Lecture 08: Synchronization Bugs

CS343 – Operating Systems

Branden Ghena – Spring 2024

Some slides borrowed from:
Stephen Tarzia (Northwestern), Harsha Madhyastha (Michigan), Shivaram Venkataraman (Wisconsin), and UC Berkeley CS162

Today's Goals

- Common synchronization bugs
  - Deadlock
  - Livelock
- Methods to avoid, prevent, and recover in the presence of deadlock
- Discuss how thread-safe data structures might work
- Touch on what concurrency looks like in other languages

Outline

- **Interrupts**
- Synchronization bugs
  - Deadlock
    - Solving deadlocks
  - Livelock
  - Priority Inversion
- Threadsafe data structures
- Concurrency in other languages

Where else does concurrency come from?

- Processors introduce it for performance reasons by running multiple processes and threads

- Interactions with the outside world introduce it because events occur whenever they feel like it
  - Network request arriving
  - User presses a key
  - Motion sensor triggers

- Also, we need some way to deal with errors the occur when executing instructions
  - No pathway for returning an error from an instruction

Interrupts

- An event that the processor handles by running special OS handler code
  - Timer expiration, Keyboard event, Network packet, etc.
  - Necessary for asynchronous event handling
    - Don’t wait around for the event, just handle it whenever it happens

- Very similar to Exceptions, which are caused by errors occurring

- A system call is a way to generate a software interrupt

Differences from system calls

- When we performed a system call:
  - We knew it was about to happen
  - Set up our registers in advance
  - Performed what looked sort of like a function call
  - And we were always switching from process to kernel

- Interrupts can happen *whenever.*
  - This can get extremely complicated on modern systems with out-of-order execution, multiple cores and threads, and caches

Interrupt Vector Table

Table 6-1. Exceptions and Interrupts

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

Table actually lives in memory somewhere, with function pointers for each vector number

```
match interrupt {
    nvic::ASTALARM => ast::AST.handle_interrupt(),

    nvic::USART0 => usart::USART0.handle_interrupt(),
    nvic::USART1 => usart::USART1.handle_interrupt(),
    nvic::USART2 => usart::USART2.handle_interrupt(),
    nvic::USART3 => usart::USART3.handle_interrupt(),

    nvic::PDCA0 => dma::DMA_CHANNELS[0].handle_interrupt(),
    nvic::PDCA1 => dma::DMA_CHANNELS[1].handle_interrupt(),
    nvic::PDCA2 => dma::DMA_CHANNELS[2].handle_interrupt(),
    nvic::PDCA3 => dma::DMA_CHANNELS[3].handle_interrupt(),
    nvic::PDCA4 => dma::DMA_CHANNELS[4].handle_interrupt(),
    nvic::PDCA5 => dma::DMA_CHANNELS[5].handle_interrupt(),
    nvic::PDCA6 => dma::DMA_CHANNELS[6].handle_interrupt(),
    nvic::PDCA7 => dma::DMA_CHANNELS[7].handle_interrupt(),
    nvic::PDCA8 => dma::DMA_CHANNELS[8].handle_interrupt(),
    nvic::PDCA9 => dma::DMA_CHANNELS[9].handle_interrupt(),
    nvic::PDCA10 => dma::DMA_CHANNELS[10].handle_interrupt(),
    nvic::PDCA11 => dma::DMA_CHANNELS[11].handle_interrupt(),
    nvic::PDCA12 => dma::DMA_CHANNELS[12].handle_interrupt(),
    nvic::PDCA13 => dma::DMA_CHANNELS[13].handle_interrupt(),
    nvic::PDCA14 => dma::DMA_CHANNELS[14].handle_interrupt(),
    nvic::PDCA15 => dma::DMA_CHANNELS[15].handle_interrupt(),
```

Example from Tock for SAM4L chip (in Rust)

Interrupt Vector Table

Table 6-1. Exceptions and Interrupts

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

Table actually lives in memory somewhere, with function pointers for each vector number

```
match interrupt {
    nvic::ASTALARM => ast::AST.handle_interrupt(),

    nvic::USART0 => usart::USART0.handle_interrupt(),
    nvic::USART1 => usart::USART1.handle_interrupt(),
    nvic::USART2 => usart::USART2.handle_interrupt(),
    nvic::USART3 => usart::USART3.handle_interrupt(),

    nvic::PDCA0 => dma::DMA_CHANNELS[0].handle_interrupt(),
    nvic::PDCA1 => dma::DMA_CHANNELS[1].handle_interrupt(),
    nvic::PDCA2 => dma::DMA_CHANNELS[2].handle_interrupt(),
    nvic::PDCA3 => dma::DMA_CHANNELS[3].handle_interrupt(),
    nvic::PDCA4 => dma::DMA_CHANNELS[4].handle_interrupt(),
    nvic::PDCA5 => dma::DMA_CHANNELS[5].handle_interrupt(),
    nvic::PDCA6 => dma::DMA_CHANNELS[6].handle_interrupt(),
    nvic::PDCA7 => dma::DMA_CHANNELS[7].handle_interrupt(),
    nvic::PDCA8 => dma::DMA_CHANNELS[8].handle_interrupt(),
    nvic::PDCA9 => dma::DMA_CHANNELS[9].handle_interrupt(),
    nvic::PDCA10 => dma::DMA_CHANNELS[10].handle_interrupt(),
    nvic::PDCA11 => dma::DMA_CHANNELS[11].handle_interrupt(),
    nvic::PDCA12 => dma::DMA_CHANNELS[12].handle_interrupt(),
    nvic::PDCA13 => dma::DMA_CHANNELS[13].handle_interrupt(),
    nvic::PDCA14 => dma::DMA_CHANNELS[14].handle_interrupt(),
    nvic::PDCA15 => dma::DMA_CHANNELS[15].handle_interrupt(),
```

Example from Tock for SAM4L chip (in Rust)

Interrupt handlers

- Interrupt context
  - Running code in a special mode
  - Pauses whatever was running previously (kernel or process) until finished
- Handler code
  - Execute some *quick* processing to deal with the interrupt
  - Return so the hardware can bring us back to our normal operation
  - Cannot pause to wait for something else to finish first because the entire core jumped to handling this interrupt
- Handled by the operating system kernel
  - Processes are interrupted, but otherwise not normally involved

Why are interrupts important to concurrency?

- Interrupts are a case where the kernel could have a data race with itself!!
  - Imagine being in the middle of an operation on a device
  - When an interrupt comes in for that same device
  - Data structures for the device could end up messed up

- Takeaway: concurrency isn’t just about processes and threads
  - Many different software designs need to deal with it

Data race fix for single-core machines: disable interrupts

```
void lock() {
  disable_interrupts();
}

void unlock() {
  enable_interrupts();
}
```

- Disable interrupts to prevent preemption during critical section
  - Scheduler can’t run if the OS never takes control
  - Also stops data races in interrupt handlers
- Problems
  - Doesn’t work by itself on multicore machines
    - Need to use it AND mutexes
  - Bad Idea™ to let processes disable the OS
    - Process could freeze the entire computer
  - Might screw up timing for interrupt handling

Outline

- Interrupts
- **Synchronization bugs**
  - Deadlock
    - Solving deadlocks
  - Livelock
  - Priority Inversion
- Threadsafe data structures
- Concurrency in other languages

Common synchronization bugs

- Atomicity violation
  - An operation that should have been atomic wasn't
- Order violation
  - Something happens sooner (or later) than expected
- Deadlock
  - Two threads wait indefinitely on each other
- Livelock (not that common in practice)
  - Two threads repeatedly block each other from proceeding and retry

Atomicity Violation

- Failing to make an entire option atomic
  - Must lock all references to shared memory which could be a data race
  - Must handle entire indeterminant state in one atomic section

Should have been included in critical section

```
lock(lck);
count++;
unlock(lck);

if (count == MAX) {
  count = 0;
}
```

Check your understanding: atomicity violation

- What's wrong here?
  - Every access is locked, right?
- Here, calling close() and setting the file to NULL need to be one atomic operation
  - Otherwise the main thread could try to use to file when it's closed
- Example of failing to resolve indeterminant state atomically

```
Main Thread
lock(lck);
if (file == NULL) {
  file = open("~/myfile.txt");
}
write(file, "hello file");
unlock(lck);

…
Some Other Thread
lock(lck);
close(file);
unlock(lck);

// do some unrelated work

lock(lck);
file = NULL;
unlock(lck);
```

Order violation

- Code often requires a certain ordering of operations, especially:
  - Objects must be initialized before they're used
  - Objects cannot be freed while they are still in use
  - Resolve with semaphores or condvars

Parent

```
file = open("file.dat");
thread_create(child_fcn);
// do some work
…
close(file);
```

Child Thread

```
child_fcn() {
  write(file, "hello");
}
```

Close must happen after *write*, but code does not enforce this ordering.

