Lecture 06: Synchronization Bugs

CS343 – Operating Systems

Branden Ghena – Fall 2024

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

Northwestern

Next week Tuesday: online recording

- Unfortunately, I’ll be out-of-town on Tuesday next week
- So, no in-person class on that day
- I will record the lecture in advance and put it out on Panopto
  - Lecture: Classical Scheduling

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

- **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

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

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

Should have been included in critical section

Check your understanding: atomicity violation

- What's wrong here?
  - Every access is locked, right?
- 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

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

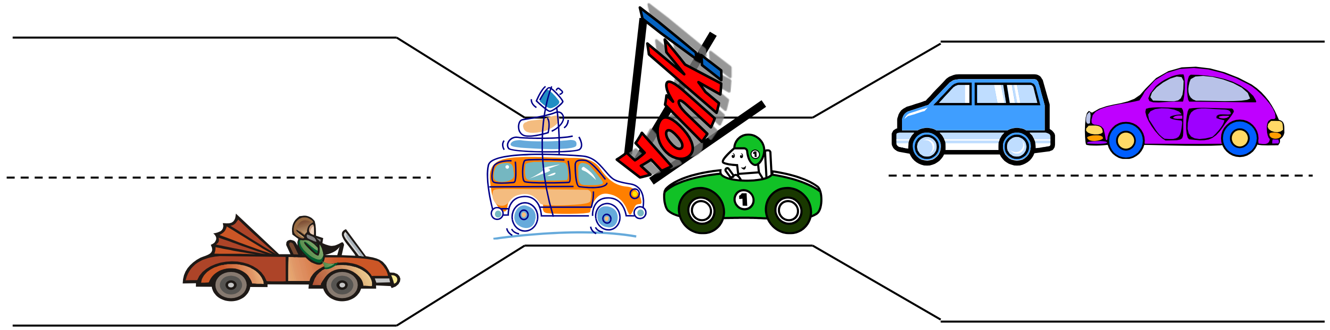

BANCO REAL
11455

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

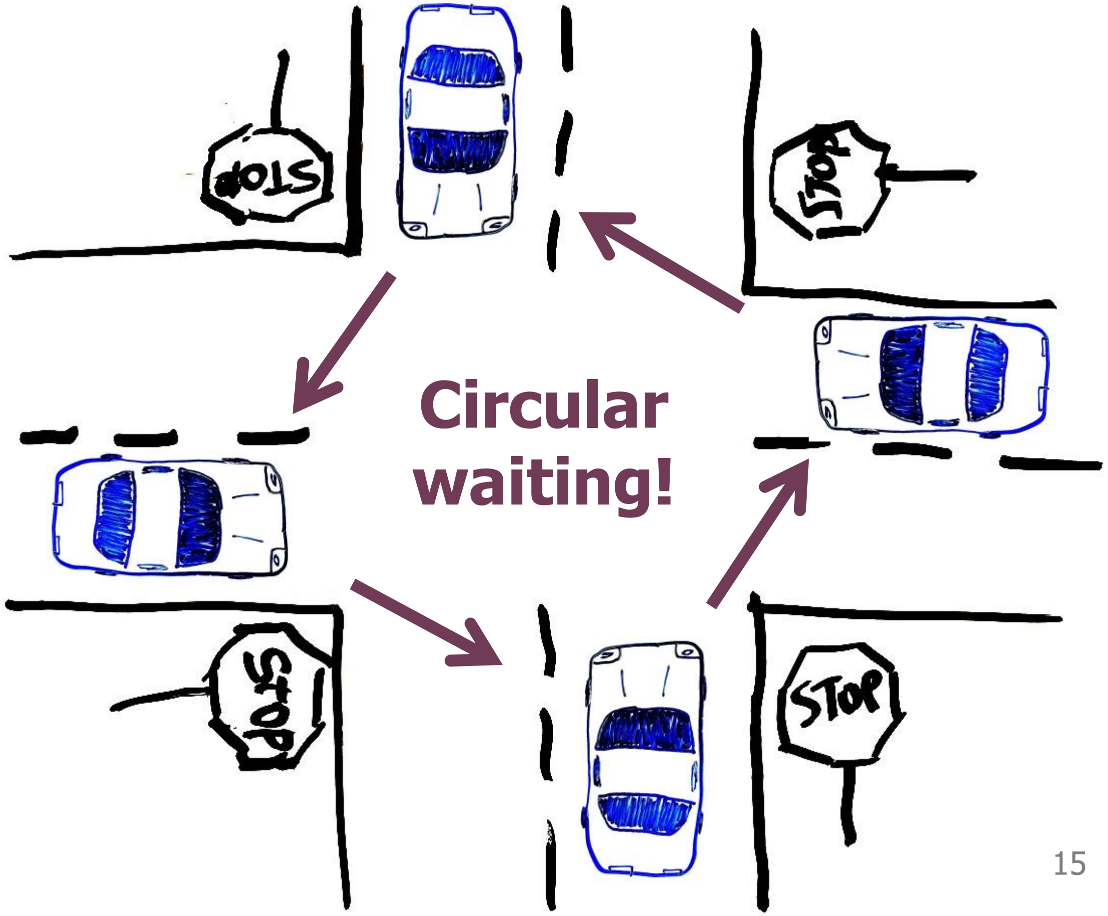


- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into

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

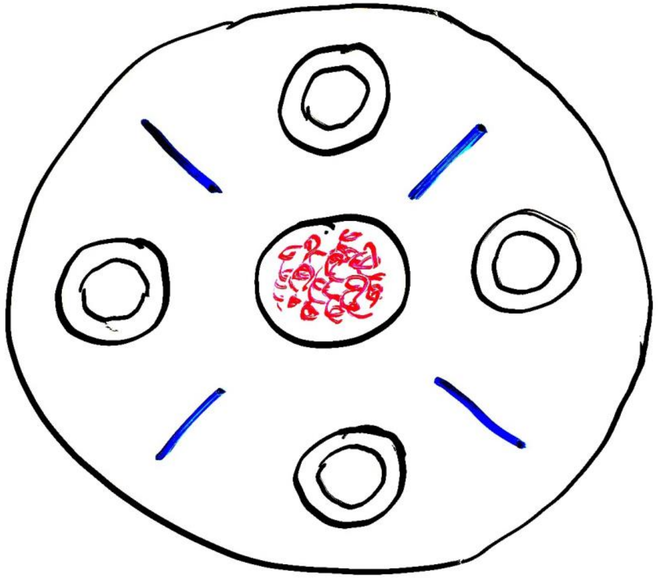
Simple example: four-way stop

- 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

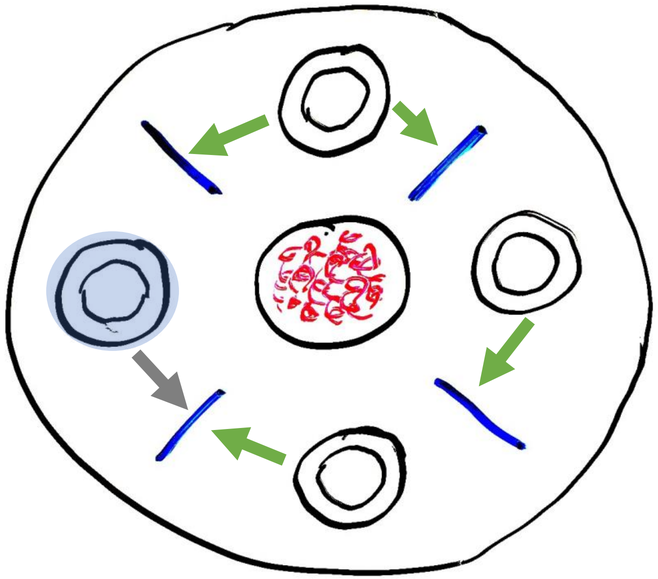
- A theoretical example of deadlock
- There are N philosophers sitting in a circle and N chopsticks
  - left and right of each philosopher
- Philosophers repeatedly run this loop:
  1. Think for some time
  2. Grab chopstick to left
  3. Grab chopstick to right
  4. Eat
  5. Replace chopsticks



