set` reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
  - If lock is busy, `test&set` reads 1 and sets value=1 (no change). It returns 1, so while loop continues.
  - When we set value = 0, someone else can get lock.
- Busy-Waiting: thread consumes cycles while waiting
- For multiprocessor cache coherence: every test&set() is a write, which makes value ping-pong around in cache (using lots of memory BW)

This is Called a *Spinlock*

- Spinlock implementation:

```
int value = 0; // Free

Acquire() {
   while (test&set(value)) {}; // spin while busy
}

Release() {
   value = 0;                  // atomic store
}
```

- Spinlock doesn't put the calling thread to sleep --- it just busy waits

Problem: Busy-Waiting for Lock

- Positives for this solution
  - Machine can receive interrupts
  - User code can use this lock (poorly)
  - Works on a multiprocessor
- Negatives
  - Very inefficient: thread will consume cycles waiting
  - Waiting thread takes cycles away from thread holding lock (no one wins!)
  - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!
- For semaphores (and monitors), waiting thread may wait for an arbitrary long time!
  - Thus even if busy-waiting was OK for locks, definitely not OK for other primitives
  - Homework/exam solutions should avoid busy-waiting!

Better Locks Using test&set

- Can we build test&set locks without busy-waiting?
  - Can't entirely, but can minimize!
  - Idea: only busy-wait to atomically *check* lock value

```
int guard = 0;
int value = FREE;

Acquire() {
   // Short busy-wait time
   while (test&set(guard));
   if (value == BUSY) {
      put thread on wait queue;
      run_new_thread() & guard = 0;
   } else {
      value = BUSY;
      guard = 0;
   }
}
```

```
Release() {
   // Short busy-wait time
   while (test&set(guard));
   if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   guard = 0;
```

- Note: sleep has to be sure to reset the guard variable
  - Why can't we do it just before or just after the sleep?

Alternative View: Bootstrapping a Spinlock

```
SpinLock guard = FREE;
int value = FREE;

Acquire() {
   // Short busy-wait time
   guard.Acquire();
   if (value == BUSY) {
      put thread on wait queue;
      run_new_thread() & guard.Release();
   } else {
      value = BUSY;
      guard.Release();
   }
}
```

```
Release() {
   // Short busy-wait time
   guard.Acquire();
   if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   guard.Release();
```