Why is this difficult?

- It seems like we can just add lots of locks and semaphores to be safe, right?
  - Still tricky! Too many locks can cause ***deadlock*** – indefinite waiting.

- How about just one big lock?
  - (+) Cannot deadlock with one lock (unless there are interrupts)
  - (–) However, this would ***limit concurrency***
    - If every task requires the same lock, then unrelated tasks cannot proceed in parallel.

- Concurrent code is always difficult to write ☹
  - Although somewhat easier with *some* higher-level languages

Locking granularity

- ***Coarse grained lock***:
  - Use one (or a few) locks to protect all (or large chunks of) shared state
  - Linux kernel < version 2.6.39 used one "Big Kernel Lock"
  - Essentially only one thread (CPU core) could run kernel code
  - It's simple but there is much contention for this lock, and concurrency is limited

- ***Fine grained locks***:
  - Use many locks, each protecting small chunks of related shared state
  - Leads to more concurrency and better performance
  - However, there is greater risk of ***deadlock***

Outline

- Interrupts
- **Synchronization bugs**
  - **Deadlock**
    - Solving deadlocks
  - Livelock
  - Priority Inversion
- Threadsafe data structures
- Concurrency in other languages

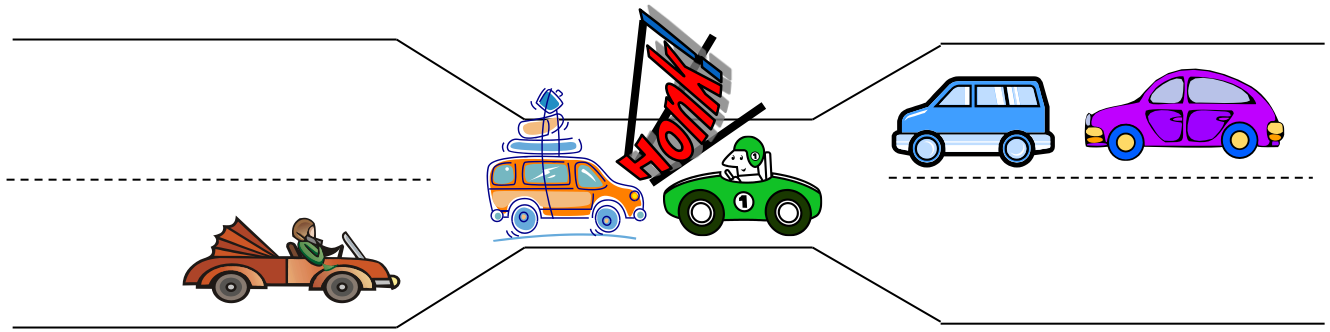

BANCO BRAL
11455

Deadlock

- A concurrency bug arising when:
  - Two threads are each waiting for the other to release a resource.
  - While waiting, the threads cannot release the resource already held.
    - Or at least *do not* release it
  - So the two threads ***wait forever***.

- Can arise when ***multiple*** shared resources are used.
  - For example, acquiring two or more locks.

Deadlock versus starvation

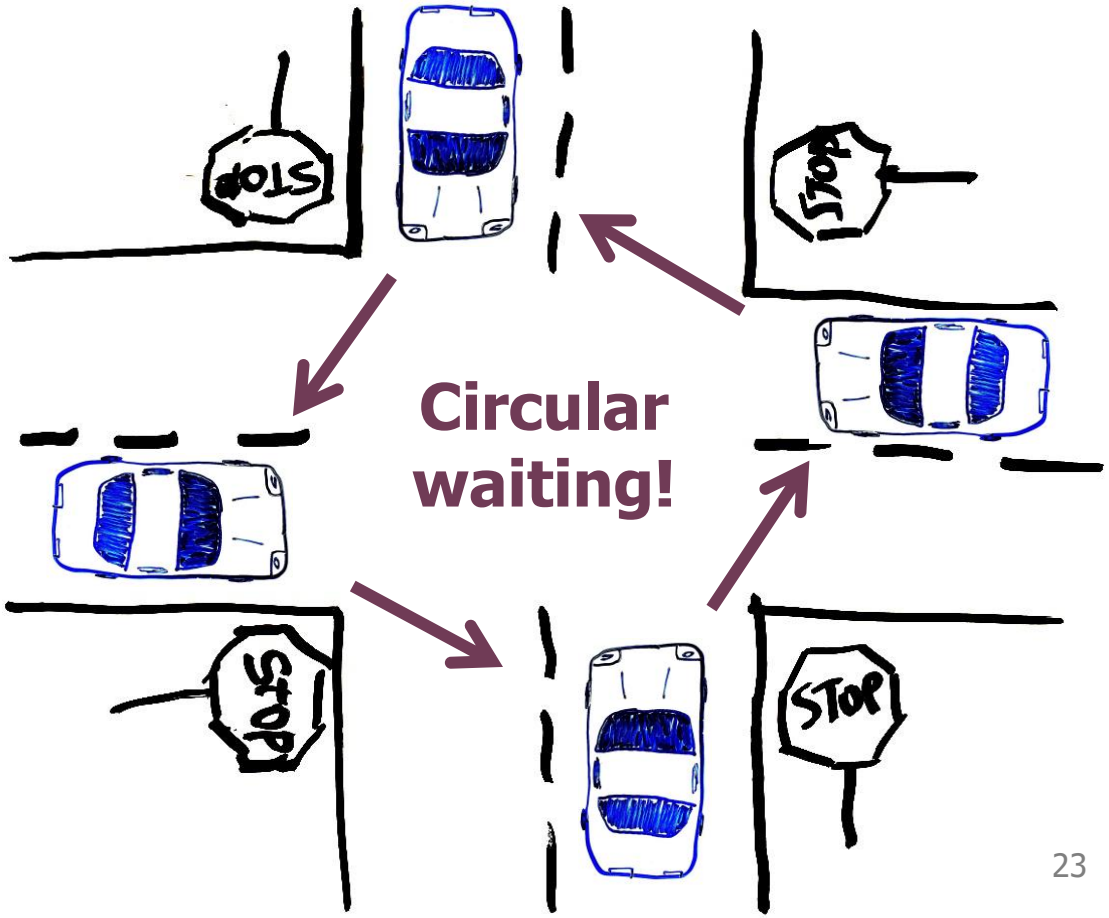


- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into

- **Deadlock:** Two cars in opposite directions meet in middle
- **Starvation** (not deadlock): Eastbound traffic doesn't stop for westbound traffic

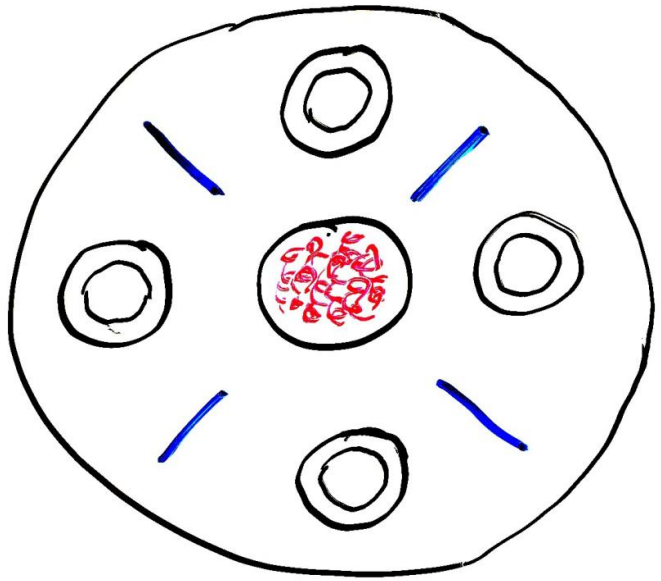
Simple example: four-way stop

- Traffic rules state that you must **yield to the car on your right** if you reach the intersection simultaneously.
- This rule usually works well.
- But there's a problem if four cars arrive simultaneously.



