Week 14 – Friday

COMP 3400

Last time

- What did we talk about last time?
- Review up to Exam 2
  - Networking
    - Application
    - Transport
    - Internet
    - Link
    - Physical
  - Socket programming
  - Peer-to-peer networks
  - HTTP
  - TCP vs. UDP
  - Network security
    - CIA
    - Symmetric and public key cryptography
    - Cryptographic hash functions

Questions?

Assignment 8

Review

Final exam format

- **Final exam will be in this room:**
  - **Wednesday, April 30, 2025**
  - **8:00 – 10:00 a.m.**
  - **50% longer than previous exams, but you have 100% more time**
- Mostly short answer questions
- One or two matching questions
- A couple of debugging questions
- A couple of programming questions

Threading

Threads and processes

- Many processes can run concurrently
  - Each one executes independently
  - Each process has its own memory layout
- Many threads can also run concurrently
  - Each one executes independently
  - Each thread has its own stack to keep track of its function calls
  - But all threads within a process **share** code, data, heap, and kernel segments
- Just as we used **`fork()`** to spawn new processes, there are libraries to spawn new threads within a process and coordinate them

Advantages of threads

- Using threads allows for more modular software since threads can call the same functions within a program
- Threads can be more efficient since there's no context switch needed for different threads to interact
- Some models of programming like GUIs depend on threads so that one unit of code needs can react to an action taken elsewhere
- Since threads share memory, there's no need for IPC libraries

Disadvantages of threads

- Threads are less isolated from each other than separate processes
- Consequences:
  - A thread crashing from a segmentation fault will kill the entire process, including the other threads
  - Bugs called **race conditions** occur, where the behavior of the program is different depending on which thread executed first

Race conditions

- Race conditions are a central problem with threads
- Thread scheduling is non-deterministic
  - It's often impossible to predict when the statements from one thread are going to be executed with respect to those in another thread
  - If the statements modify the same memory, the results can be inconsistent
- One of the most frustrating issues with race conditions is that they can occur rarely
  - This means that you can run your program 1,000 times with no problems, only to crash badly on time 1,001

Critical sections

- A **critical section** is a series of statements that *must* be executed atomically to get the right result
- **Atomic** execution means that all the statements happen as if they happened at once, without other statements from other threads interfering
- Even statements that look atomic like **`i++`** are actually several different operations in assembly language

```
movq   _globalvar(%rip), %rsi     # copy from memory into %rsi register
addq   $1, %rsi                   # increment the value in the register
movq   %rsi, _globalvar(%rip)     # store the result back into memory
```

Incrementing variables

- Consider two threads that share an **`int`** variable called **`global`** that is initially set to **`0`**:

Thread A

```
for (int i = 0; i < 200; ++i)
     ++global;
```

Thread B

```
for (int j = 0; j < 300; ++j)
     ++global;
```

- What are the largest and smallest values that **`global`** could have after these threads run to completion?

Thread safety

- Many functions are **thread safe**, meaning that they can be called by many threads at the same time and still give the right answers
- Other functions are not thread safe
  - Examples: **`rand()`** and **`strtok()`**
- The usual reason that functions are not thread safe is because they contain static local variables
- Because these variables are shared by all threads, they can become corrupted

POSIX Threads

POSIX threads

- Just as we could create a new process with **`fork()`**, there are libraries for making new threads
- POSIX threads (also called pthreads) are perhaps the most widely used thread library
  - Windows (of course) has its own threading library, though people have built POSIX-like libraries on top of it
- Key POSIX concepts
  - Creating a thread starts it running
  - A thread can exit, stopping its running
  - Joining a thread means waiting for a thread to finish (and potentially getting its result)
  - We keep track of processes with an ID of type **`pid_t`**, but we keep track of threads with an ID of type **`pthread_t`**

POSIX thread functions

- Here are POSIX functions mapping to concepts from the previous slide

```
int pthread_create (pthread_t *thread, const pthread_attr_t *attr,
      void *(*start_routine)(void*), void *arg);
```

  - Create a new thread (not as bad as it looks)

```
void pthread_exit (void *value_ptr);
```

  - Exit from the current thread (giving a pointer to the result, if any)

```
void pthread_join (pthread_t thread, void *value_ptr);
```

  - Join a thread (getting a pointer to its result, if any)

Creating a thread

- Creating a thread is the most complicated function, partly because it takes a function pointer and potentially arguments

```
int pthread_create (pthread_t *thread, const pthread_attr_t *attr,
      void *(*start_routine)(void*), void *arg);
```

  - **`thread`** is a pointer to a **`pthread_t`** that will get filled in with the thread's ID
  - **`attr`** is a pointer to possible thread attributes (often left **`NULL`**)
  - **`start_routine`** is a pointer to a function that takes a **`void*`** and returns a **`void*`**
  - **`arg`** is a pointer to arguments, **`NULL`** if no arguments needed

Simple threading example

```c
#include <stdio.h>
#include <pthread.h>        // POSIX thread library
#include <assert.h>

void *
start_thread (void *args)  // Function to start thread with
{
  printf ("Hello from thread!\n");
  pthread_exit (NULL);
}

int
main (int argc, char **argv)
{
  pthread_t child_thread;

  // Create new thread with function start_thread
  assert (pthread_create (&child_thread, NULL, start_thread, NULL) == 0);

  pthread_join (child_thread, NULL);  // Wait for other thread to finish
  pthread_exit (NULL);                // main() exits like any other thread
}
```

Common mistakes

- Passing in a garbage **`pthread_t*`** instead of the address of a real **`pthread_t`**

```
pthread_t *thread; // No!
```

- Calling the threading function (with parentheses) instead of passing a function pointer in

```
pthread_create (thread, NULL, start (), NULL); // No!
```

- Joining with a **`pthread_t*`** instead of a **`pthread_t`**

```
pthread_join (thread, NULL); // No!
```

Passing arguments

- Passing arguments to threads is tricky
  - Passing addresses to objects on the stack is dangerous in case the function creating the threads returns
  - Passing pointers to the same object to multiple threads can cause problems if they fight over it
  - There are no timing guarantees over which thread will run when

A useful hack

- On most modern machines, a pointer is either 32 bits or 64 bits
- An **`int`** is usually 32 bits
- We can cast an **`int`** to a pointer and pass that to the thread
- The thread will then cast the pointer back to an **`int`**
- Since the size of an **`int`** is almost always less than a pointer, we don't lose any information
- It's icky, but it allows us to pass simple values like a **`char`**, **`short`**, or **`int`**
  - Both floating-point types are harder since they have to be tricked into behaving like integers (which pointers fundamentally are)
  - And **`double`** is risky since it needs a 64-bit pointer to hold it all

A thread function that uses a pointer like an `int`

```
void * child_thread (void *args)
{
  int value = (int) args; // Now, I pretend it's an int!
  printf ("I'm a thread with value: %d\n", value);
  pthread_exit (NULL);
}

int main (int argc, char **argv)
{
  pthread_t threads[10];  // Array to hold thread IDs

  // Start up those threads, pretending ints are pointers
  for (int i = 0; i < 10; i++)
    pthread_create (&threads[i], NULL, child_thread, (void*)i);

  for (int i = 0; i < 10; i++)
    pthread_join(threads[i], NULL);
  pthread_exit (NULL);
}
```

Passing multiple arguments to a thread

- To pass multiple arguments, they're often grouped in a struct
- Remember that threads all have their own stacks
- Thus, we need to pass in a struct that has been dynamically allocated on the heap (which is shared)
  - Also, any pointers that struct contains should point at memory that isn't on the stack

Multiple argument example

```
struct thread_args
{
  int value;
  const char* string;
};

int main (int argc, char **argv)
{
  pthread_t thread;
  struct thread_args* args = malloc(sizeof(struct thread_args));
  args->value = 42;
  args->string = "wombat";

  // Thread casts void* to struct thread_args* when it gets it
  pthread_create (&thread, NULL, child_thread, args);

  pthread_join(thread, NULL);
  pthread_exit (NULL);
}
```

Returning values from threads

- A common model for threads is for them to go and perform some work
- After the work is done, they need to give back the answer
- There are three ways to do this:
  1. Store the answer back into the dynamically allocated struct passed in for its arguments
  2. Use the hack like before to return a "pointer" through the join that's actually an **`int`**
  3. Return a pointer through the join to a dynamically allocated struct containing the answer

Synchronization

Synchronization

- Now you have all the tools needed to create, run, and join threads
- But you don't have any tools to avoid the problem of race conditions
- **Synchronization** is used to coordinate between threads, often by enforcing critical sections, sections of code that only one thread can be executing at a time
- Common synchronization tools:
  - Locks (mutexes)
  - Semaphores
  - Barriers
  - Condition variables
- If used incorrectly, however, synchronization tools can lead to other problems such as deadlock and livelock

Examples of synchronization

- The following are common examples of synchronization:
  - Multiple threads share a data structure, but only one can write to it at a time
  - Only so many threads can access a shared resource to avoid slowdowns
  - Certain events need to happen in a certain order
  - Some calculations must be done before an action can be taken
- Performing synchronization so that the result is correct while avoiding performance penalties is challenging