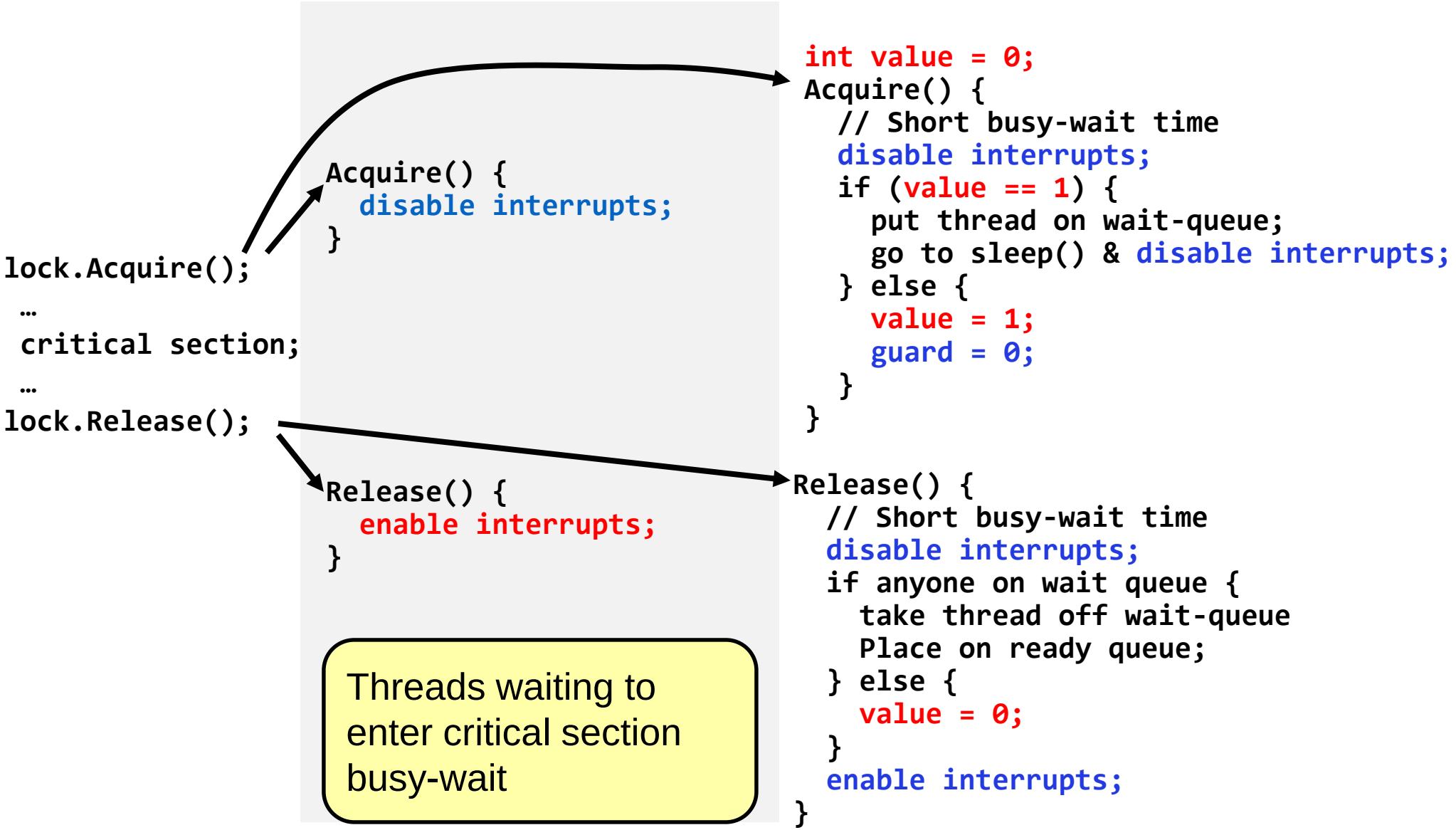
Comparison to Disabling Interrupts

- We changed disable interrupts → spinlock.Acquire() [while (test&set(guard))]
- We changed enable interrupts → spinlock.Release() [guard = 0]

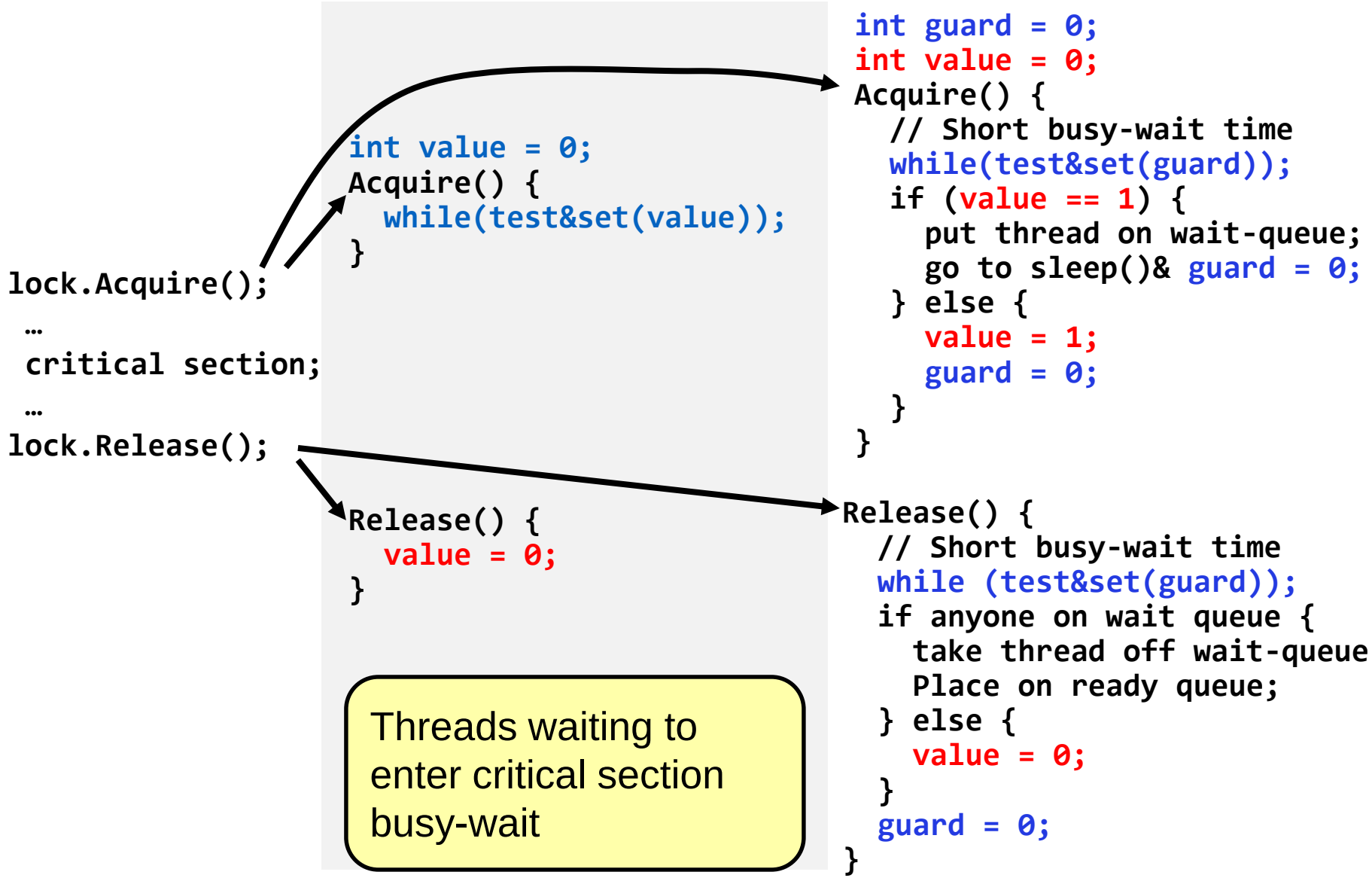
```
int value = FREE;

Acquire() {
   // Short busy-wait time
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      run_new_thread();
      // scheduler enables interrupts
   } else {
      value = BUSY;
      enable interrupts;
   }
}
```

```
Release() {
   disable interrupts;
   if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
```