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

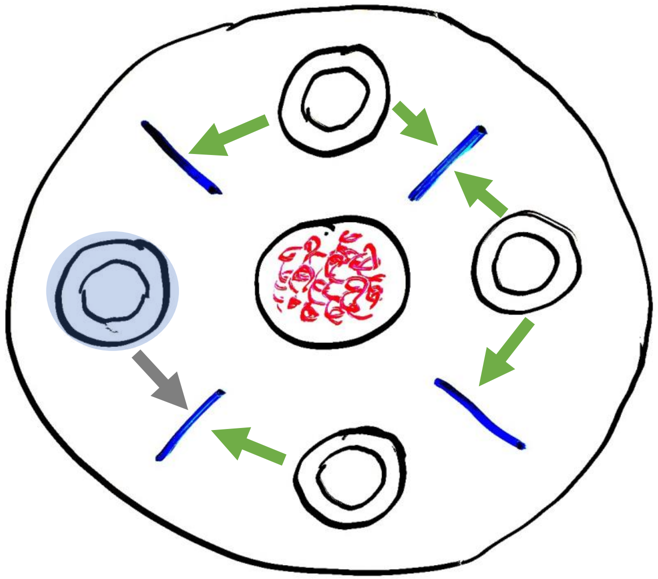
Dining philosophers

- A solution: one philosopher must grab right before left



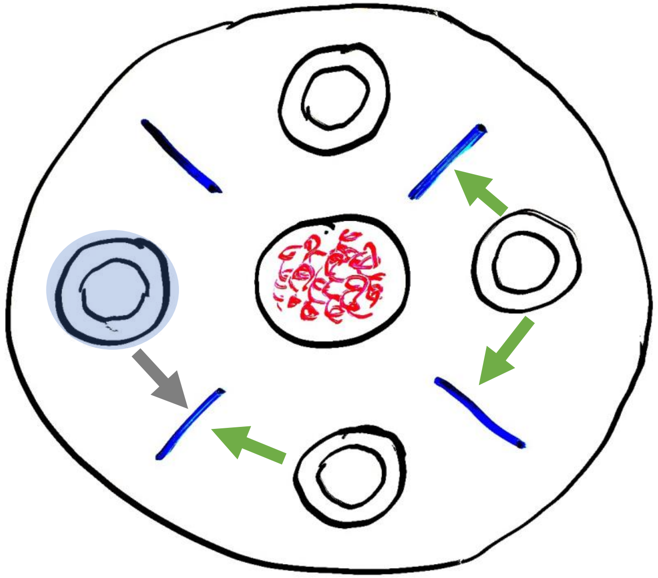
Dining philosophers

- 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();