Critical sections

- Recall that a critical section is a section of code that it's safe for only a single thread to be executing
- Often this is because non-atomic memory accesses (such as reading a value, doing calculations, and then writing back to memory) can get inconsistent results if more than one thread is executing them concurrently
- A common use of synchronization tools is to block threads trying to access a critical section if a thread is already executing it

Locks

- A key synchronization tool is called a **lock** (or a **mutex**, short for *mutual exclusion*)
- Critical sections can be protected by a lock
  - First code acquires the lock
  - Then it performs the code in the critical section
  - Then it releases the lock
- For POSIX threads, lock functionality is provided by several mutex functions that operate on **`pthread_mutex_t`** objects

Lock features

- Mutual exclusion
  - Locks start unlocked
  - Only one thread can acquire a lock at a time
  - No other thread can acquire a lock until it's been released
- Non-preemption
  - A lock must be voluntarily released by the thread that acquired it
- Atomic operations
  - Acquire and release are atomic operations
- Blocking acquires
  - If a thread tries to acquire a lock, it's blocked and added to the queue
  - When the thread holding the lock releases it, only one thread acquires it

POSIX mutex functions

```
int pthread_mutex_init (pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
```

- Create a mutex with the specified attributes

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

- Destroy an existing mutex

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

- Acquire a mutex, blocking until you succeed

```
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

- Try to acquire a mutex, returning non-zero if another thread has the mutex

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

- Release the mutex

How long should critical sections be?

- Now that you have locks that you can use to protect a critical section, how should you use them?
- In general, you want critical sections to be short so that one thread won't block another unnecessarily
- Nevertheless, breaking up one section of code into several critical sections will introduce penalties because acquiring and releasing locks isn't free

Semaphores

- We mentioned semaphores in the context of synchronizing processes that shared memory
- We can use semaphores to synchronize threads as well
- Recall that we think of a semaphore as a non-negative integer that can be incremented and decrementing atomically
  - Calling **`sem_wait()`** (decrement) on a semaphore at **`0`** will block until another thread calls **`sem_post()`** (increment)

Semaphore functions

```
sem_t *sem_open (const char *name, int oflag,
/* mode_t mode, unsigned int value */ );
```

- Return (and possibly create) a named semaphore, using the usual **`oflag`** and **`mode`** flags
- **`value`** determines the initial value of the semaphore (often 0)

```
int sem_wait (sem_t *sem);
```

- Block if the semaphore's value is 0, decrement after continuing

```
int sem_post (sem_t *sem);
```

- Increment the semaphore's value, unblocking a process if the value is 0

```
int sem_close (sem_t *sem);
```

- Close a semaphore

```
int sem_unlink (const char *name);
```

- Delete a semaphore

Semaphores for signaling

- We can use semaphores to signal some event to another thread
- As in our earlier examples with semaphores, we initialize the semaphore to **`0`**
  - The thread waiting for the event will call **`sem_wait()`** on the semaphore
  - The thread signaling that the event has happened will call **`sem_post()`**
  - The waiting thread will be awoken when the signaling thread posts
  - If the signaling thread posts before the waiting starts waiting, it won't have to wait

Mutual exclusion with semaphores

- It should be unsurprising that we can use semaphores instead of locks (POSIX mutexes)
- To do so, we initialize the semaphore to a value of **1**
  - When entering a critical section, a thread waits on (downs) the semaphore
  - When leaving a critical section, the thread posts on (ups) the semaphore
- The first thread reaching the critical section is allowed in because the value is **1**
- If we had initialized to **0**, no threads could enter the critical section

Semaphores as multiplexing

- Semaphores can also be used for multiplexing, in which a maximum number of threads are allowed to access a resource
- Consider a club where the bouncer only lets 100 people in
- This kind of synchronization is used less than signaling and mutexes, but it can be useful to prevent slowdown from too many threads using a resource
- Also, it can be used to prevent possible race conditions when there's a fixed number of items but the threads themselves have to select the one they want
  - No more than the maximum number of threads will be allowed to do selection

Semaphore summary

- Semaphores are a flexible tool that can be used for signaling, mutual exclusion, and multiplexing
- The key is the initial value of the semaphore
  - **0** for signaling
  - **1** for mutual exclusion
  - Greater than **1** for multiplexing
- Conceptually, the initial value of the semaphore is the maximum number of concurrent accesses

Barriers

- Sometimes a bunch of threads are working on a task that has phases
- We want to guarantee that all threads have finished Phase 1 before moving on to Phase 2
- To guarantee this, we can use **barriers**
- A barrier prevents threads from continuing unless ***k*** threads have reached it
  - It's common for ***k*** to be the total number of threads
  - Sometimes, however, the calculation is fine as long as at least ***k*** are done
- It's possible to do this kind of coordination with semaphores, but it's hard to get it exactly right

Barrier functions

```
int pthread_barrier_init (pthread_barrier_t *barrier, const
  pthread_barrierattr_t *attr, unsigned count);
```

- Create a barrier with the attributes given (often NULL) and the count of threads blocked

```
int pthread_barrier_destroy (pthread_barrier_t *barrier);
```

- Free up the resources associated with a barrier