Recap: Locks Using Interrupts



Recap: Locks Using test&set



Recall: *Spinlock*

- Spinlock implementation:

```
int value = 0; // Free

Acquire() {
   while (test&set(value)) {}; // spin while busy
}

Release() {
   value = 0;                  // atomic store
}
```

- Spinlock doesn't put the calling thread to sleep—it just busy waits
  - When might this be preferable?
- For multiprocessor cache coherence: every test&set() is a write, which makes value ping-pong around in cache (using lots of memory BW)

Better Spinlock: `test&test&set`

- A better spinlock solution:

```
int mylock = 0; // Free

Acquire() {
   do {
     while(mylock);   // Wait until might be free
   } while(test&set(&mylock)); // exit if get lock
}


Release() {
   mylock = 0;
}
```

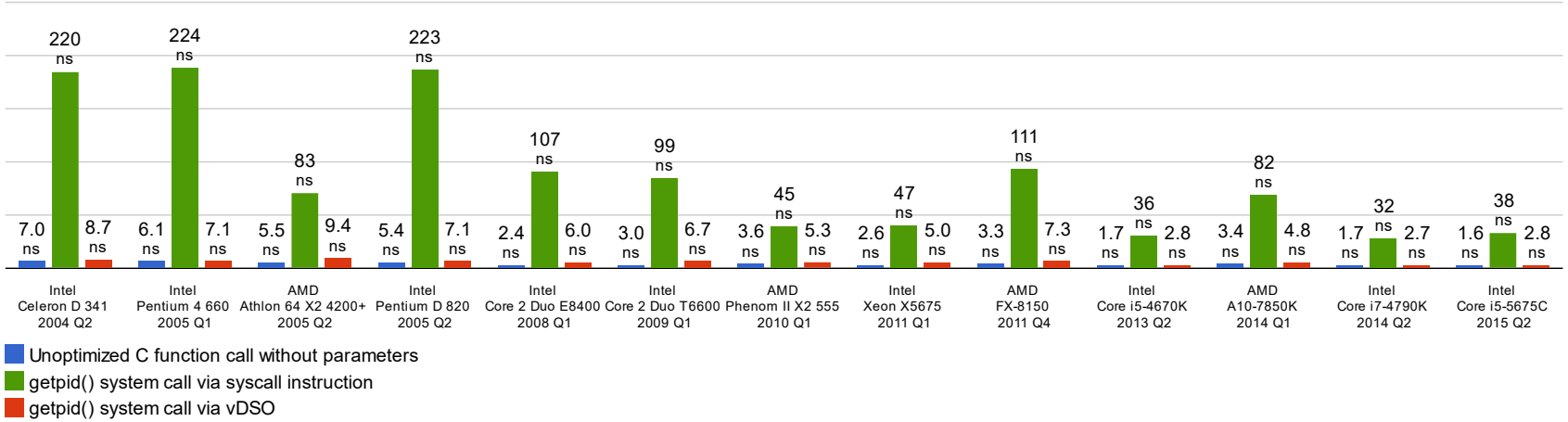
- Explanation:
  - Wait until lock might be free (only reading – stays in cache)
  - Then, try to grab lock with `test&set`
  - Repeat if fail to actually get lock
- Busy-Waiting: no longer impacts other processors!

Locks in Userspace?