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

Thread B waits until Thread A is finished

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--

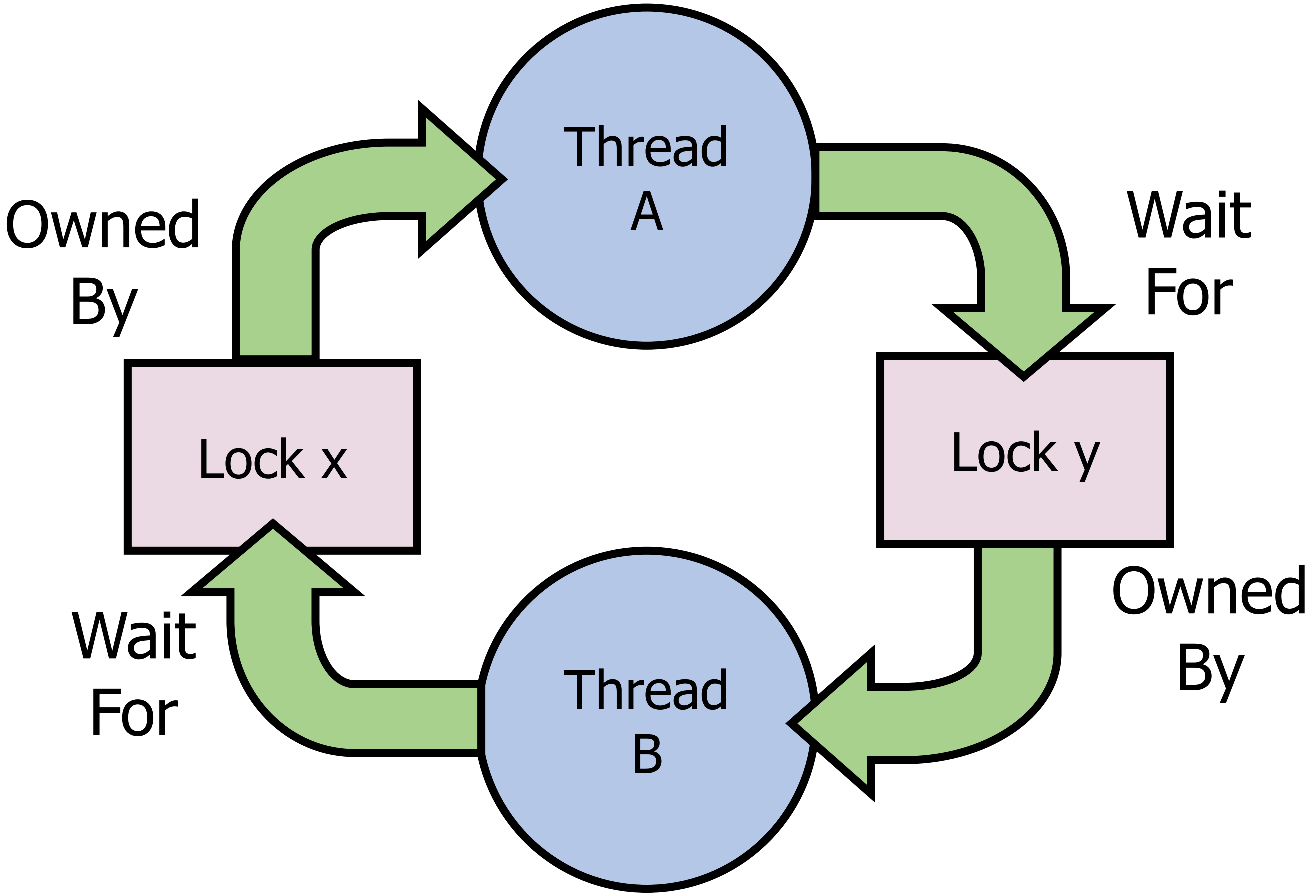
Thread A:

```
…
y.Release();
x.Release();
```

Thread B:

```
…
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

- **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	Thread B
`x.Acquire();`	
	`y.Acquire();`
	`x.Acquire();`
`y.Acquire();`	…
…	`x.Release();`
`y.Release();`	`y.Release();`
`x.Release();`	

Must stop acquire here to prevent unsafe state (arrow pointing to Thread B's `y.Acquire();`)

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:

```
top:
  lock(L1);
  if (trylock(L2) == -1) {
    unlock(L1);
    goto top;
  }
```

- This code *cannot deadlock*, even if another thread does the same with L2 first, then L1.
- However it can *livelock*… we'll come back to this

3. No preemption

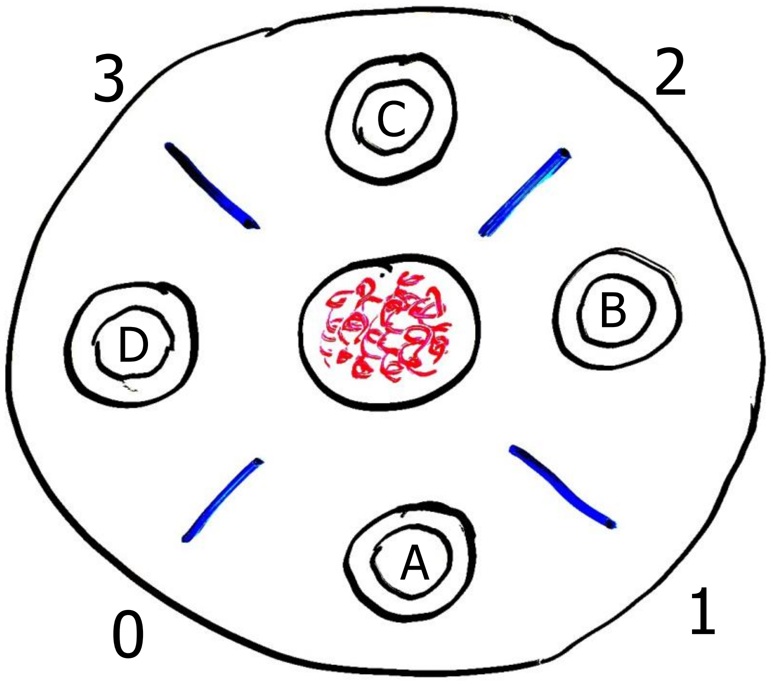
- The OS *could* take away the lock from a blocked thread and give it back before the thread resumes
  - This sounds pretty complicated to get right
- Non-lock resources are easier here
  - Temporarily take away memory from a thread by swapping it to disk

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	**Thread B**
`y.Acquire();`	`y.Acquire();`
`x.Acquire();`	`x.Acquire();`
`z.Acquire();`	`z.Acquire();`
…	…
`z.Release();`	`z.Release();`
`x.Release();`	`x.Release();`
`y.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	**Thread B**	**Thread C**
`usb.Acquire();`	`printer.Acquire();`	`webcam.Acquire();`
`webcam.Acquire();`	`usb.Acquire();`	`printer.Acquire();`
…	…	…
`webcam.Release();`	`usb.Release();`	`printer.Release();`
`usb.Release();`	`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

- **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