```
int pthread_barrier_wait (pthread_barrier_t *barrier);
```

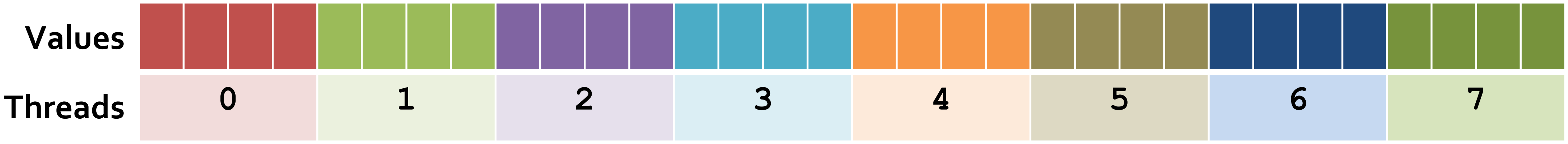
- Wait on a barrier until enough threads reach it

Merge sort

- We can imagine a threaded merge sort that works in this way:
  - Each thread is assigned a section of the array to sort
  - Each thread uses merge sort to sort that part of the array
  - All threads wait on a barrier
- Then
  - Even numbered threads merge together their section with the neighboring section
  - Threads that are multiples of four merge together double sections with other double sections
  - Threads that are multiples of eight merge together quadruple sections with other quadruple sections
  - …

Threaded merge sort visualized

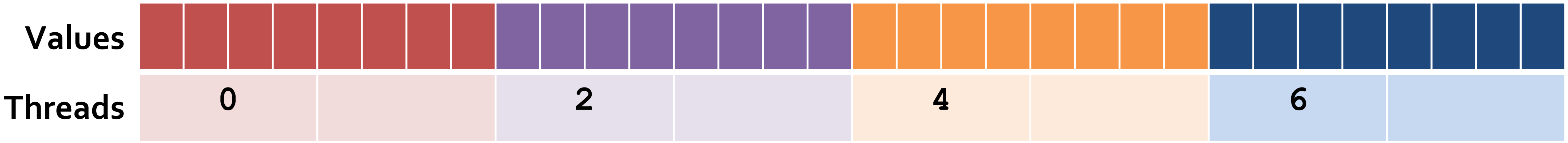
- Each thread is assigned a section of an array and sorts it



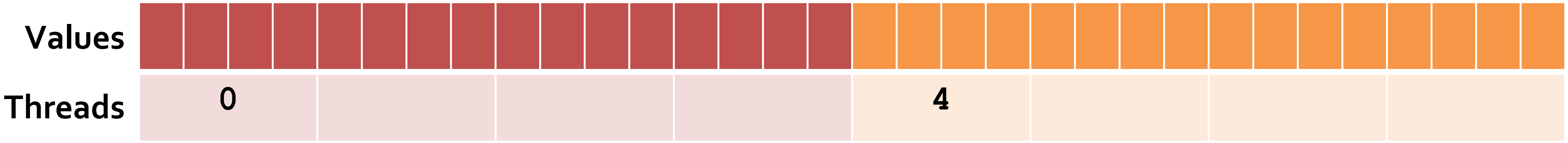
- Since there's no overlap, each thread can work independently
- After sorting, all threads wait on a barrier to be sure that every thread has finished sorting

Final merging visualized

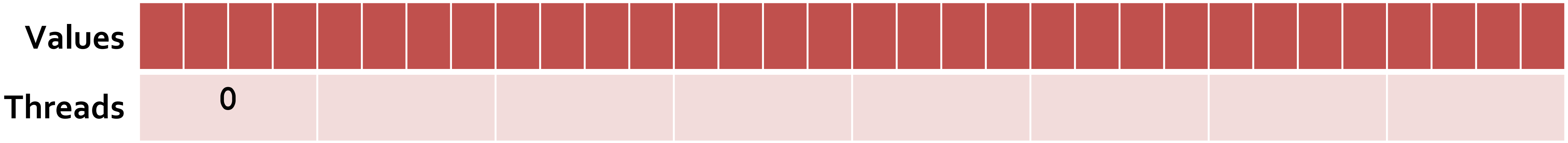
- Threads can't merge the same parts of the array without causing race conditions
- Half the threads merge with their neighbors



- Then, half of those merge



- And so on, until it's all merged



Weaknesses of semaphores

- Semaphores are very general purpose concurrency tool, but they have some weaknesses:
  - Semaphores take thought to use correctly: Incrementing and decrementing values don't map clearly to synchronization problems
  - Different implementations of semaphores have different features
  - Some systems (like macOS) don't have a full implementation of semaphores
  - Semaphores can only signal to one thread: no broadcasting
  - After getting a signal, threads have to take another step (like acquiring a lock) to get mutually exclusive access, time that can allow a race condition

Condition variables

- **Condition variables** try to overcome some weaknesses of semaphores by tying themselves directly to a lock
- They also have the ability to broadcast, waking up all waiting threads
- Like semaphores, they still have a function to wait and a function to signal
- However, something sneaky happens with wait:
  - First, the thread must acquire a lock
  - Then, it calls the wait function
  - If it has to wait, it releases the lock but then reacquires it when it gets woken up
  - All of which happens atomically
- This allows a thread to safely check a condition and wait until it gets signaled
- Think of a condition variable as a queue for waiting threads

Deadlock

Deadlock

- In order to avoid race conditions, we introduced several synchronization tools:
  - Locks (mutexes)
  - Semaphores
  - Barriers
  - Condition variables
- Each of these can be misused, failing to avoid race conditions
- Likewise, each introduces overhead, slowing the system down
- But an even worse possibility is **deadlock**

Deadlock

- Deadlock occurs when the use of synchronization primitives cause threads to get stuck so that they will never make progress again
  - A lock that never gets unlocked
  - A semaphore that never gets posted on
  - A barrier that is never reached by enough threads
  - A condition variable that is never signaled on
- Like many concurrency problems, deadlock can occur rarely or it can happen every time a program runs

Deadlock example

- In the following code, deadlock is possible

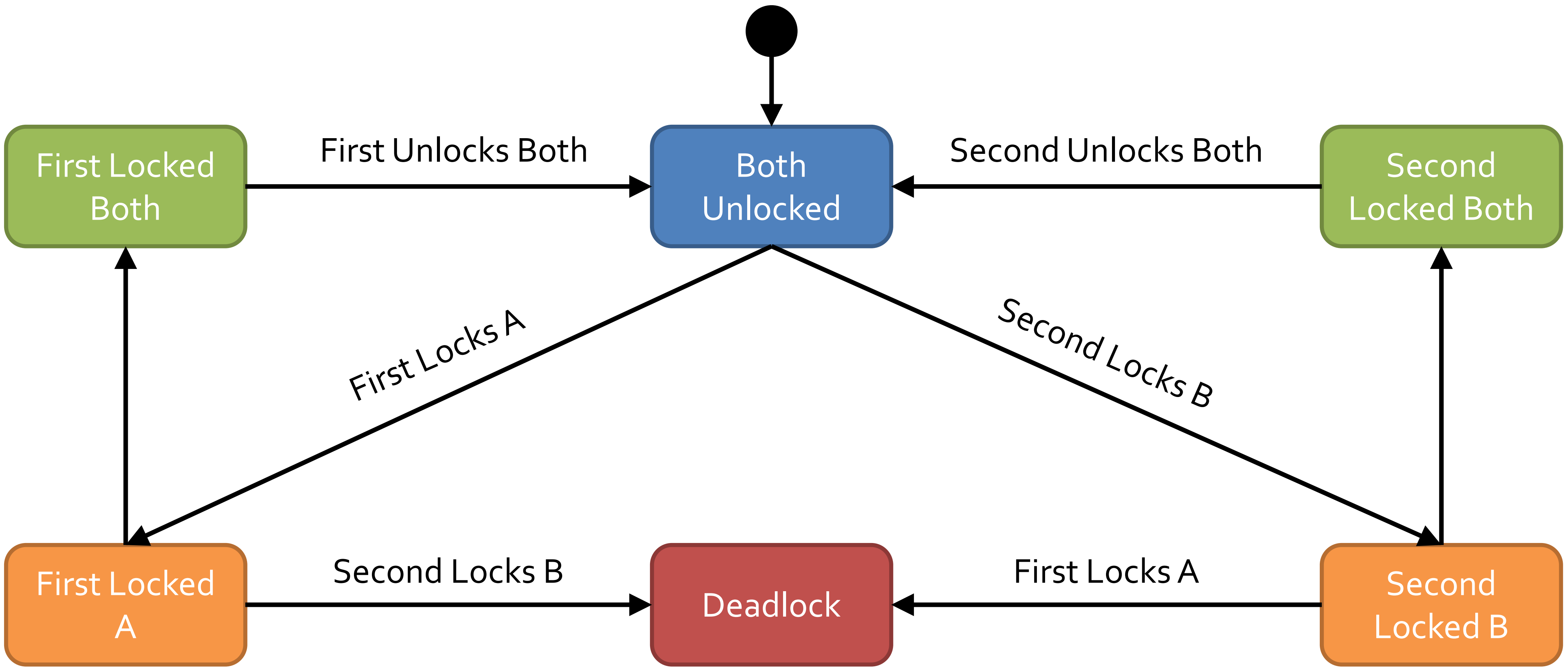
```
struct args {
  pthread_mutex_t lock_a;
  pthread_mutex_t lock_b;
};

void * first (void * args)
{
  struct args *data = (struct args *) args;
  pthread_mutex_lock (&data->lock_a); // Lock A
  pthread_mutex_lock (&data->lock_b); // Then lock B
  // More code (that would eventually unlock A and B)
}

void * second (void * args)
{
  struct args *data = (struct args *) args;
  pthread_mutex_lock (&data->lock_b); // Lock B
  pthread_mutex_lock (&data->lock_a); // Then lock A
  // More code (that would eventually unlock A and B)
}
```

Possible states

- The following state diagram shows the states the threads can be in:



Why does this happen?

- The two threads try to acquire locks in different orders:
  - First tries to get lock A followed by lock B
  - Second tries to get lock B followed by lock A
- If they tried to get the locks in the same order, we would never have this problem
- Even so, real situations are more complex
- Threads might need to acquire a number of locks for a number of resources
- The order might be hard to predict ahead of time

Necessary conditions

- Four conditions are needed for deadlock to be possible:
  1. **Mutual exclusion:** Once a resource has been acquired, no other thread gets it
  2. **No preemption:** Threads can't be made to give up their resources
  3. **Hold and wait:** Threads can get one resource and keep it while trying to get others
  4. **Circular wait:** Thread A needs a resource held by Thread B, and Thread B needs a resource held by Thread A (or extend to a chain of threads)
- Conditions 1 through 3 are unavoidable, so solutions often focus on avoiding circular wait

Livelock

- **Livelock** is similar to deadlock
- It's a condition where, due to bad timing, processes continue executing code, but they never make progress beyond a certain point
  - They're acquiring resources, giving them up, and acquiring them again, but still blocking each other
- If the system is set up in a certain way or is very unlucky, livelock could continue indefinitely
- Livelock can also sometimes resolve

Avoiding deadlock

- As mentioned before, we usually concentrate on the circular wait condition of deadlock:
  - Order the resources and always acquire them in the same order
  - Use timed or non-blocking versions of functions that acquire resources, potentially causing livelock
  - Limit the number of threads that can access the resources, insuring that there's always enough resources to go around

Synchronization Design Patterns

Signaling

- **Signaling** is a design pattern we've already discussed
  - One thread needs to wait until a event has occurred
  - A second thread signals the first
- POSIX thread implementation:
  - Initialize a semaphore to **`0`**
  - Have the first thread call **`sem_wait()`** on the semaphore when it needs to wait
  - Have a second thread call **`sem_post()`** when the event has occurred
- Because semaphores have an integer value, the scheduling of the threads doesn't matter
  - If the second thread has already signaled, the first thread will immediately return from **`sem_wait()`**

Turnstiles

- Unlike signaling, which unblocks a *single* thread, the **turnstile** design pattern is used to unblock *many* threads when an event occurs
- POSIX implementation:
  - Initialize a semaphore to **`0`**
  - Have a thread call **`sem_post()`** on the semaphore when the event occurs
  - All threads that need to wait call **`sem_wait()`** followed by **`sem_post()`**
  - Each thread waking up will wake up one more
- Turnstiles work similarly to the broadcast function for condition variables
  - But broadcasts will only wake up those threads that are currently waiting
  - Turnstiles let all threads pass through, even if they reach the **`sem_wait()`** after the event has already happened

Rendezvous

- In the **rendezvous** pattern, two threads signal that they have both reached a specific point in execution
- POSIX implementation:
  - Initialize semaphore A and semaphore B to **`0`**
  - Thread 1 calls **`sem_post()`** on semaphore A and **`sem_wait()`** on semaphore B
  - Thread 2 calls **`sem_post()`** on semaphore B and **`sem_wait()`** on semaphore A
- Each thread will only get blocked until the other one signals
  - Order matters! Flip the waits with the posts and you'll have deadlock
- For larger numbers of threads, using a barrier might be a better approach

Multiplexing

- **Multiplexing** is another design pattern we've already mentioned
- Multiplexing is useful when mutual exclusion is more restrictive than you need, but you still want to limit the total number of threads able to execute a section of code
- POSIX implementation:
  - Initialize a semaphore to ***n***, where ***n*** is the maximum number of concurrent accesses you want to allow
  - Each thread calls **`sem_wait()`** on the semaphore before executing the code
  - Each thread calls **`sem_post()`** on the semaphore after executing the code
- This design pattern can be useful when spawning threads on a server to handle requests
  - We want to prevent too many threads from being created in order to avoid bogging down the server

Lightswitches

- We sometimes want to allow multiple threads of a certain kind to run code concurrently but force others to use mutual exclusion
  - Many threads that only read memory, for example, could access the memory at the same time
  - But only one thread that writes memory should be allowed in
- The **lightswitch** design pattern allows this kind of access
  - The name comes from the idea that the first person into a room turns on a lightswitch and the last person turns it off
- POSIX implementation:
  - Initialize a semaphore to **`1`**
  - Initialize a counter variable to **`0`**
  - Create a lock
  - Whenever a reader thread wants to read:
    - It acquires the lock
    - Increments the counter
    - If the counter is **`1`**, call **`sem_wait()`** on the semaphore
    - Unlock the lock
  - Whenever a reader thread is done reading:
    - It acquires the lock
    - It decrements the counter
    - If the counter is **`0`**, it calls **`sem_post()`** on the semaphore
    - Unlock the lock
  - Writers simply call **`sem_wait()`** to start writing and **`sem_post()`** when done

Producer-Consumer

Producer-consumer

- The **producer-consumer problem** comes up all the time in concurrent systems
  - One or more threads is producing elements that go into a buffer
  - One or more threads is consuming elements from the buffer
- A producer can't put an item into a full buffer and must block
- A consumer can't remove an item from an empty buffer and must block
- Example:
  - An OS thread is putting data into a buffer that's coming across the network
  - A user thread is trying to read data out of that buffer

Unsafe producer-consumer with a bounded queue

- Our implementation uses a circular array (that wraps back around to the beginning)
- The following code is unsafe for two reasons:
  - It doesn't check to see if the buffer is full when enqueuing or empty when dequeuing
  - Changing queue data is **unsafe** for a multi-threaded application