- We've looked at locks in the kernel
  - Uniprocessor case (disable interrupts)
  - Multiprocessor case (`test&set`)

- What about locks in userspace?
- Spinlocks just work
- Simple idea for non-busy-waiting lock:
  - For each userspace lock, allocate a lock in the kernel
  - Make a syscall for each acquire/release operation to acquire the lock in the kernel

Recall: Overhead of Syscalls



- Syscalls are 25x more expensive than function calls (~100 ns)

- `read`/`write` a file byte by byte? Max throughput of **~10MB/second**
- With **`fgetc`**? Keeps up with your SSD

Userspace Locks: Syscall Overhead

- Can we avoid syscall overhead when acquiring a non-busy-waiting lock in userspace?
  - No: can't put a thread to sleep (i.e., block the thread) without entering the kernel
- What we can do: avoid system calls in the uncontended case (i.e., the case where we can acquire the lock without blocking)
  - Helps both uniprocessor case and multiprocessor case

Linux futex: Fast Userspace Mutex

```
#include <linux/futex.h>
#include <sys/time.h>

int futex(int *uaddr, int futex_op, int val,
      const struct timespec *timeout );
```

- uaddr points to a 32-bit value in user space
- futex_op
  - FUTEX_WAIT – if val == *uaddr sleep till FUTEX_WAIT
    - ***Atomic*** check that condition still holds
  - FUTEX_WAKE – wake up at most val waiting threads
  - FUTEX_FD, FUTEX_WAKE_OP, FUTEX_CMP_REQUEUE
- timeout
  - ptr to a *timespec* structure that specifies a timeout for the op

Linux `futex`: Fast Userspace Mutex

- Idea: Userspace lock is *syscall-free* in the uncontended case

- Lock has three states
  - Free (no syscall when acquiring lock)
  - Busy, no waiters (no syscall when releasing lock)
  - Busy, possibly with some waiters

- `futex` is not exposed in libc; it is used within the implementation of pthreads

Example: Userspace Locks with `futex`

```
int value = 0; // free
bool maybe_waiters = false;

Acquire() {
   while (test&set(value)) {
      maybe_waiters = true;
      futex(&value, FUTEX_WAIT, 1);
      // futex: sleep if lock is acquired
      maybe_waiters = true;
   }
}
```

```
Release() {
   value = 0;
   if (maybe_waiters) {
      maybe_waiters = false;
      futex(&value, FUTEX_WAKE, 1);
      // futex: wake up a sleeping thread
   }
}
```

- This is syscall-free in the uncontended case
  - Temporarily falls back to syscalls if multiple waiters, or concurrent acquire/release
- But it can be considerably optimized!
  - See "Futexes are Tricky" by Ulrich Drepper

Conclusion

- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Talked about hardware atomicity primitives:
  - Disabling of Interrupts, test&set, swap, compare&swap, load-locked & store-conditional
- Showed several constructions of Locks
  - Must be very careful not to waste/tie up machine resources
    - Shouldn't disable interrupts for long
    - Shouldn't spin wait for long
  - Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable

Bonus Slides (If Time)

Further Reducing Overhead

- Make locks less contended [how?]

- **Move synchronization and scheduling into userspace**

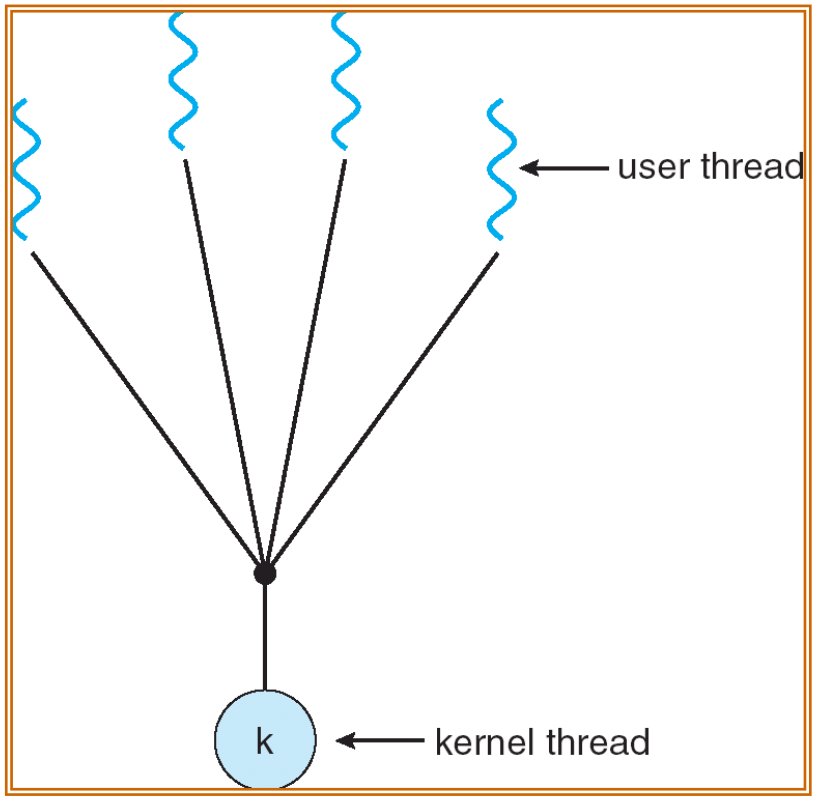
We've Looked At: Kernel-Supported Threads