Dining philosophers

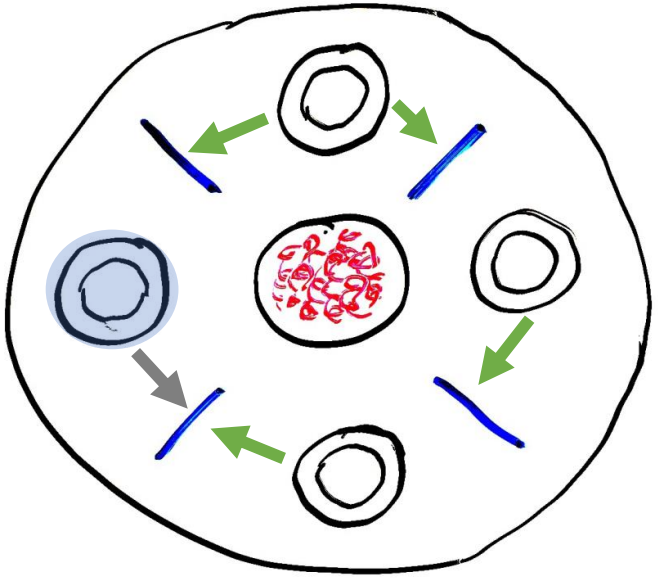
- A theoretical example of deadlock
- There are N philosophers sitting in a circle and N chopsticks
  - left and right of each philosopher
- Philosophers repeatedly run this loop:
  1. Think for some time
  2. Grab chopstick to left
  3. Grab chopstick to right
  4. Eat
  5. Replace chopsticks



- If they all grab the left chopstick simultaneously (step 2), they will deadlock and starve!

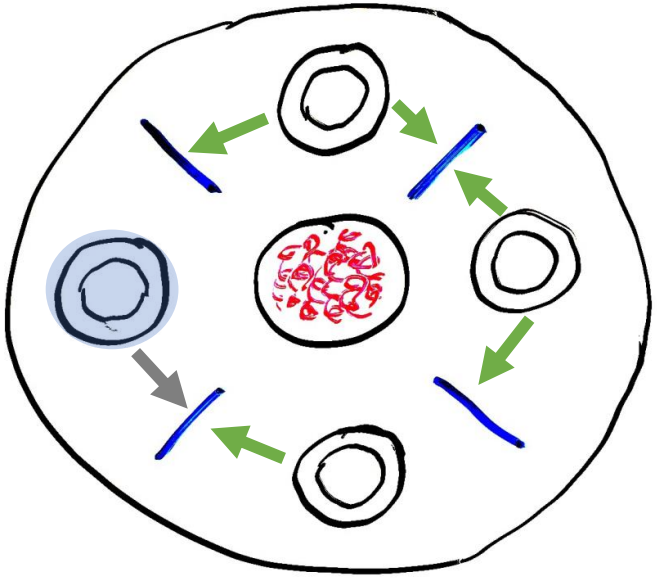
Dining philosophers

- A solution: one philosopher must grab right before left



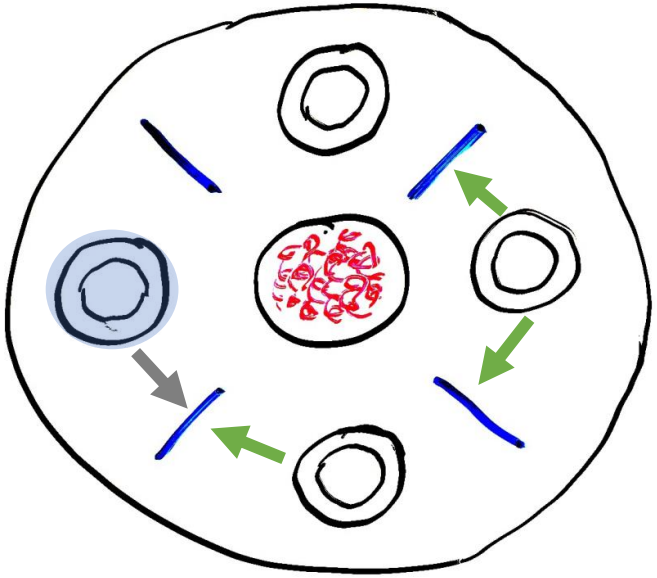
Dining philosophers

- A solution: one philosopher must grab right before left



Dining philosophers

- A solution: one philosopher must grab right before left
  - Adding an asymmetry will allow both resources to eventually be obtained



Deadlock with locks

Thread A

```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

Thread B

```
y.Acquire();
x.Acquire();
…
x.Release();
y.Release();
```

- This is a Nondeterministic Deadlock
  - Whether it occurs depends on scheduling

No deadlock in the lucky case

Thread A

```
x.Acquire();
y.Acquire();

…
y.Release();
x.Release();
```

Thread B

```
y.Acquire();
```

Thread B waits until Thread A is finished

```
x.Acquire();
…
x.Release();
y.Release();
```

But deadlock can still occur

Thread A

```
x.Acquire();

y.Acquire();
```

Thread B

```
y.Acquire();

x.Acquire();
```

Thread A waits until y is available

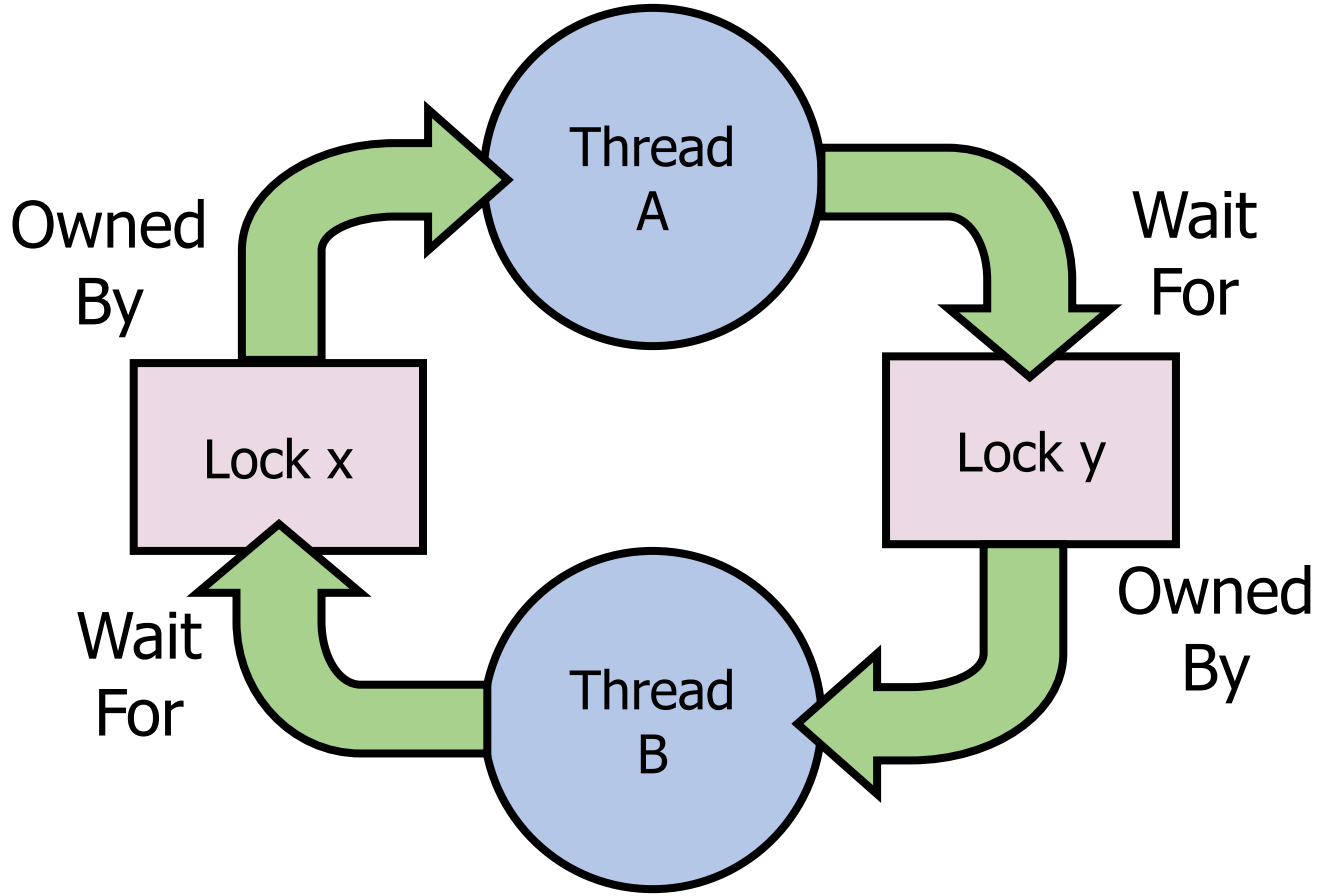
Thread B waits until x is available

--Unreachable--