```
top:
  lock(L1);
  if (trylock(L2) == -1) {
    unlock(L1);
    goto top;
  }
```

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

```
top:
  lock(L1);
  if (trylock(L2) == -1) {
    unlock(L1);
    goto top;
  }
```

- 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

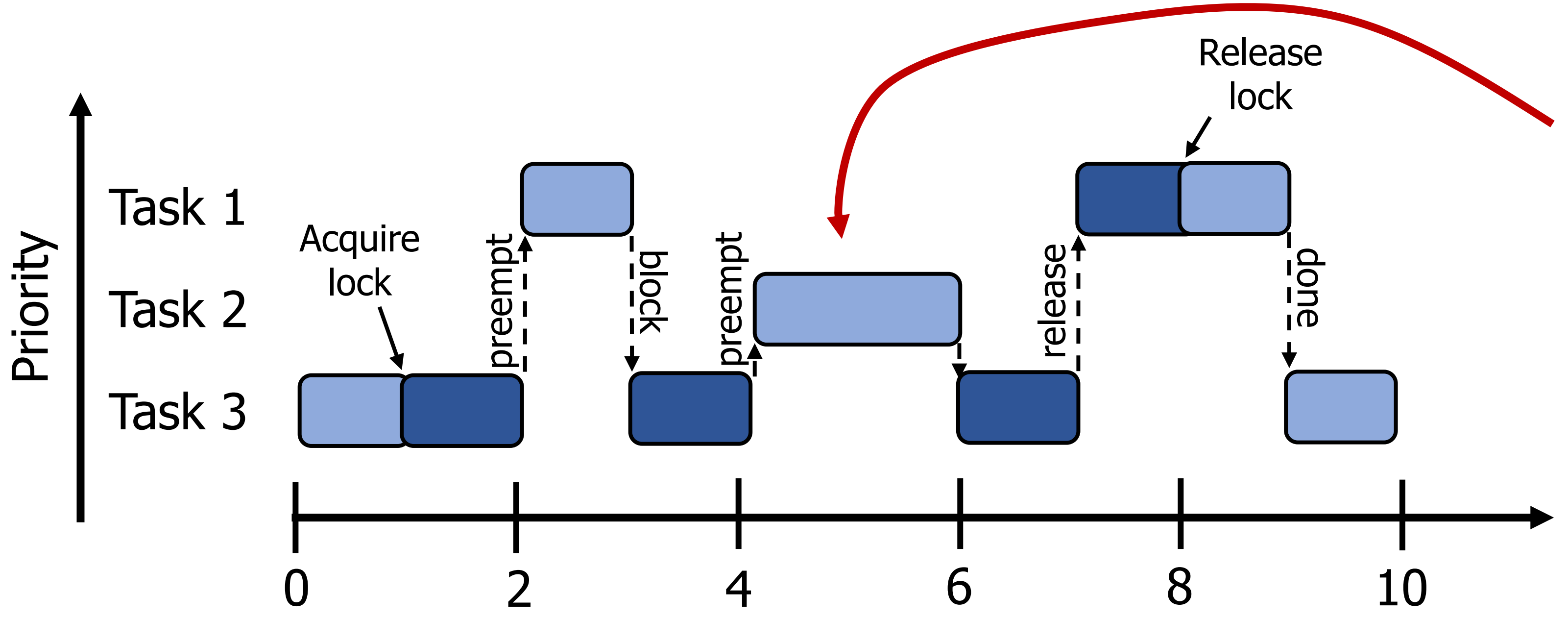
- Scheduling and Concurrency problems are not exclusive