- Threads run and block (e.g., on I/O) independently
- One process may have multiple threads waiting on different things
- Two mode switches for every context switch (expensive)
- Create threads with syscalls

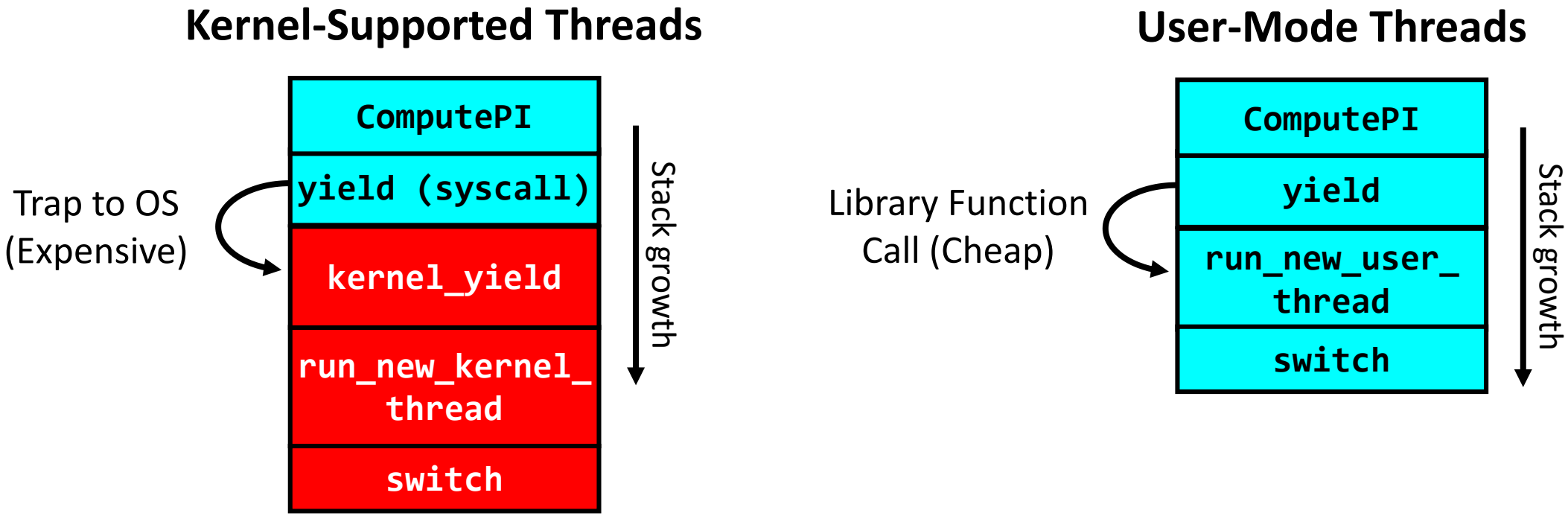
- Alternative: multiplex several streams of execution (at user level) on top of a single OS thread
  - E.g., Java, Go, … (and many many user-level threads libraries before it)