```
…
y.Release();
x.Release();
```

```
…
x.Release();
y.Release();
```

Deadlocks involve *circular dependencies*



Deadlock can occur on any shared resource

- Example deadlock if the system only has 2 MB of memory

```
Thread A                  Thread B
AllocateOrWait(1 MB)      AllocateOrWait(1 MB)
AllocateOrWait(1 MB)      AllocateOrWait(1 MB)
Free(1 MB)                Free(1 MB)
Free(1 MB)                Free(1 MB)
```

- Could deadlock on access to hardware as well

Interrupts can cause deadlocks too

```
Thread A
acquire()
                Interrupt Handler
…               acquire()
```

Deadlock

```
release()
```

- Thread cannot continue until the interrupt is finished
- Interrupt cannot finish until the thread continues

Reentrant library functions

- Functions that can safely and successfully be called again while currently in the middle of its execution are called “reentrant”
  - Reentrant functions must only modify local variables and input
  - Must also never call non-reentrant functions

- `malloc()` is thread-safe because it uses locks around shared memory
  - Malloc is **NOT** reentrant and it will cause deadlock
  - Same goes for `printf`!!!
  - Must not be called in an interrupt or signal handler!
    - This matters in PCLab too

Break + Check your understanding

```
void List_Insert(list_t *L, int key) {
  pthread_mutex_lock(&L->lock);
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    pthread_mutex_unlock(&L->lock);
    return; // fail
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
  pthread_mutex_unlock(&L->lock);
  return; // success
}
```

Is it safe to call List_Insert from an interrupt?

Break + Check your understanding

```
void List_Insert(list_t *L, int key) {
  pthread_mutex_lock(&L->lock);
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    pthread_mutex_unlock(&L->lock);
    return; // fail
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
  pthread_mutex_unlock(&L->lock);
  return; // success
}
```

Not safe!

If another thread has acquired the mutex, there will be a deadlock

Outline

- Interrupts
- **Synchronization bugs**
  - **Deadlock**
    - **Solving deadlocks**
  - Livelock
  - Priority Inversion
- Threadsafe data structures
- Concurrency in other languages

How Should a System Deal With Deadlock?

- Three different approaches:

1. **Deadlock avoidance:** dynamically delay resource requests so deadlock doesn't happen

2. Deadlock prevention: write your code in a way that it isn't prone to deadlock

3. Deadlock recovery: let deadlock happen, and then figure out how to recover from it

Deadlock avoidance

- Idea: When a thread requests a resource, OS checks if it would result in an unsafe state that could lead to deadlock
  - If not, grant the resource
  - If so, wait until other threads release resources

Thread A

```
x.Acquire();


y.Acquire();
…
y.Release();
x.Release();
```

Thread B

```
y.Acquire();
x.Acquire();
…
x.Release();
y.Release();
```

Must stop acquire here to prevent unsafe state

Banker's Algorithm for avoiding deadlock

- Each thread states maximum resource needs in advance
- OS allows a particular thread to claim a resource if
  - (available resources - requested) $\geq$ maximum remaining that might be needed by any thread
- For Dining Philosophers, a request for a chopstick is allowed if:
  1. Not the last chopstick
  2. Or is the last chopstick but a philosopher will have two afterwards
- See the textbook for more details

How Should a System Deal With Deadlock?

- Three different approaches:

1. Deadlock avoidance: dynamically delay resource requests so deadlock doesn't happen

2. **Deadlock prevention:** write your code in a way that it isn't prone to deadlock

3. Deadlock recovery: let deadlock happen, and then figure out how to recover from it

Preventing Deadlocks: deadlock requires four conditions

1. **Mutual exclusion**
   - Threads cannot access a critical section simultaneously.
   - In other words, we're using locks so there is the potential for waiting.
2. **Hold-and-wait**
   - Threads do not release locks while waiting for additional locks.
3. **No preemption**
   - Locks are always held until released by the thread.
     - E.g., if there is no method to *cancel* a lock.
4. **Circular wait**
   - Thread is waiting on a thread that is waiting on the original thread.
   - This can involve just two threads or a chain of many threads.

Can eliminate deadlock by eliminating any one of these conditions

1. Do not have mutual exclusion

- Lockfree/waitfree data structures

```
void* mythread(void* arg) {
  for (int i=0; i<LOOPS; i++) {
    pthread_mutex_lock(&lock);
    counter++;
    pthread_mutex_unlock(&lock);
  }
  return NULL;
}
```

```
void* mythread(void* arg) {
  for (int i=0; i<LOOPS; i++) {
    atomic_fetch_and_add(
        &counter, 1);
  }
  return NULL;
}
```

2. Avoid hold and wait with trylock()

- We can avoid deadlock if we release the first lock after noticing that the second lock is unavailable.
- ***Trylock()*** tries to acquire a lock, but returns a failure code instead of waiting if the lock is taken:
- This code *cannot deadlock*, even if another thread does the same with L2 first, then L1.