- 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

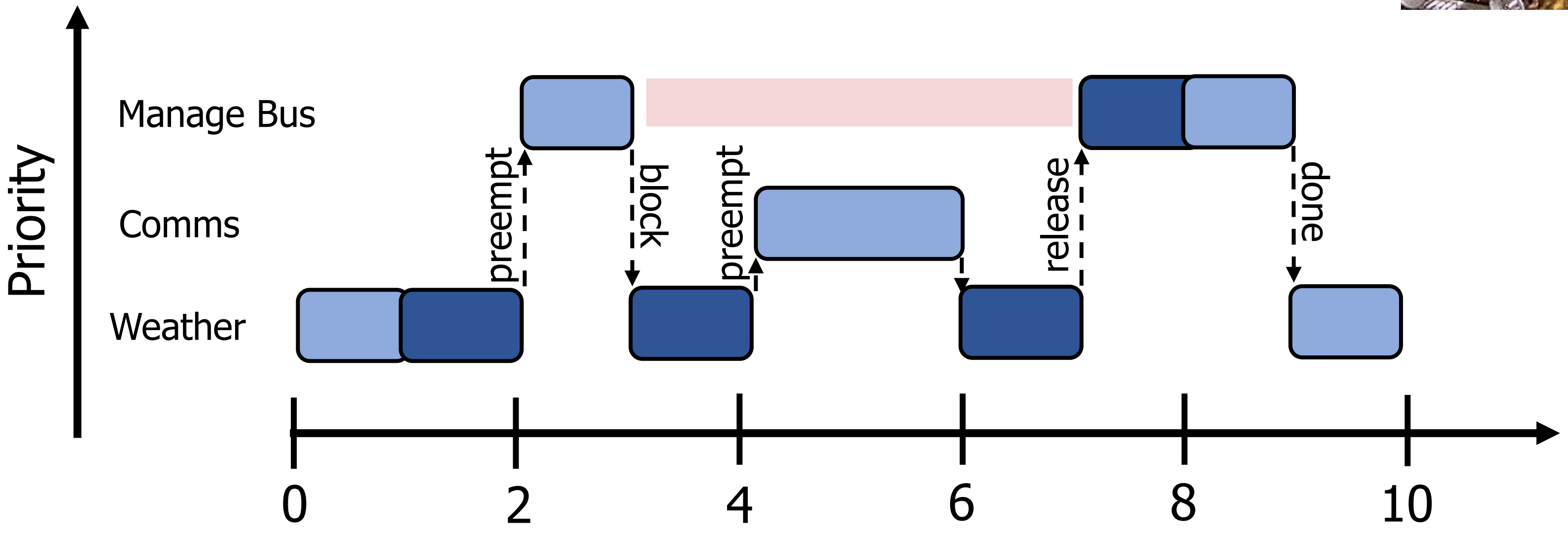
- Imagine Task 1 and Task 3 both need to share a lock



Task 1 is waiting on Task 2!!

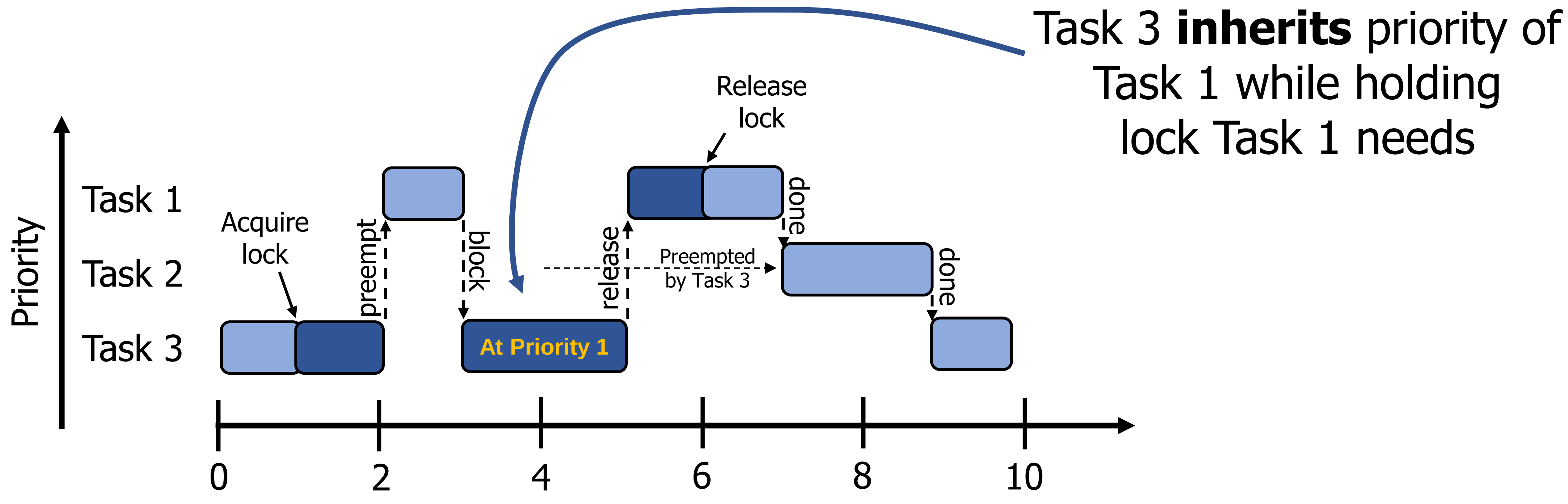
Priority inversion occurred on Pathfinder!

- 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