```
void enqueue_unsafe (queue_t *queue, data_t *data)
{
  // Store the data in the array and advance the index
  queue->contents[queue->back++] = data;
  queue->back %= queue->capacity;
}

data_t * dequeue_unsafe (queue_t *queue)
{
  data_t * data = queue->contents[queue->front++];
  queue->front %= queue->capacity;
  return data;
}
```

Safe producer-consumer with a bounded queue and a single producer and consumer

- We could use locks and check a variable giving the total number of elements in the queue
- However, semaphores have this feature built in
- We initialize the **`space`** semaphore to the maximum size of the queue
- We initialize the **`items`** semaphore to **0**

```
void enqueue (queue_t *queue, data_t *data, sem_t *space, sem_t *items)
{
  sem_wait (space);
  enqueue_unsafe (queue, data);
  sem_post (items);
}

data_t * dequeue (queue_t * queue, sem_t *space, sem_t *items)
{
  sem_wait (items);
  data_t * data = dequeue_unsafe (queue);
  sem_post (space);
  return data;
}
```

Safe producer-consumer with a bounded queue and multiple producers and consumers

- Unfortunately, the two semaphores aren't enough when there are multiple producers and consumers
- In that situation, two producers could both be calling **`enqueue_unsafe()`**, potentially causing a race condition with the increment
- The solution is to one lock for producers and one lock for consumers
- We could use a single lock for both, but using two locks allows producers and consumers to modify the queue concurrently yet safely

```
void enqueue (queue_t *queue, data_t *data, sem_t *space, sem_t *items, pthread_mutex_t *producer_lock)
{
  sem_wait (space);
  pthread_mutex_lock (producer_lock);
  enqueue_unsafe (queue, data);
  pthread_mutex_unlock (producer_lock);
  sem_post (items);
}