```
1   top:
2     lock(L1);
3     if (trylock(L2) == -1) {
4       unlock(L1);
5       goto top;
6     }
```

- However it can *livelock*… we'll come back to this

3. No preemption

- The OS *could* take away the lock from a blocked thread and give it back before the thread resumes
  - This sounds pretty complicated to get right

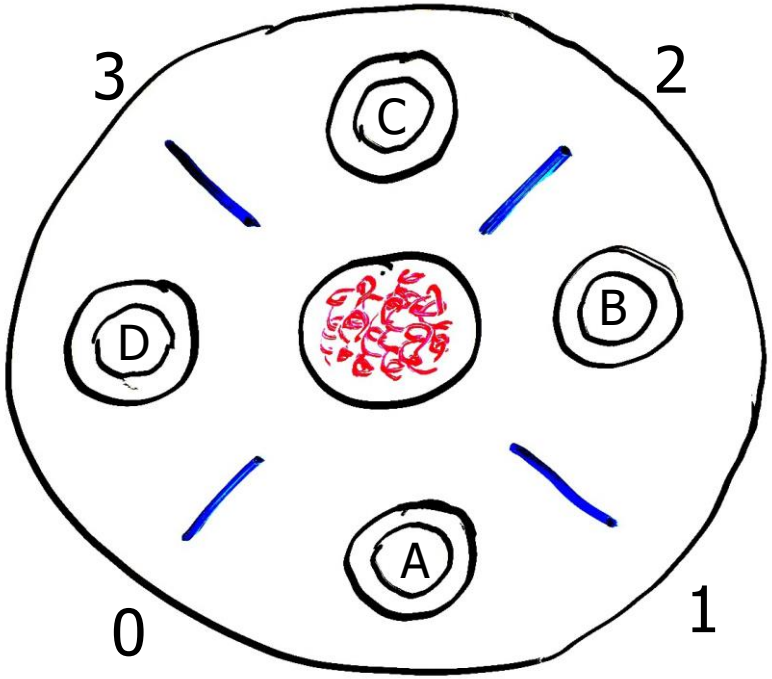
- Non-lock resources are easier here
  - Temporarily take away memory from a thread by swapping it to disk

4. Avoiding Circular Wait

- This is the most practical way to avoid deadlock.
- The simplest solution is to always acquire locks in the same order.
  - If you hold lock X and are waiting for lock Y,
  - Then holder of Y cannot be waiting on you,
  - Because they would have already acquired X before acquiring Y.

- However, in practice it can be difficult to know when locks will be acquired because they can be buried in subroutines.

Ordered locking for dining philosophers



- The chopsticks are shared resources, like locks
- If we require the **lower-numbered chopstick to be grabbed first**, this eliminates circular waiting.
  - Philosophers A, B, C grab *left then right*.
  - However philosopher D will grab *right then left*.
  - If everyone tries to start at once, A & D race to grab chopstick 0 first, and the winner eats first.
  - While one is waiting to grab its first chopstick a neighbor will be able to grab two chopsticks.

Check your understanding

- In what order must Thread B acquire the three locks to avoid deadlock?

Thread A

```
y.Acquire();
x.Acquire();
z.Acquire();
…
z.Release();
x.Release();
y.Release();
```

Thread B

```
???
```

Check your understanding

- In what order must Thread B acquire the three locks to avoid deadlock?
  - The same order!! (at least y first, for the two-thread case)

Thread A

```
y.Acquire();
x.Acquire();
z.Acquire();
…
z.Release();
x.Release();
y.Release();
```

Thread B

```
y.Acquire();
x.Acquire();
z.Acquire();
…
z.Release();
x.Release();
y.Release();
```

How Should a System Deal With Deadlock?

- Three different approaches:

1. Deadlock avoidance: dynamically delay resource requests so deadlock doesn't happen

2. Deadlock prevention: write your code in a way that it isn't prone to deadlock

3. **Deadlock recovery:** let deadlock happen, and then figure out how to recover from it

Deadlock Recovery: how to deal with a deadlock?

- Terminate thread, force it to give up resources
  - Dining Philosophers Example: Remove a dining philosopher
  - In `AllocateOrWait` example, OS kills a process to free up some memory
  - Not always possible—killing a thread holding a lock leaves world inconsistent

- Roll back actions of deadlocked threads
  - Common techniques in databases (transactions)
  - Of course, if you restart in exactly the same way, you may enter deadlock again

Modern OS approach to deadlocks

- Make sure the *system* isn't involved in any deadlock
    - Hopefully by prevention
    - Generally, be very careful about this stuff in the kernel

- Ignore deadlock in applications ("Ostrich Algorithm")
    - User can just restart them anyways

Break + Check your understanding

- Is there a possibility of deadlock?
  - If so, how could we fix it?

Thread A

```
usb.Acquire();
webcam.Acquire();
…
webcam.Release();
usb.Release();
```

Thread B

```
printer.Acquire();
usb.Acquire();
…
usb.Release();
printer.Release();
```

Thread C

```
webcam.Acquire();
printer.Acquire();
…
printer.Release();
webcam.Release();
```

Break + Check your understanding

- Is there a possibility of deadlock? **Yes**
  - If so, how could we fix it? **One solution: Global ordering of resources**
    - Example: usb, then webcams, then printers always in that order

Thread A

```
usb.Acquire();
webcam.Acquire();
…
webcam.Release();
usb.Release();
```

Thread B

```
printer.Acquire();
usb.Acquire();
usb.Acquire();
printer.Acquire();
…
usb.Release();
printer.Release();
printer.Release();
usb.Release();
```

Thread C

```
webcam.Acquire();
printer.Acquire();
…
printer.Release();
webcam.Release();
```

Break + Check your understanding

- Is there a possibility of deadlock? **Yes**
  - If so, how could we fix it? **One big lock still works too!**

Thread A
```
lock.acquire();
usb.Acquire();
webcam.Acquire();
…
webcam.Release();
usb.Release();
lock.release();
```

Thread B
```
lock.acquire();
printer.Acquire();
usb.Acquire();
…
usb.Release();
printer.Release();
lock.release();
```

Thread C
```
lock.acquire();
webcam.Acquire();
printer.Acquire();
…
printer.Release();
webcam.Release();
lock.release();
```

Outline

- Interrupts
- **Synchronization bugs**
  - Deadlock
    - Solving deadlocks
  - **Livelock**
  - Priority Inversion
- Threadsafe data structures
- Concurrency in other languages

Common synchronization bugs

- Atomicity violation
  - An operation that should have been atomic wasn't
- Order violation
  - Something happens sooner (or later) than expected
- Deadlock
  - Two threads wait indefinitely on each other
- Livelock (not that common in practice)
  - Two threads repeatedly block each other from proceeding and retry

Livelock while avoiding deadlock

```
// thread 1
getLocks12(lock1, lock2) {
  lock1.acquire();
  while (lock2.locked()) {
    // attempt to step aside
    // for the other thread
    lock1.release();
    wait();
    lock1.acquire();
  }
  lock2.acquire();
}
```

```
// thread 2
getLocks21(lock1, lock2) {
  lock2.acquire();
  while (lock1.locked()) {
    // attempt to step aside
    // for the other thread
    lock2.release();
    wait();
    lock2.acquire();
  }
  lock1.acquire();
}
```

Avoiding hold and wait could lead to livelock

- Avoiding hold and wait can *livelock*
  - Two threads *could* get stuck in this loop forever
  - Unlikely to occur for any length in personal computing setting
  - Very possibly stuck forever (or at least extended periods) in a constrained computing setting
    - Example: embedded system with known tasks at the start

```
1   top:
2     lock(L1);
3     if (trylock(L2) == -1) {
4       unlock(L1);
5       goto top;
6     }
```

Livelock in agents

- Livelock is more common in agent-based programs
  - All of agent's options lead to a lack of forward progress

- One example: video games
  - The character can still move and take actions
  - But cannot complete the level

Livelock versus Deadlock