```c
typedef struct __node_t {
    int              value;
    struct __node_t *next;
} node_t;

typedef struct __queue_t {
    node_t          *head;
    node_t          *tail;
    pthread_mutex_t  headLock;
    pthread_mutex_t  tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->headLock, NULL);
    pthread_mutex_init(&q->tailLock, NULL);
}
```

```c
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;

    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}

int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->headLock);
        return -1; // queue was empty
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->headLock);
    free(tmp);
    return 0;
}
```

Concurrent Queue

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

```
typedef struct __node_t {
    int                 value;
    struct __node_t    *next;
} node_t;

typedef struct __queue_t {
    node_t             *head;
    node_t             *tail;
    pthread_mutex_t     headLock;
    pthread_mutex_t     tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->headLock, NULL);
    pthread_mutex_init(&q->tailLock, NULL);
}
```

```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;

    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}

int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->headLock);
        return -1; // queue was empty
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->headLock);
    free(tmp);
    return 0;
}
```

Concurrent Queue

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

```
typedef struct __node_t {
    int                 value;
    struct __node_t    *next;
} node_t;

typedef struct __queue_t {
    node_t             *head;
    node_t             *tail;
    pthread_mutex_t     headLock;
    pthread_mutex_t     tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->headLock, NULL);
    pthread_mutex_init(&q->tailLock, NULL);
}
```

```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;

    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}

int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->headLock);
        return -1; // queue was empty
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->headLock);
    free(tmp);
    return 0;
}
```

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.

```
#define BUCKETS (101)

typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
    int i;
    for (i = 0; i < BUCKETS; i++) {
        List_Init(&H->lists[i]);
    }
}

int Hash_Insert(hash_t *H, int key) {
    int bucket = key % BUCKETS;
    return List_Insert(&H->lists[bucket], key);
}

int Hash_Lookup(hash_t *H, int key) {
    int bucket = key % BUCKETS;
    return List_Lookup(&H->lists[bucket], key);
}
```

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

Break + Question: Where is the critical section for vector?