User-Mode Threads

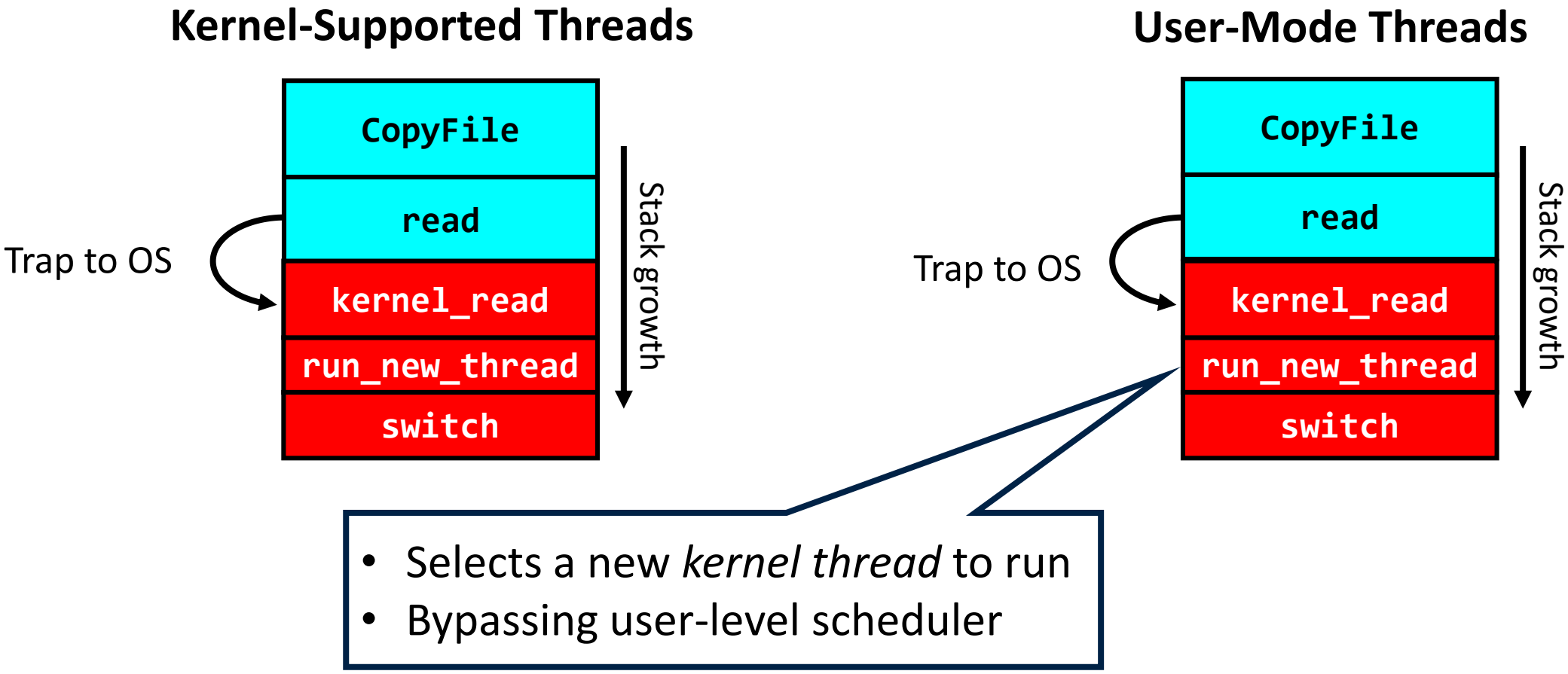
- User program contains its own scheduler
- Several user threads per kernel thread
- User threads may be scheduled non-preemptively
  - Only switch on `yield`
- Context switches cheaper
  - Copy registers and jump (`switch` in userspace)



Thread Yield



Thread I/O


Kernel-Supported Threads
CopyFile
read
kernel_read
run_new_thread
switch
Trap to OS
Stack growth
User-Mode Threads
CopyFile
read
kernel_read
run_new_thread
switch
Trap to OS
Stack growth
• Selects a new *kernel thread* to run
• Bypassing user-level scheduler

User-Mode Threads: Problems

- One user-level thread blocks on I/O: they all do
  - Kernel cannot adjust scheduling among threads it doesn't know about
- Multiple Cores?
- Can't completely avoid blocking (syscalls, page fault)
- One Solution: *Scheduler Activations*
  - Have kernel inform user-level scheduler when a thread blocks
  - Evolving the contract between OS and application

- Alternative Solution: Language Support?
  - Make the scheduler aware of the blocking operation

Go Goroutines

- Goroutines are lightweight, user-level threads
  - Scheduling not preemptive (relies on goroutines to yield)
  - Yield statements inserted by compiler

- Advantages relative to regular threads (e.g., pthreads)
  - More lightweight
  - Faster context-switch time

- Disadvantages
  - Less sophisticated scheduling at the user-level
  - OS is not aware of user-level threads