data_t * dequeue (queue_t * queue, sem_t *space, sem_t *items, pthread_mutex_t *consumer_lock)
{
  sem_wait (items);
  pthread_mutex_lock (consumer_lock);
  data_t * data = dequeue_unsafe (queue);
  pthread_mutex_unlock (consumer_lock);
  sem_post (space);
  return data;
}
```

Readers-Writers

Readers-Writers

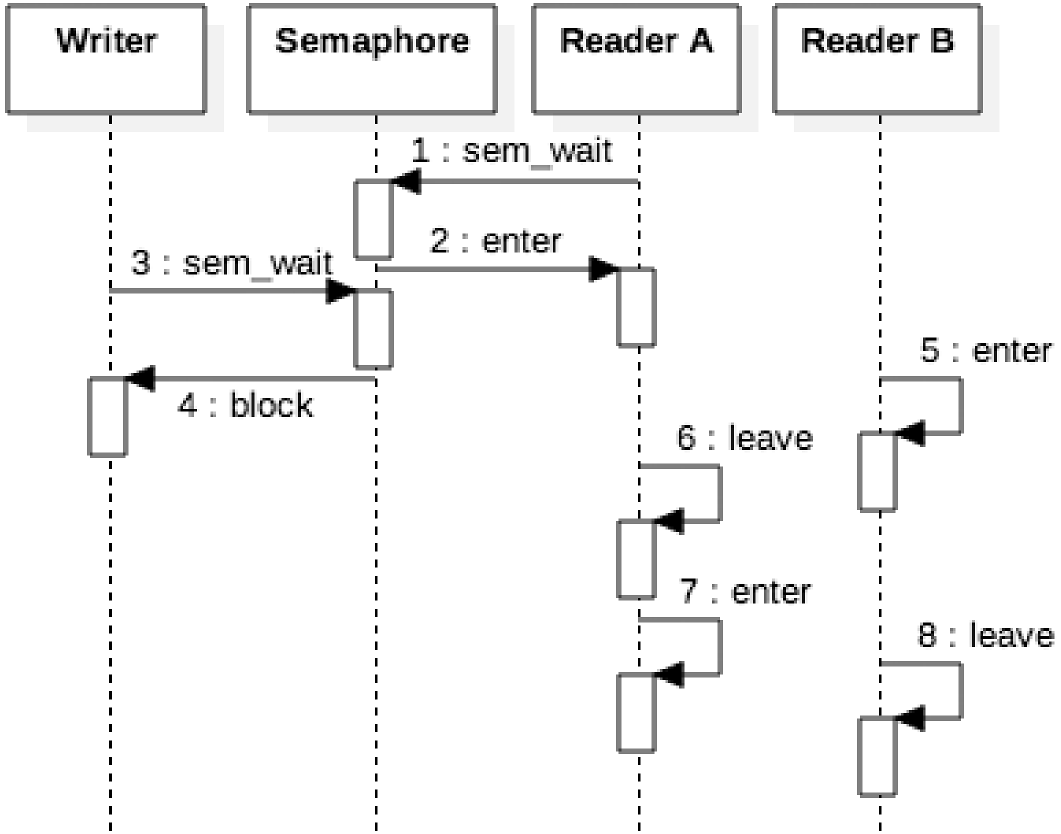
- What if we have a situation where we want to allow an unlimited number of reader threads to read data?
- But if a single writer needs to write, no other threads can access the data
- Changing the data can cause race conditions, but merely reading it concurrently is fine
  - And can make reading much faster!

First solution: Lightswitches

- This is exactly the scenario we solved with lightswitches:
  - Initialize a semaphore to **`1`**
  - Initialize a counter variable to **`0`**
  - Create a lock
  - Whenever a reader thread wants to read:
    - It acquires the lock
    - Increments the counter
    - If the counter is **`1`**, call **`sem_wait()`** on the semaphore
    - Unlock the lock
  - Whenever a reader thread is done reading:
    - It acquires the lock
    - It decrements the counter
    - If the counter is **`0`**, it calls **`sem_post()`** on the semaphore
    - Unlock the lock
  - Writers call **`sem_wait()`** to start writing and **`sem_post()`** when done

What's the problem with this solution?

- When a reader comes into the room, it becomes blocked for writers
- If more readers come in before others leave, writers might *never* get to enter
- What do we do?



Second solution: Add a turnstile

- We add a turnstile for the readers
  - They pass through without any problem at first
- When a writer wants to write, it waits on the reader semaphore
- This blocks any new readers from entering

Illustration of second solution

- The system starts off with its two semaphores having the following values:
  - Lightswitch: 1
  - Turnstile: 1

Reader 1 Enters
- Lightswitch: 0
- Turnstile: 1

Reader 2 Enters
- Lightswitch: 0
- Turnstile: 1

Writer Tries to Enter (Blocked)
- Lightswitch: 0
- Turnstile: 0

Reader 3 Tries to Enter (Blocked)
- Lightswitch: 0
- Turnstile: 0

Reader 1 Exits
- Lightswitch: 0
- Turnstile: 0

Reader 2 Exits
- Lightswitch: 1
- Turnstile: 0

Writer Enters
- Lightswitch: 0
- Turnstile: 1

Writer Exits
- Lightswitch: 1
- Turnstile: 1

Reader 3 Enters
- Lightswitch: 0
- Turnstile: 1

Search-insert-delete problem

- The readers-writers problem can be extended to a problem with the following characteristics:
  - Searchers are searching for data (similar to regular readers)
  - Inserters are a kind of writer that only adds data
  - Deleters are a kind of writer that only removes data
- Rules:
  - Searchers can be concurrent with each other and an inserter
  - Inserters can be concurrent with searchers, but there can only be one inserter at a time
  - Deleters must be mutually exclusive with everyone
- You can imagine a version of this problem for concurrent accesses to databases

Search-insert-delete solution

- Searchers use a lightswitch as before
- Inserters use their own lightswitch but also have a lock to prevent concurrent insertions with each other
- Deleters must wait on both lightswitches
- This solution works because a deleter can enter only when there are no searchers or inserters

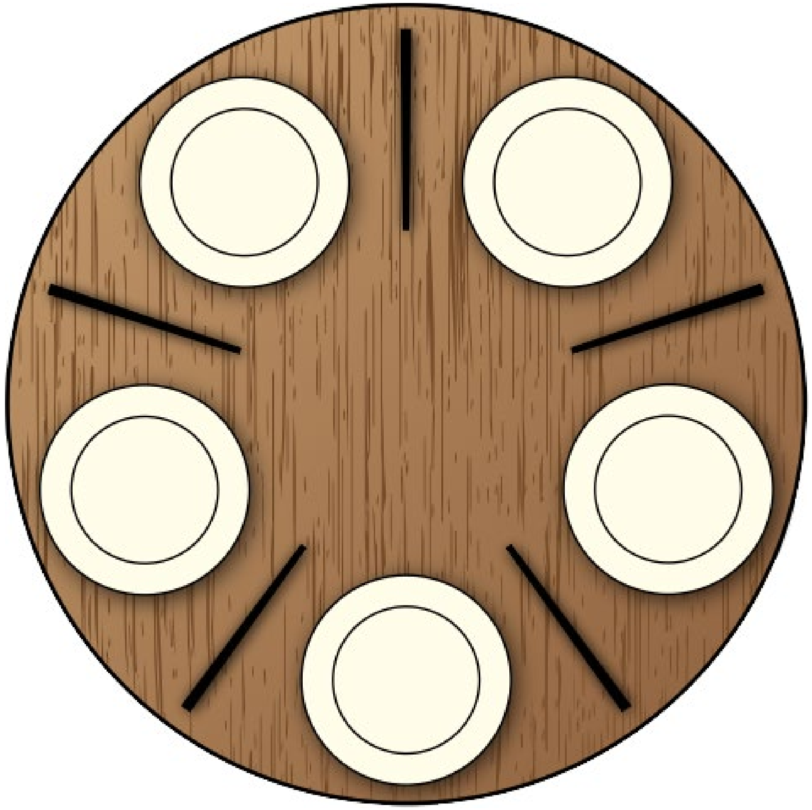
Issues with this solution

- Like our first solution for readers-writers, deleters can be starved if searchers or inserters continue to arrive
  - Never getting to run is called **starvation**
- We could increase fairness for this solution by adding turnstiles as well
  - One turnstile semaphore could be shared by all searcher and inserters
  - When a deleter comes along, it waits on the turnstile, blocking all new searchers and inserters from entering
  - When a deleter gets access to the critical section, it posts on the turnstile, allowing all waiting threads to get to their respective lightswitches

Dining Philosophers

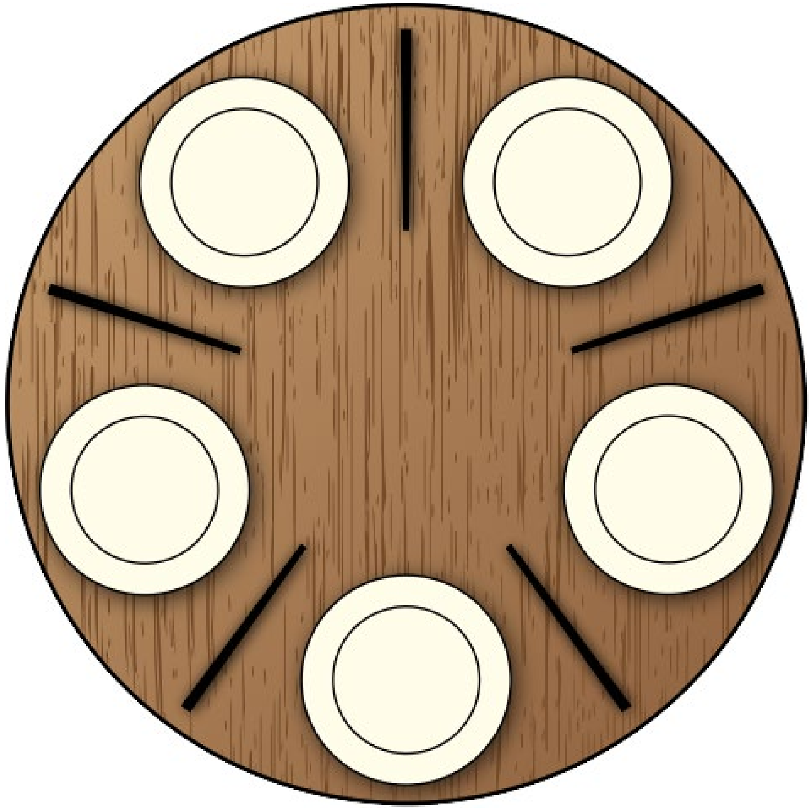
Dining philosophers

- A classic problem illustrating the difficulties of concurrency is the **dining philosophers problem**
- Some number of philosophers sit at a round table and only do two things:
  - Think
  - Eat rice
- In order to eat rice, they have to pick up two chopsticks, one on the left and one on the right
  - The book has them eat with forks, but chopsticks make more sense for the problem
  - You can eat rice with one fork, but you can't eat rice with one chopstick
  - **Critically important:** The numbers of chopsticks and philosophers are equal



The problem

- We have to enforce mutual exclusion for the chopsticks
- Two philosophers can't hold onto the same chopstick at the same time
- It's unpredictable when each philosopher is going to finish thinking and start eating
- We need a solution that works no matter what



A solution with deadlock

- Let's say there are **`SIZE`** philosophers (and **`SIZE`** chopsticks)
- We can create **`SIZE`** locks, one for each chopstick
- Then, each philosopher will acquire the lock for her left chopstick followed by the lock for her right chopstick
- In the following code, **`self`** is the index of the philosopher

```
void * philosopher (void * _args)
{
  struct args *args = (struct args *) _args;
  int self = args->self;                        // Philosopher index
  int next = (self + 1) % SIZE;
  pthread_mutex_lock (args->locks[self]);       // Pick up left chopstick
  pthread_mutex_lock (args->locks[next]);       // Pick up right chopstick
  // Eat rice
  pthread_mutex_unlock (args->locks[next]);     // Put down right chopstick
  pthread_mutex_unlock (args->locks[self]);     // Put down left chopstick
  // Do other work and exit thread
}
```

Why it has deadlock

- Imagine that every philosopher picks up her left chopstick at the same moment
- Now, each will wait for another one to give up what would be their right chopstick…forever
- We have the four conditions for deadlock:
  - **Mutual exclusion:** Only one philosopher can hold the lock for a chopstick
  - **Hold-and-wait:** Each philosopher acquires chopstick and tries to get another
  - **No preemption:** No philosopher can force another to give up her chopstick
  - **Circular wait:** Under the right circumstances, every philosopher can be waiting for every other in a circle

Solution by limiting access

- One solution is to add a semaphore initialized to **`SIZE – 1`**
- Then, only **`SIZE – 1`** philosophers could try to grab a chopstick

```
void * philosopher (void * _args)
{
  struct args *args = (struct args *) _args;
  int self = args->self;                       // Philosopher index
  int next = (self + 1) % SIZE;
  sem_wait (args->can_eat);                    // Multiplexing semaphore
  pthread_mutex_lock (args->locks[self]);      // Pick up left chopstick
  pthread_mutex_lock (args->locks[next]);      // Pick up right chopstick
  sem_post (args->can_eat);                    // Multiplexing semaphore
  // Eat rice
  pthread_mutex_unlock (args->locks[next]);    // Put down right chopstick
  pthread_mutex_unlock (args->locks[self]);    // Put down left chopstick
  // Do other work and exit thread
}
```

Solution by breaking hold and wait

- In our example, the philosopher gets the first chopstick and immediately tries to get the second
- In real situations, some work might need to get done between acquiring resources
- To avoid delays, it might be desirable to instead get a chopstick and then *try* to get the second, releasing the first if that fails

```
while (! success)
  {
    pthread_mutex_lock (args->locks[self]);      // Pick up left chopstick
    // Perform some work
    // Then, try to get the right chopstick
    if (pthread_mutex_trylock (args->locks[next]) != 0)
      {
        // Undo current progress
        pthread_mutex_unlock (args->locks[self]); // Put down left chopstick
      }
    else
      success = true;
  }
```

Solution by imposing order

- We can break the circular wait condition with a clever ordering
- If every philosopher picks up her left chopstick at the same time, we're stuck
- But what if exactly one picked up her right chopstick first?
  - Deadlock would become impossible!