```
1    top:
2      lock(L1);
3      if (trylock(L2) == -1) {
4        unlock(L1);
5        goto top;
6      }
```

- Livelock is a condition where two threads repeatedly take action, but still don't make progress.
- Differs from deadlock because deadlock is always permanent.
- Livelock involves retries that ***may*** lead to progress, but there is no ***guarantee*** of progress.
  - A malicious scheduler can always keep the livelock stuck
- Any randomness in the timing of retries will fix livelock.
- In practice, livelock is a much less serious concern than deadlock.

Outline

- Interrupts
- **Synchronization bugs**
  - Deadlock
    - Solving deadlocks
  - Livelock
  - **Priority Inversion**
- Threadsafe data structures
- Concurrency in other languages

Systems interact with each other

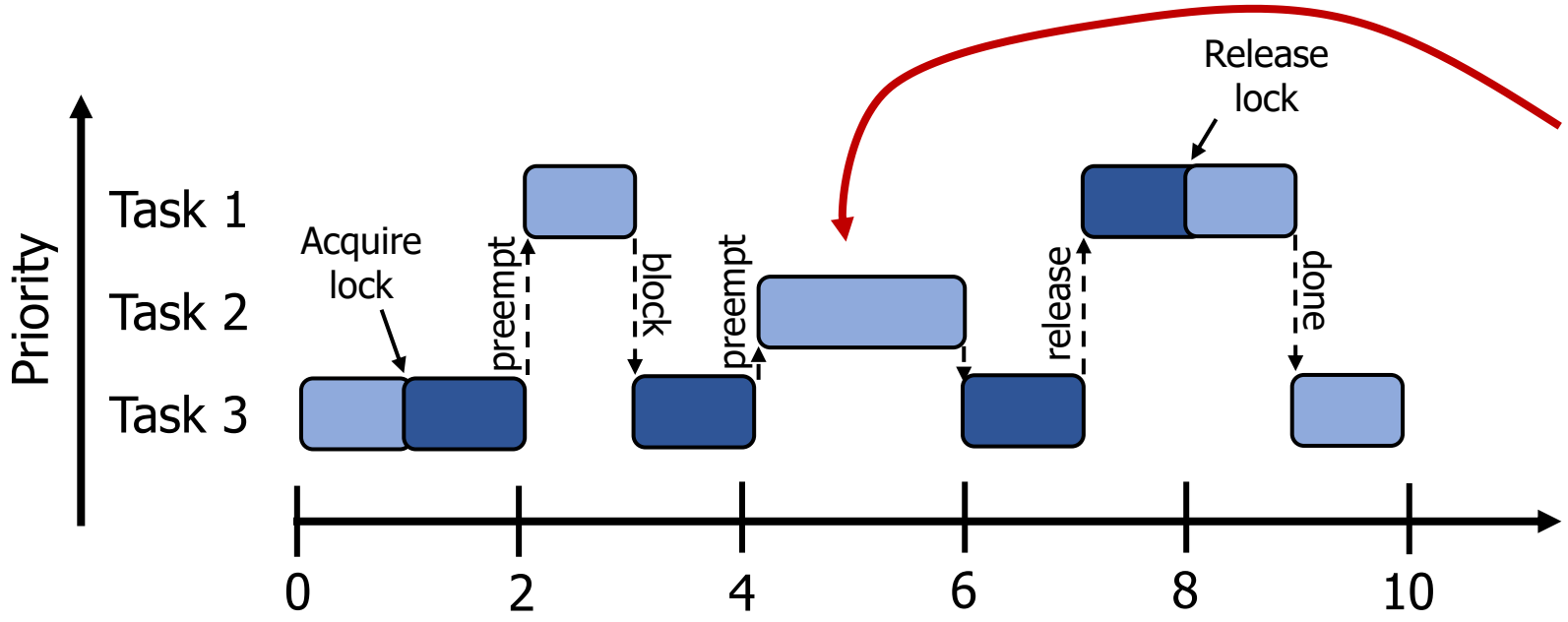
- Scheduling and Concurrency problems are not exclusive

- Sharing mutexes between threads can lead to a big problem for schedulers based on priority
  - Especially dangerous for real-time OS scenarios

A problem with priority schedulers: priority inversion

- Other concepts from OS still apply when we're scheduling
  - Particularly locks and synchronization

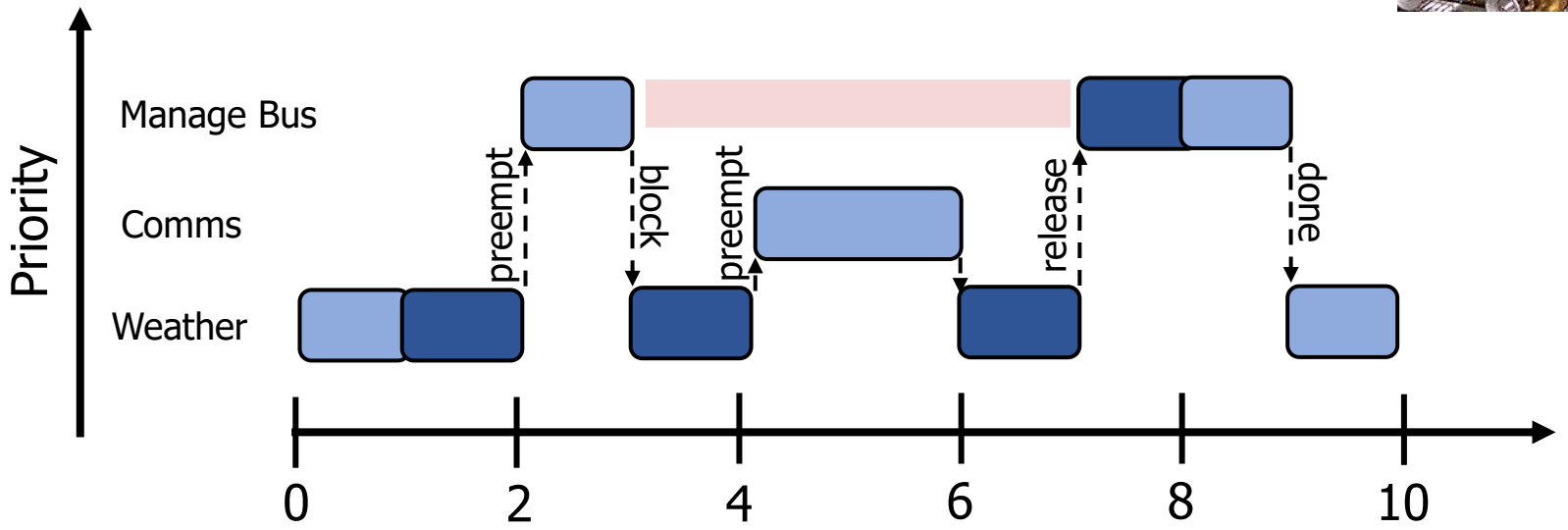
- Imagine Task 1 and Task 3 both need to share a lock



Task 1 is waiting on Task 2!!

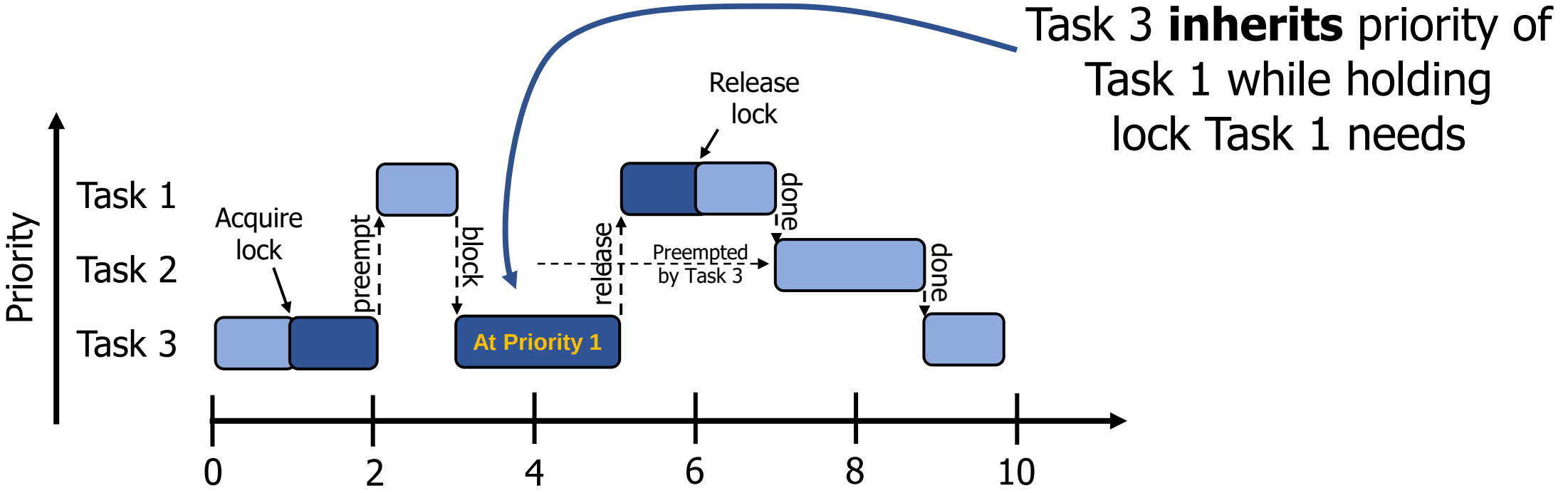
Priority inversion occurred on Pathfinder!

- Bus management missed deadlines while waiting on meteorology because medium-priority tasks were taking too long
  - System rebooted when deadline was missed



Priority inheritance solution to priority inversion

- A solution is to temporarily increase priority for tasks holding resources that high priority tasks need



Break + Tools

- Helgrind (part of the Valgrind tool) detects many common errors when using the POSIX pthreads library
  - Bad library calls: unlocking an unlocked mutex, destroying a locked mutex
  - Deadlocks and Data races
  - http://valgrind.org/docs/manual/hg-manual.html

- ThreadSanitizer (in the family of Address Sanitizer) is compiler instrumentation that detects data races
  - 5-15x slowdown for running code
  - https://clang.llvm.org/docs/ThreadSanitizer.html

Outline

- Interrupts
- Synchronization bugs
  - Deadlock
    - Solving deadlocks
  - Livelock
  - Priority Inversion
- **Threadsafe data structures**
- Concurrency in other languages

Thread-safe data structures

- "Thread safe" – works even if used by multiple threads concurrently
  - Can apply to various libraries, functions, and data structures

- Simple data structures implementations are usually not thread safe
  - Some global state needs to be shared among all threads
  - Need to protect critical sections

- Challenge: multiple function calls each access same shared structure
  - Need to identify the critical section in each and lock it with shared lock

Linked List