```
typedef struct {
  size_t size;
  size_t count;
  int** data;
} vector_t;


void vector_add(vector_t* v, int* item) {
  if (v->count == v->size) {
    v->size *= 2;
    v->data = realloc(v->data, sizeof(int*)*v->size);
  }
  v->data[v->count++] = item;
}
```

Break + Question: Where is the critical section for vector?

```
typedef struct {
  size_t size;
  size_t count;
  int** data;
} vector_t;


void vector_add(vector_t* v, int* item) {
  if (v->count == v->size) {
    v->size *= 2;
    v->data = realloc(v->data, sizeof(int*)*v->size);
  }
  v->data[v->count++] = item;
}
```

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

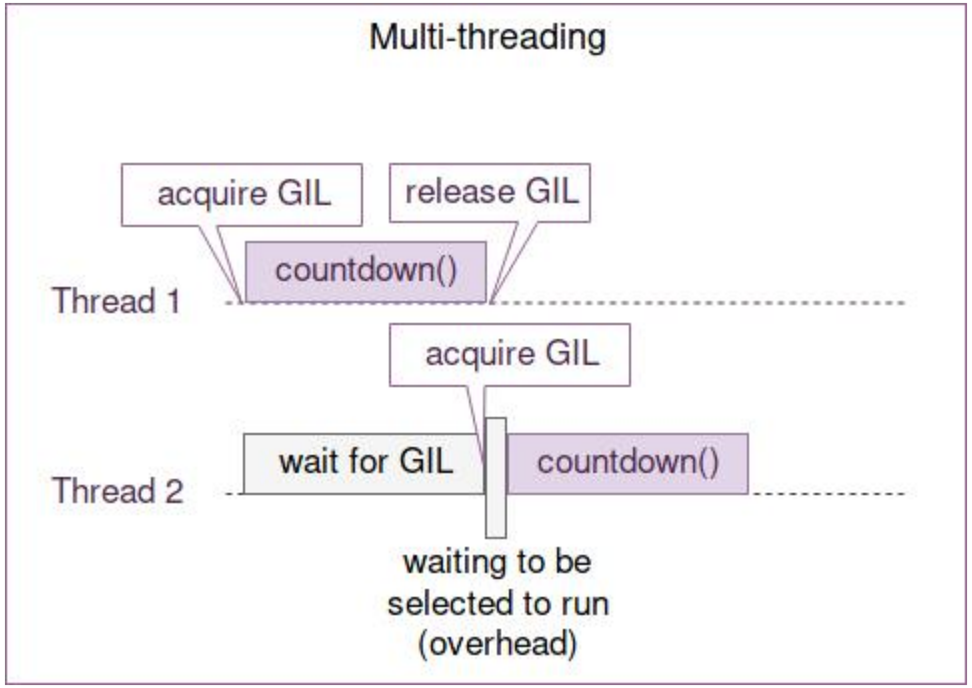
```
with some_lock:
  # do something…
```

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/

Modern C++ has standard synchronization primitives

- Condition variable takes a function as an argument for determining the condition (since C++11)

```
void waits() {
   std::unique_lock<std::mutex> lk(cv_m);
   std::cerr << "Waiting... \n";
   cv.wait(lk, []{ return i == 1; });
   std::cerr << "...finished waiting. i == 1\n";
}
```

- Semaphores were added in C++20. Two forms:
  - Counting semaphores (arbitrary integer value)
  - Binary semaphores (internal value of 1 or 0)

Modern C++ Locking

- `scoped_lock` object is constructed with one or more `mutex` objects as a local variable (since C++17)
  - When the scoped_lock is constructed, it acquires all mutexes
    - And does so with a global ordering to avoid deadlocks
  - When the scoped_lock is destructed it releases the mutexes
    - Usually happens automatically at the end of a function

```
// Each Employee has its own mutex to protect updates to itself
void assign_partner(Employee& e1, Employee& e2) {
  std::scoped_lock lock(e1.mutex, e2.mutex); // deadlock avoided!
  e1.partners.push_back(e2.id);
  e2.partners.push_back(e1.id);

  // mutexes are automatically released when the function ends
}
```

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