```
void * philosopher (void * _args)
{
  struct args *args = (struct args *) _args;
  int self = args->self;                      // Philosopher index
  int next = (self + 1) % SIZE;
  if (self > next) swap (&self, &next);       // Last philosopher swaps order
  pthread_mutex_lock (args->locks[self]);     // Pick up left chopstick
  pthread_mutex_lock (args->locks[next]);     // Pick up right chopstick
  // Eat rice
  pthread_mutex_unlock (args->locks[next]); // Put down right chopstick
  pthread_mutex_unlock (args->locks[self]); // Put down left chopstick
  // Do other work and exit thread
}
```

Distributed Computing

Parallelism vs. concurrency

- Although a lot of computation involves both parallelism and concurrency, they're two different things
- **Concurrency** means that tasks can interact with each other
- **Parallelism** means that two tasks are running at the same time
- You can have concurrency without parallelism
  - Example: A multi-threaded program on a single-core system, which can still have race conditions
- You can have parallelism without concurrency
  - Example: Programs running on separate cores or processors that are computing part of a larger answer without coordination

Task parallelism and data parallelism

- There are two fundamental kinds of parallelism that are possible
- **Task parallelism**
  - Breaking up a problem into subtasks that can be run in parallel
  - Example: Alice cooks dinner, Bob cleans the house, and Catherine gets vengeance on their enemies
- **Data parallelism**
  - Doing the same tasks in parallel but on different data
  - Alice, Bob, and Catherine each chop up 1/3 of the total amount of carrots for a soup

Embarrassingly parallel

- The easiest kind of problems to parallelize are called **embarrassingly parallel**
  - Maybe there are many unrelated tasks that all need to get done
  - Maybe there's lots of data to process, and no coordination is necessary to process it
- The following code shows an embarrassingly parallel problem, since initializing the array could easily be divided up among many tasks

```
for (int i = 0; i < 100000000; ++i)
  array[i] = i * i;
```

Divide-and-conquer

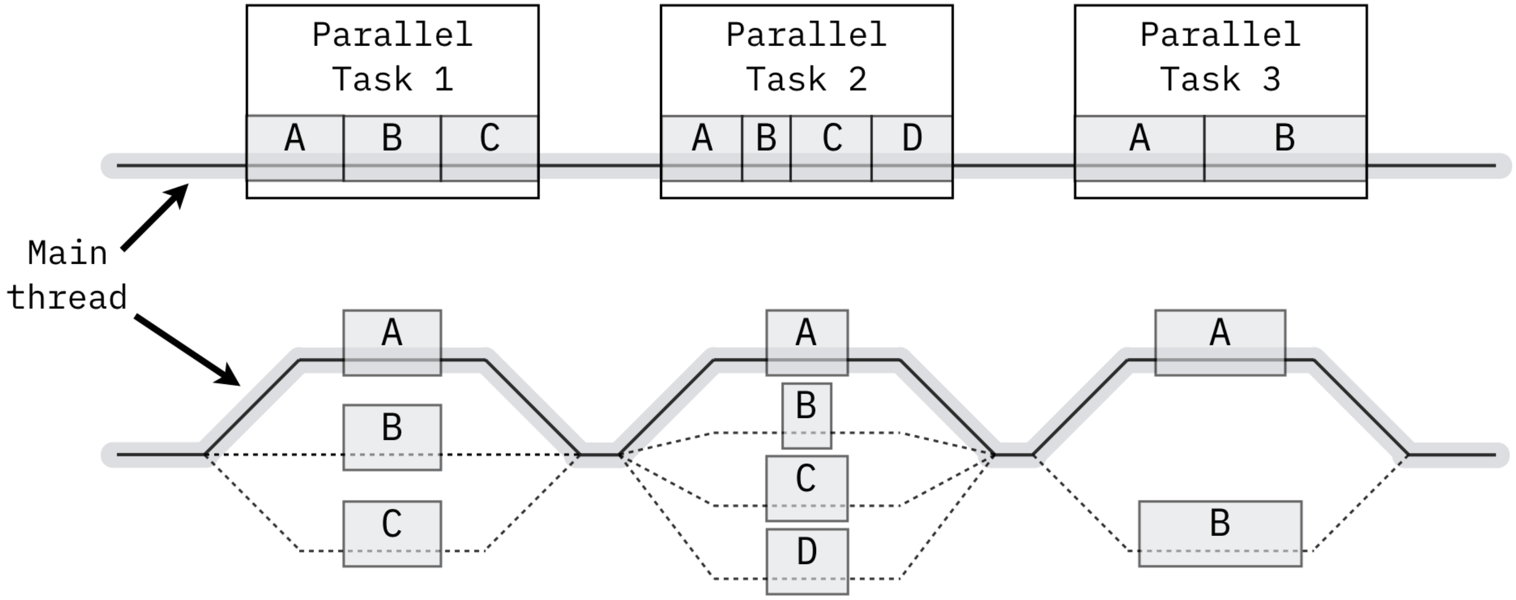
- Algorithms themselves can suggest approaches for parallelism
- Divide-and-conquer algorithms divide problems into parts, find answers for the sub-problems, and then combine those answers into an overall solution
  - Quicksort partitions into two subarrays and then recursively sorts
  - Merge sort also divides and recursively sorts
- As discussed in COMP 4500, many important algorithms have a divide-and-conquer shape, and it's often possible to let each divided task be handled by a separate thread

Pipelines

- The idea of a pipeline is to divide a task into independent steps, each of which can be performed by dedicated hardware or software
- Example RISC pipeline:
  1. Instruction fetch
  2. Decode
  3. Execute
  4. Memory Access
  5. Writeback

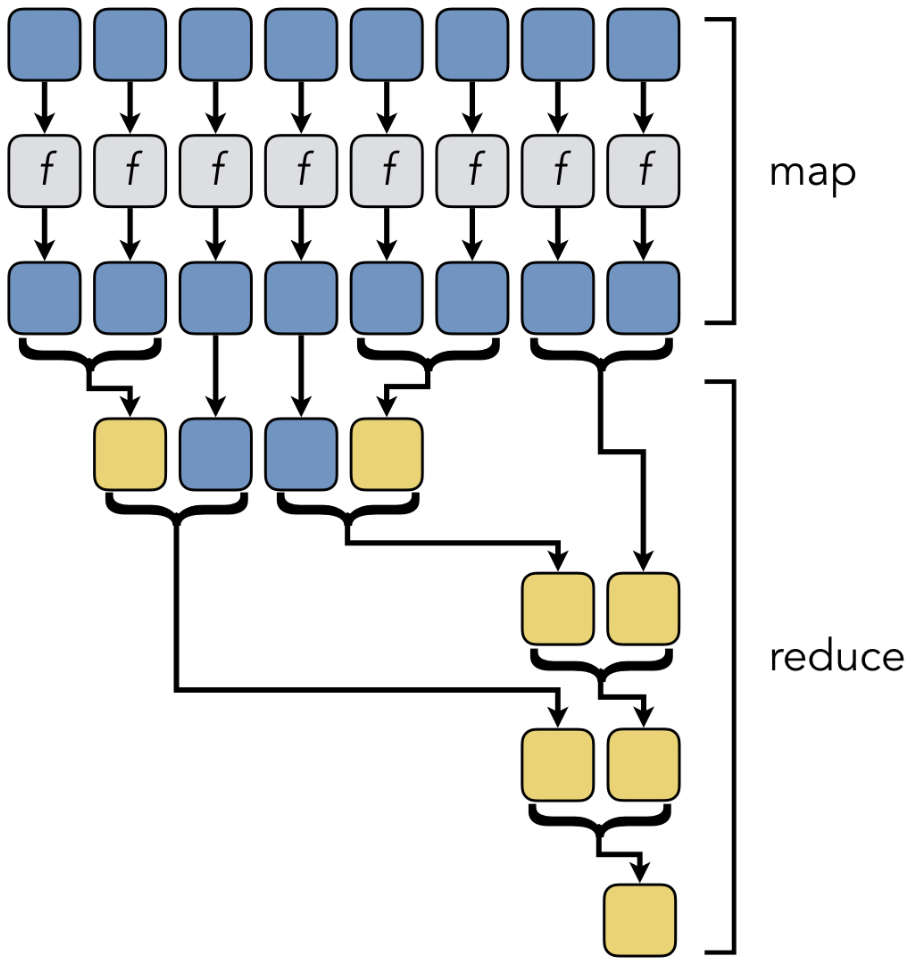
Fork/join

- The **fork/join** pattern uses a main thread that spawns additional threads when there are parallel tasks to be done
- After those tasks complete, the main thread joins the spawned threads
- A fork/join pattern could be used for either task parallelization or data parallelization



Map/reduce

- **Map/reduce** is similar to fork/join
- The biggest difference is a philosophical one about how the work is described
- Map/reduce has two stages:
  - Map applies a function to each piece of input data
  - Reduce combines the results to get a final answer
- Map/reduce is commonly used on clusters and distributed systems
  - The open source Apache Hadoop is a popular tool for map/reduce computing



Manager/worker

- The manager/worker thread pattern is commonly used with task parallelism
- Independent tasks are given to work threads that communicate with a central management thread
  - Event handling, for example, can be viewed as manager/worker
  - Workers can also wait for a data value to change from **NULL**, as in the code below