```
void List_Insert(list_t *L, int key) {
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    return; // fail
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
  return; // success
}
```

Concurrent Linked List – Big lock approach

```
void List_Insert(list_t *L, int key) {
  pthread_mutex_lock(&L->lock);
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    pthread_mutex_unlock(&L->lock);
    return; // fail
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
  pthread_mutex_unlock(&L->lock);
  return; // success
}
```

Most important part of this example. Don't forget to unlock if returning early.

- Much better than counter example, because we are only serializing the list itself. Hopefully the rest of the code can run concurrently.

Better Concurrent Linked List – Only lock critical section

```
void List_Insert(list_t *L, int key) {
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    return; // fail
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
  return; // success
}
```

Check your understanding:

Where is the critical section here?

Better Concurrent Linked List – Only lock critical section

```
void List_Insert(list_t *L, int key) {
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    return; // fail
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
  return; // success
}
```

Check your understanding:

Where is the critical section here?

What about malloc? Is that safe to use??

```
void List_Insert(list_t *L, int key) {
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    return; // fail
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
  return; // success
}
```

- Thread-safe functions
  - Capable of being called concurrently and still functioning correctly
  - (Because they use locks!)
- How would we know if malloc is thread-safe?
  - Must check the documentation

Must check the library documentation to determine thread safety

- https://man7.org/linux/man-pages/man3/malloc.3.html

- Malloc (and free) is indeed thread-safe

ATTRIBUTES top

For an explanation of the terms used in this section, see attributes(7).

Interface	Attribute	Value
malloc(), **free**(), **calloc**(), **realloc**()	Thread safety	MT-Safe

- If it wasn't, we would have to consider it another shared resource that needs to be locked

Better Concurrent Linked List – Only lock critical section

```
void List_Insert(list_t *L, int key) {
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    return; // fail
  }
  new->key = key;
  pthread_mutex_lock(&L->lock);
  new->next = L->head;
  L->head = new;
  pthread_mutex_unlock(&L->lock);
  return; // success
}
```

- Now new node is created locally in parallel
- Only actual access to the linked list is serialized

Concurrent Queue

- Separate head & tail locks
- Allows concurrent add & remove
  - Up to 2 threads can access without waiting

```
1   typedef struct __node_t {
2       int                 value;
3       struct __node_t    *next;
4   } node_t;
5
6   typedef struct __queue_t {
7       node_t             *head;
8       node_t             *tail;
9       pthread_mutex_t     headLock;
10      pthread_mutex_t     tailLock;
11  } queue_t;
12
13  void Queue_Init(queue_t *q) {
14      node_t *tmp = malloc(sizeof(node_t));
15      tmp->next = NULL;
16      q->head = q->tail = tmp;
17      pthread_mutex_init(&q->headLock, NULL);
18      pthread_mutex_init(&q->tailLock, NULL);
19  }
```

```
21  void Queue_Enqueue(queue_t *q, int value) {
22      node_t *tmp = malloc(sizeof(node_t));
23      assert(tmp != NULL);
24      tmp->value = value;
25      tmp->next  = NULL;
26
27      pthread_mutex_lock(&q->tailLock);
28      q->tail->next = tmp;
29      q->tail = tmp;
30      pthread_mutex_unlock(&q->tailLock);
31  }
32
33  int Queue_Dequeue(queue_t *q, int *value) {
34      pthread_mutex_lock(&q->headLock);
35      node_t *tmp = q->head;
36      node_t *newHead = tmp->next;
37      if (newHead == NULL) {
38          pthread_mutex_unlock(&q->headLock);
39          return -1; // queue was empty
40      }
41      *value = newHead->value;
42      q->head = newHead;
43      pthread_mutex_unlock(&q->headLock);
44      free(tmp);
45      return 0;
46  }
```

Concurrent Queue

- "tailLock" controls adding elements
- Looks similar to ListInsert

```
1   typedef struct __node_t {
2       int                 value;
3       struct __node_t    *next;
4   } node_t;
5
6   typedef struct __queue_t {
7       node_t             *head;
8       node_t             *tail;
9       pthread_mutex_t     headLock;
10      pthread_mutex_t     tailLock;
11  } queue_t;
12
13  void Queue_Init(queue_t *q) {
14      node_t *tmp = malloc(sizeof(node_t));
15      tmp->next = NULL;
16      q->head = q->tail = tmp;
17      pthread_mutex_init(&q->headLock, NULL);
18      pthread_mutex_init(&q->tailLock, NULL);
19  }
```

```
21  void Queue_Enqueue(queue_t *q, int value) {
22      node_t *tmp = malloc(sizeof(node_t));
23      assert(tmp != NULL);
24      tmp->value = value;
25      tmp->next  = NULL;
26
27      pthread_mutex_lock(&q->tailLock);
28      q->tail->next = tmp;
29      q->tail = tmp;
30      pthread_mutex_unlock(&q->tailLock);
31  }
32
33  int Queue_Dequeue(queue_t *q, int *value) {
34      pthread_mutex_lock(&q->headLock);
35      node_t *tmp = q->head;
36      node_t *newHead = tmp->next;
37      if (newHead == NULL) {
38          pthread_mutex_unlock(&q->headLock);
39          return -1; // queue was empty
40      }
41      *value = newHead->value;
42      q->head = newHead;
43      pthread_mutex_unlock(&q->headLock);
44      free(tmp);
45      return 0;
46  }
```

Concurrent Queue

- Head lock controls removing elements from front
- Needs to lock almost entire function

```
1   typedef struct __node_t {
2       int                 value;
3       struct __node_t    *next;
4   } node_t;
5
6   typedef struct __queue_t {
7       node_t             *head;
8       node_t             *tail;
9       pthread_mutex_t     headLock;
10      pthread_mutex_t     tailLock;
11  } queue_t;
12
13  void Queue_Init(queue_t *q) {
14      node_t *tmp = malloc(sizeof(node_t));
15      tmp->next = NULL;
16      q->head = q->tail = tmp;
17      pthread_mutex_init(&q->headLock, NULL);
18      pthread_mutex_init(&q->tailLock, NULL);
19  }
```