Go User-Level Scheduler

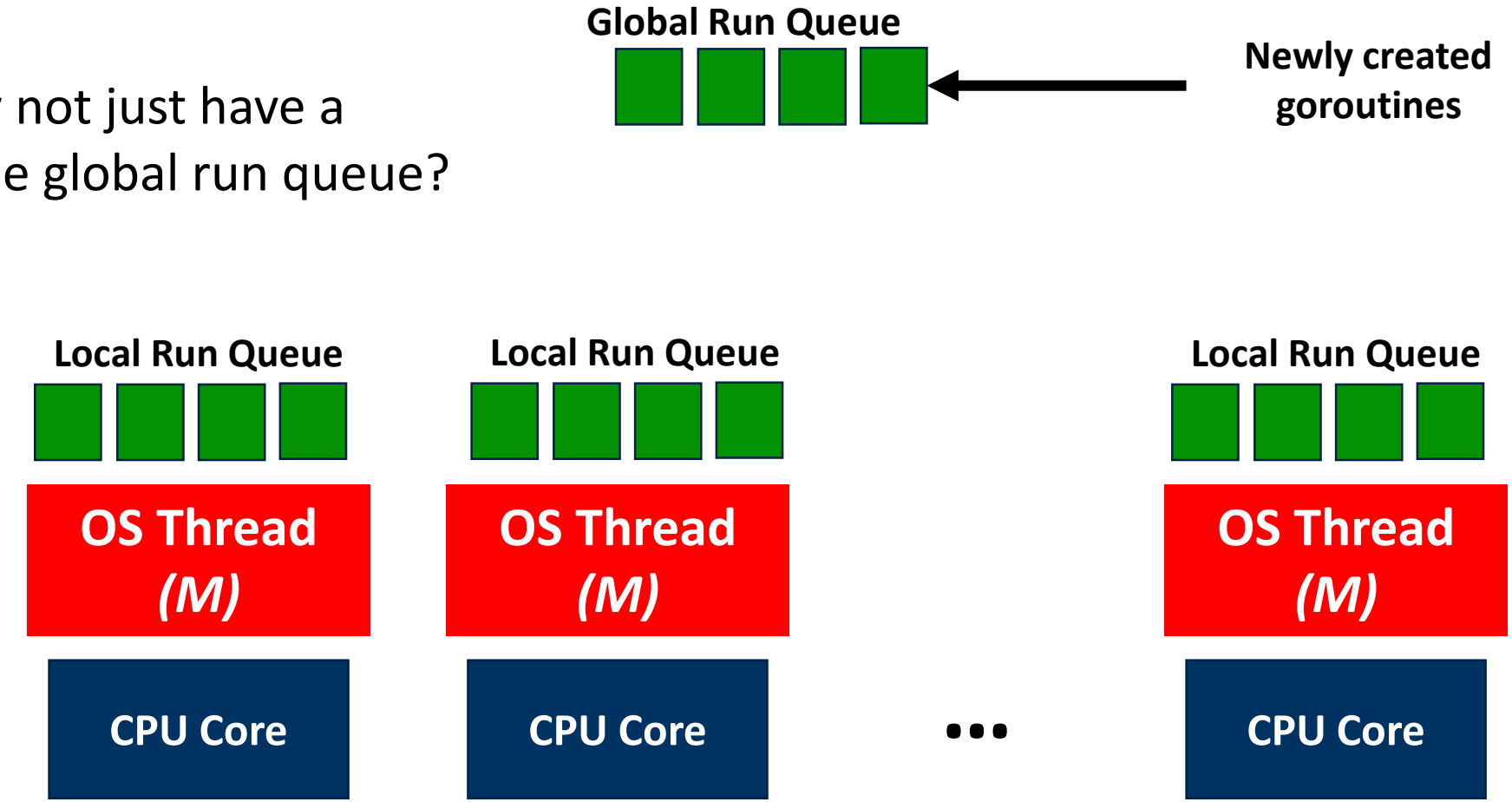
Why this approach?

- 1 OS (kernel-supported) thread per CPU core: allows go program to achieve *parallelism* not just *concurrency*
  - Fewer OS threads? Not utilizing all CPUs
  - More OS threads? No additional benefit
    - We'll see one exception to this involving syscalls

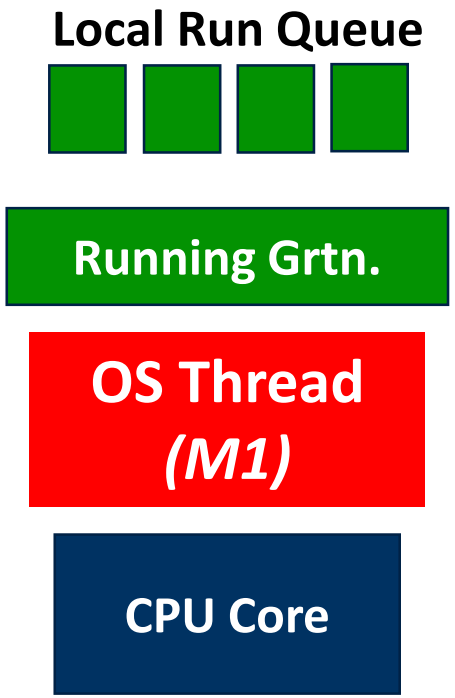
- Keep goroutine on same OS thread: *affinity*, nice for caching and performance

Go User-Level Thread Scheduler

- Why not just have a single global run queue?