```
void * worker (struct args * _args)
{
  struct args *args = (struct args *) _args;
  pthread_mutex_lock (args->lock);
  while (true)
    {
      while (args->data == NULL) // Wait for data
        pthread_cond_wait (args->data_received, args->lock);
      if (! args->running) pthread_exit (NULL);
        // Process data
    }
}
```

Thread pools

- Rather than worrying about creating too many threads initially or dynamically creating threads, one approach is a **thread pool**
  - A thread pool is a fixed number of threads with a queue of tasks
  - When a thread finishes its work, it can dequeue a new task
- Thread pools advantages:
  - The cost of creating threads is only paid once
  - Resource consumption is more predictable because there won't suddenly be a lot more threads
  - Each thread self-manages the load by getting more work when it finishes
- Thread pools disadvantages:
  - Cache performance can be poor because there's no coordination between which thread is doing what
  - Crashes and errors can be hard to recover from since we won't know which thread was doing the thing that failed
  - Managing the task queue requires synchronization that could slow things down

Limits of Parallelism

Speedup

- **Speedup** is how much faster a parallel solution is compared to a sequential one
- The formula is $\frac{T_{sequential}}{T_{parallel}}$
  - $T_{sequential}$ is the amount of time the sequential solution takes
  - $T_{parallel}$ is the amount of time the parallel solution takes
- Thus, if a sequential solution to a problem takes 100 seconds, and the parallel solution takes 50 seconds, the speedup is 2

Amdahl's Law and strong scaling

- What if you had 16 cores? Or 1,000 cores? Or a million?
- How much speedup can you get?
- Some part of the program has to be executed sequentially
  - Reading input
  - Starting threads
  - Combining results
- Amdahl's law says that the maximum speedup possible is $\frac{1}{(1-p)+\frac{p}{N}}$
  - p is the fraction of a program that can be parallel
  - N is the number of processors

Consequences of Amdahl's law

- What if we had unlimited cores?
- We can take the equation $\frac{1}{(1-p)+\frac{p}{N}}$ and plug in ∞ for N
- Doing so would mean, even with infinite cores, we could never have better speedup than $\frac{1}{(1-p)}$
- Let's say that 90% of a program can be parallelized
- What's the maximum possible speedup you can get?
- $S = \frac{1}{(1-p)} = \frac{1}{(1-.9)} = \frac{1}{.1} = 10$

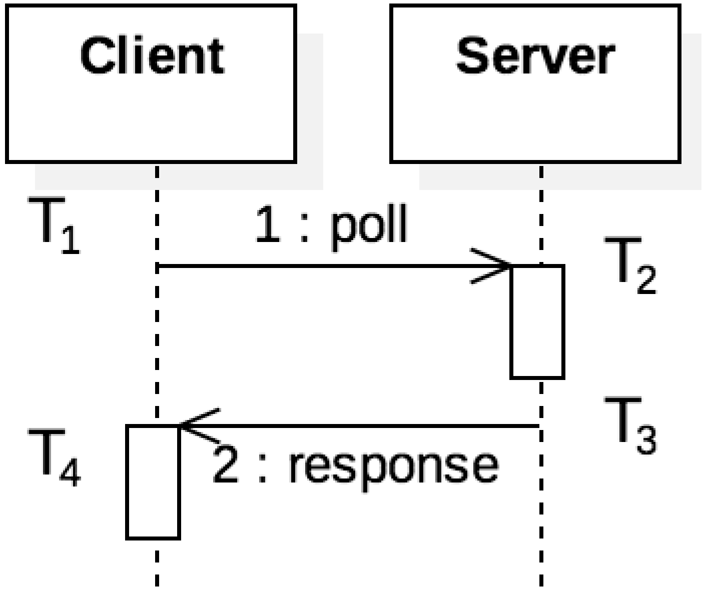
Timing in Distributed Environments

Timing in distributed environments

- When working on a single computer, there's only one clock
- Thus, multiple threads can use this clock to record events in a mutually consistent way
  - Like adding timestamps to log files
- Distributed systems don't have a single, reliable clock
  - Each computer might have a slightly (or completely) different time
  - Clocks on each computer drift with respect to each other
  - These problems get worse as distance (and network delays) increase

Clock synchronization

- We can synchronize clocks based on a centralized server
- A problem is that the time a message takes in the network is unpredictable
- Network Time Protocol (NTP) is a protocol to do this:
  - Client sends a message at T_1
  - Server receives the message at T_2
  - Server replies at T_3
  - Client receives the message at T_4
- Offset = $\frac{(T_2-T_1)+(T_3-T_4)}{2}$
  - The offset is a measurement of the difference in times between the client and server
- Delay = $(T_4 - T_1) - (T_3 - T_2)$
  - The delay is a measurement of how long it takes for the messages to make a round trip
- Algorithms process a number of offset and delay values to try to find the most accurate offset

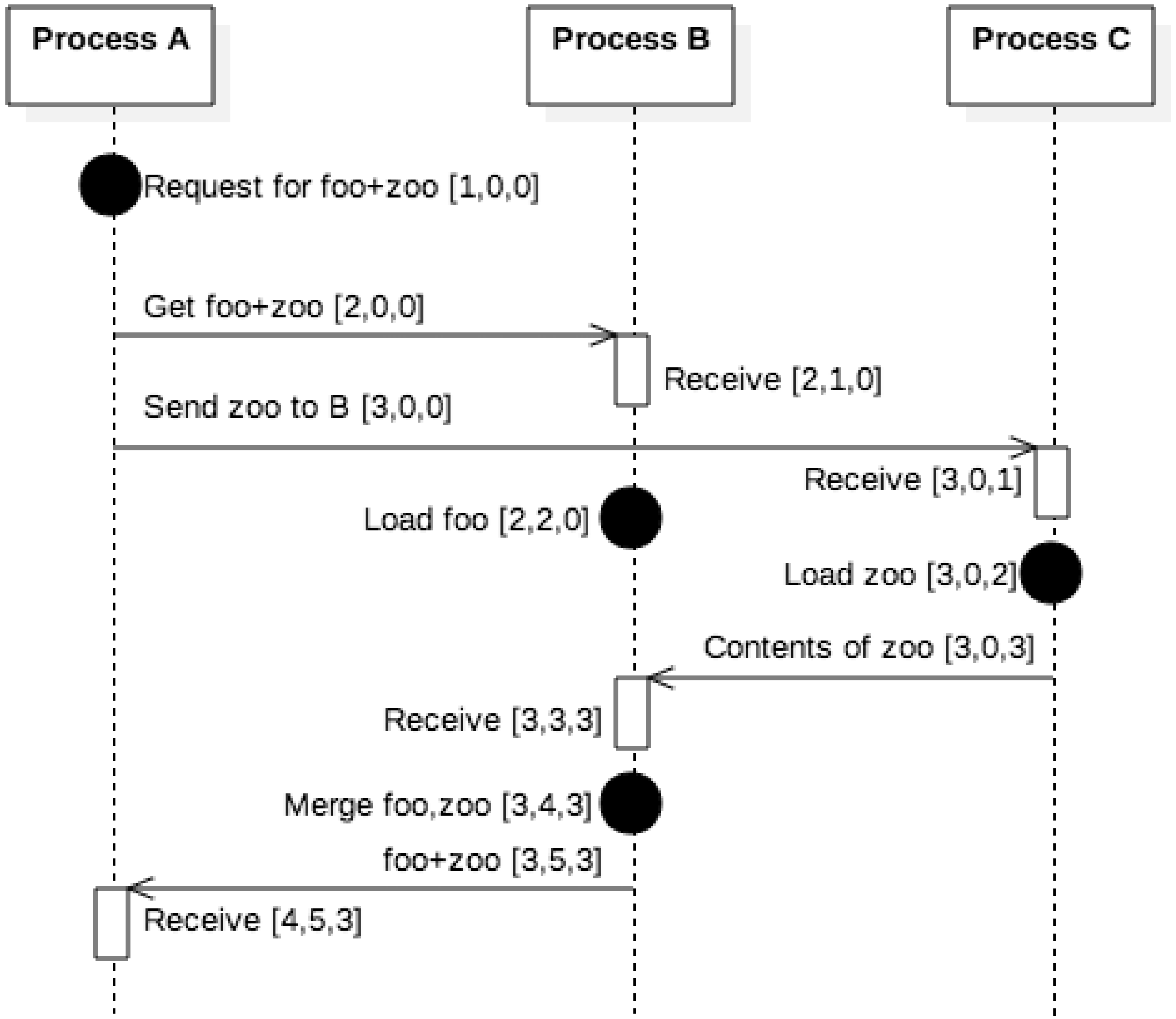


Lamport timestamps

- Logical clocks are an alternative to using exact times, using messages to track the order of events
- Lamport timestamps are a way to implement logical clocks
- Each process keeps an internal counter of events that it sees
  - When a local event occurs, the counter is incremented
  - When a process sends or receives a message, it increments its counter
- Messages have timestamps