```
21  void Queue_Enqueue(queue_t *q, int value) {
22      node_t *tmp = malloc(sizeof(node_t));
23      assert(tmp != NULL);
24      tmp->value = value;
25      tmp->next  = NULL;
26
27      pthread_mutex_lock(&q->tailLock);
28      q->tail->next = tmp;
29      q->tail = tmp;
30      pthread_mutex_unlock(&q->tailLock);
31  }
32
33  int Queue_Dequeue(queue_t *q, int *value) {
34      pthread_mutex_lock(&q->headLock);
35      node_t *tmp = q->head;
36      node_t *newHead = tmp->next;
37      if (newHead == NULL) {
38          pthread_mutex_unlock(&q->headLock);
39          return -1; // queue was empty
40      }
41      *value = newHead->value;
42      q->head = newHead;
43      pthread_mutex_unlock(&q->headLock);
44      free(tmp);
45      return 0;
46  }
```

Concurrent Hash Table

- Each bucket is implemented with a Concurrent List
  - We don't have to define any locks!
  - (Locks are in the lists)
- A thread can access a bucket without blocking other threads' access to ***other*** buckets.
- Hash tables are great for concurrency.
  - Hash (bucket id) can be calculated without accessing a shared resource.
  - ***Distributed hash tables*** are used for huge NoSQL databases.

```
1   #define BUCKETS (101)
2
3   typedef struct __hash_t {
4       list_t lists[BUCKETS];
5   } hash_t;
6
7   void Hash_Init(hash_t *H) {
8       int i;
9       for (i = 0; i < BUCKETS; i++) {
10          List_Init(&H->lists[i]);
11      }
12  }
13
14  int Hash_Insert(hash_t *H, int key) {
15      int bucket = key % BUCKETS;
16      return List_Insert(&H->lists[bucket], key);
17  }
18
19  int Hash_Lookup(hash_t *H, int key) {
20      int bucket = key % BUCKETS;
21      return List_Lookup(&H->lists[bucket], key);
22  }
```

Lock-free data structures

- In our original example, we put a lock around `counter++`
  - We could have instead used `atomic_fetch_and_add` to update counter
  - Lock-free and *still* atomic!!

- This is possible with more complex data structures as well
  - Often based on a compare-and-swap (CAS) approach
  - https://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf

- Warning: these are not to be taken lightly
  - Atomic instructions have performance costs on processors
  - Getting this correct involves really understanding hardware
  - https://abseil.io/docs/cpp/atomic_danger

Outline

- Interrupts
- Synchronization bugs
  - Deadlock
    - Solving deadlocks
  - Livelock
  - Priority Inversion
- Threadsafe data structures
- **Concurrency in other languages**

Javascript

- Javascript (in browsers) is strictly single-threaded
  - Therefore, no data races!

- A Javascript function will never be interrupted unless it makes an asynchronous call

```
console.log("1");
setTimeout(function(){console.log("2");},0);
console.log("3");
setTimeout(function(){console.log("4");},1000);
```

  - Will always output: **1 3 2 4** in that order
    - Even timers only trigger whenever the current code is finished

Python

- Provides all the same primitives we discussed! https://docs.python.org/3/library/concurrency.html

`threading` — Thread-based parallelism
- Thread-Local Data
- Thread Objects
- Lock Objects
- RLock Objects
- Condition Objects
- Semaphore Objects
  - `Semaphore` Example
- Event Objects
- Timer Objects
- Barrier Objects

And some nicer things

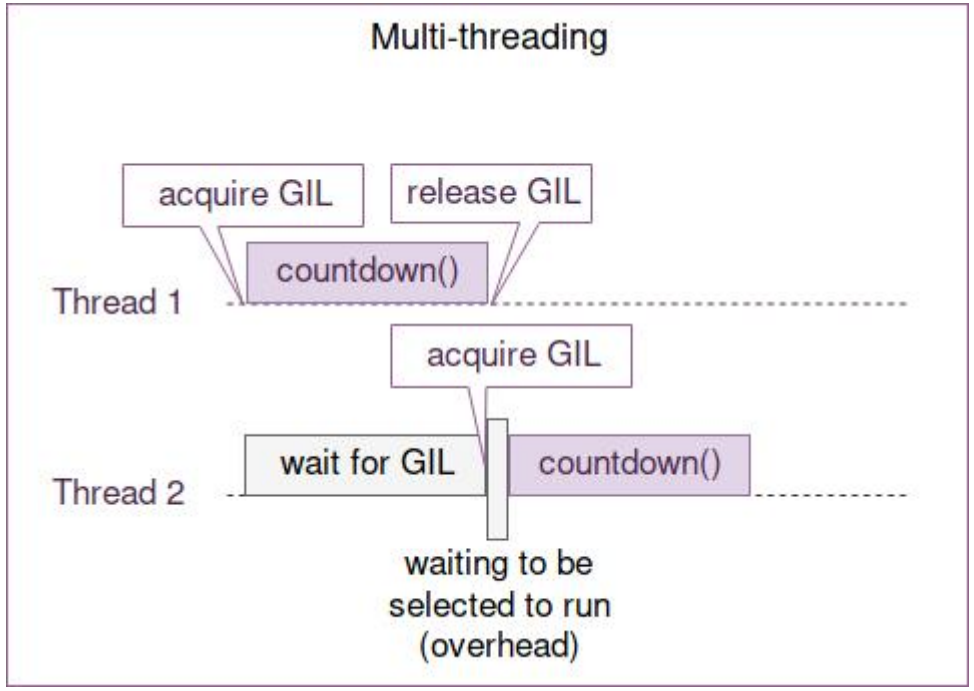
```
with some_lock:
  # do something…
```

Is equivalent to

```
some_lock.acquire()
try:
  # do something…
finally:
  some_lock.release()
```

Python threads are concurrent but not parallel

- Python uses one big lock technique for thread safety
  - Global Interpreter Lock (GIL)
  - Threads that are I/O bound still get a performance boost
  - Threads that are CPU bound do not increase performance



- *Multiprocessing* library does employ parallelism by spawning entirely new processes
  - Each with their own python interpreter

https://hackernoon.com/concurrent-programming-in-python-is-not-what-you-think-it-is-b6439c3f3e6a

Active work in changing this: https://peps.python.org/pep-0703/

Java

- Java has ***synchronized*** keyword for surrounding critical sections
- Automatically releases the lock when exiting early:
- Similar to
  - Python: "`with self.lock:`"
  - Objective-C: "`@synchronized`"

```
public class Counter {
    int mTotal = 0;

    public synchronized void addOne() {
        int val = mTotal;
        val++;
        mTotal = val;
    }

    public void addOneVersion2() throws Exception {
        int val;
        synchronized(this) {
            val = mTotal;
            val++;
            if (val == Integer.MAX_VALUE) {
                throw new Exception("value is too large");
            }
            mTotal = val;
        }
        System.out.println("new value is " + val);
    }
}
```

Rust

- Rust's opinion on sharing memory is amusingly to refer to Go's opinion

> *Do not communicate by sharing memory; instead, share memory by communicating.*
>
> *--Effective Go*

- Rust has a strong concept of ownership
  - A writeable (mutable) reference to an object can only be held in one place
  - Once an object is passed to another thread, the passer no longer has access
  - Solves many concurrency issues due to lack of shared memory

- Rust locks have lifetimes enforced by the compiler
  - Lock goes out-of-scope at the end of the function, relocking automatically

https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html

Advice for the future

- Be aware of issues when writing multithreaded code

- Use threadsafe data structures when possible
  - In languages that provide them…

- Map your problem onto a classical concurrency problem
  - Producer/Consumer
  - Readers/Writers

- One big lock for *correctness* isn't the worst idea ever
  - But with some care (possibly a lot of care) we can do better

Outline

- Interrupts
- Synchronization bugs
  - Deadlock
    - Solving deadlocks
  - Livelock
  - Priority Inversion
- Threadsafe data structures
- Concurrency in other languages