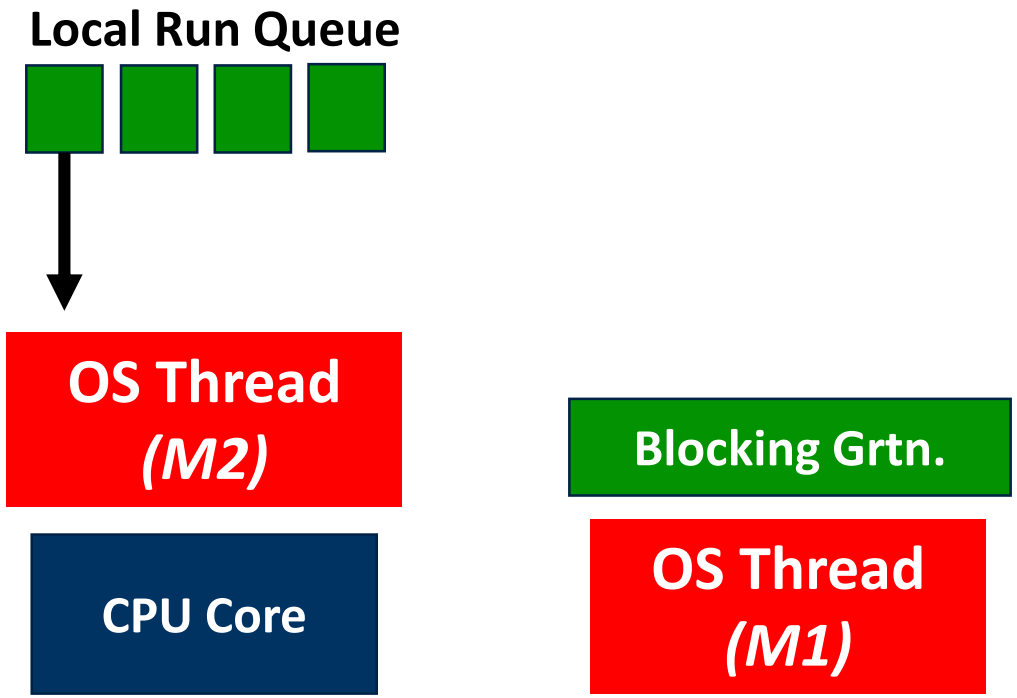


Dealing with Blocking Syscalls



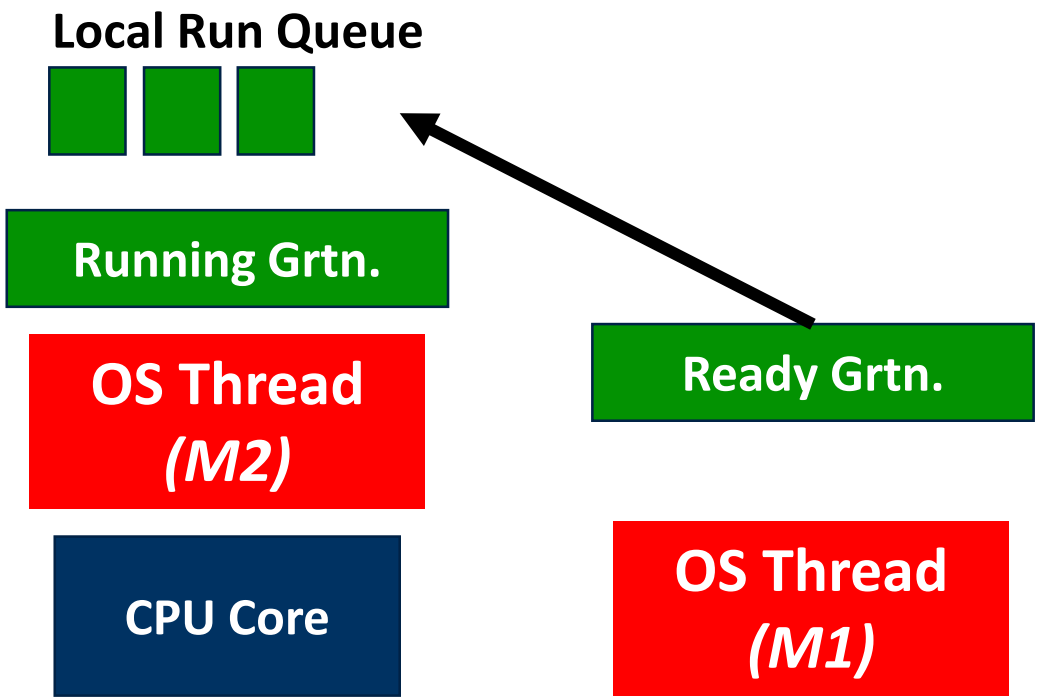
- What if a goroutine wants to make a blocking syscall?
  - Example: File I/O

Dealing with Blocking Syscalls



- What if a goroutine wants to make a blocking syscall?
  - Example: File I/O
- While syscall is blocking, allocate new OS thread (M2)
  - M1 is blocked by kernel, M2 lets us continue using CPU

Dealing with Blocking Syscalls



- Syscall completes: Put invoking goroutine back on queue
- Keep *M1* around in a spare pool
- Swap it with *M2* upon next syscall, no need to pay thread creation cost