  - When a process receives a message, it updates its internal counter to the message's timestamp if that timestamp is larger

Vector clocks

- Lamport timestamps only give indirect information about the state of other processes
- **Vector clocks** extend the idea of Lamport timestamps by making every process keep a counter for every process
- When a message from one process arrives, the receiving process can update all of its counters based on whatever is larger
- Vector clocks give much more information about how many events have been experienced by other processes



Reliable Storage and Location

Reliable data storage

- If you want to get a file from a web server, you can go to a URL and make an HTTP request
- Unfortunately, if that server is down or unreachable, you can't get the file
- For this reason, distributed systems are often used to store data
- A key feature of distributed data storage is **replication**, keeping multiple copies of the same data
  - Replication avoids a single point of failure
  - If done correctly, replication can also do load balancing, improving performance by providing multiple sources for data

Google File System

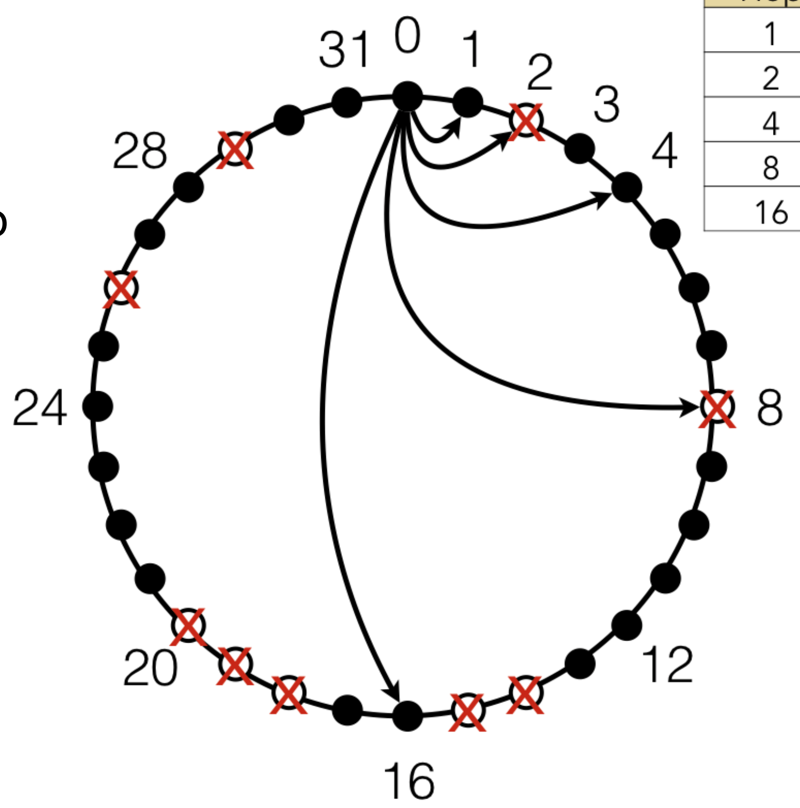
- The Google File System (GFS) is a distributed storage system
- GFS was designed to store Google's internal data, like the data structures used for PageRank
- Files are often large, so they're broken into chunks
- Chunks are stored on chunkservers as regular files
- A master server stores a table mapping files chunks to their locations

Distributed hash tables

- GFS was designed by Google for its own purposes
  - It uses a central server
  - Servers keep information about each other
- What if we have no idea what servers are going to be in the network?
- **Distributed hash tables (DHT)** are an approach for mapping arbitrary objects to arbitrary servers
- DHTs are a way to organize a peer-to-peer network to avoid query flooding

Chord DHT

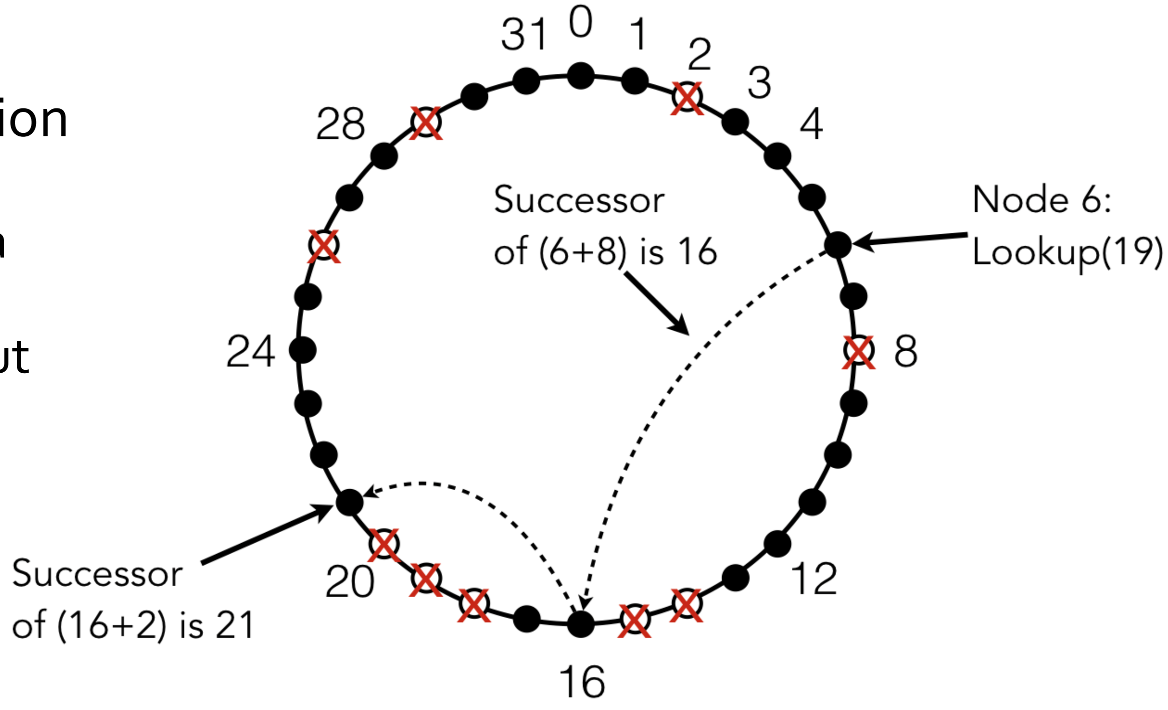
- Chord was one of the first algorithms for a DHT, introduced in 2001
- Each node has a unique identifier (often its IP address) that's hashed to provide a location in a circle
  - If the hash is ***n*** bits long, the DHT can support up to 2^n nodes
- Most locations in the circle are empty
- Each node has a "finger table," tracking successor elements in increasing powers of 2 away on the circle
  - If the power of 2 node is missing, it tracks the next non-missing node
- The example on the right is only for $2^5 = 32$ nodes



Hop	Successor(0+hop)
1	1
2	3
4	5
8	9
16	16

Files in Chord DHT

- When a file is added, it's hashed
- Whichever node has that hash value (or is its successor) is the location of that file
- On the right, node 6 is looking for a file at location 19 (the successor of 18)
  - It looks at 6 + 8 = 14, which doesn't exist but has a successor of 16
  - Then it looks at 16 + 2 = 18, which doesn't exist but has a successor of 21
  - Node 21 is where the file is supposed to be
- The details get a little more complex, but the practical result is that a file can be found with O(log ***n***) requests, where ***n*** is the size of the network
- Replication is done by caching files at nodes that were part of the lookup to find the file



Consensus in Distributed Systems

Consensus

- Reaching consensus is the goal of many distributed protocols
- To reach consensus, a protocol must have three properties:
  - **Termination:** Every correct (non-failing) process will eventually decide on a value
  - **Integrity:** If every correct process proposes the same value, any correct process must decide that value
  - **Agreement:** All correct processes decide the same value
- Examples
  - In GFS, a consensus protocol could tell any node whether a particular file was on a particular node
  - In NTP, nodes will be able to agree on synchronized time

Failure

- However, processes do fail in distributed systems
- Failure could mean making some error, crashing, going into an infinite loop, or losing connection to the network
  - Processes could even be malicious, trying to undermine the system
- Even in the face of (many?) failures, we'd like the distributed system to reach consensus
- A common analogy used to describe this problem is the Byzantine generals problem

Byzantine generals

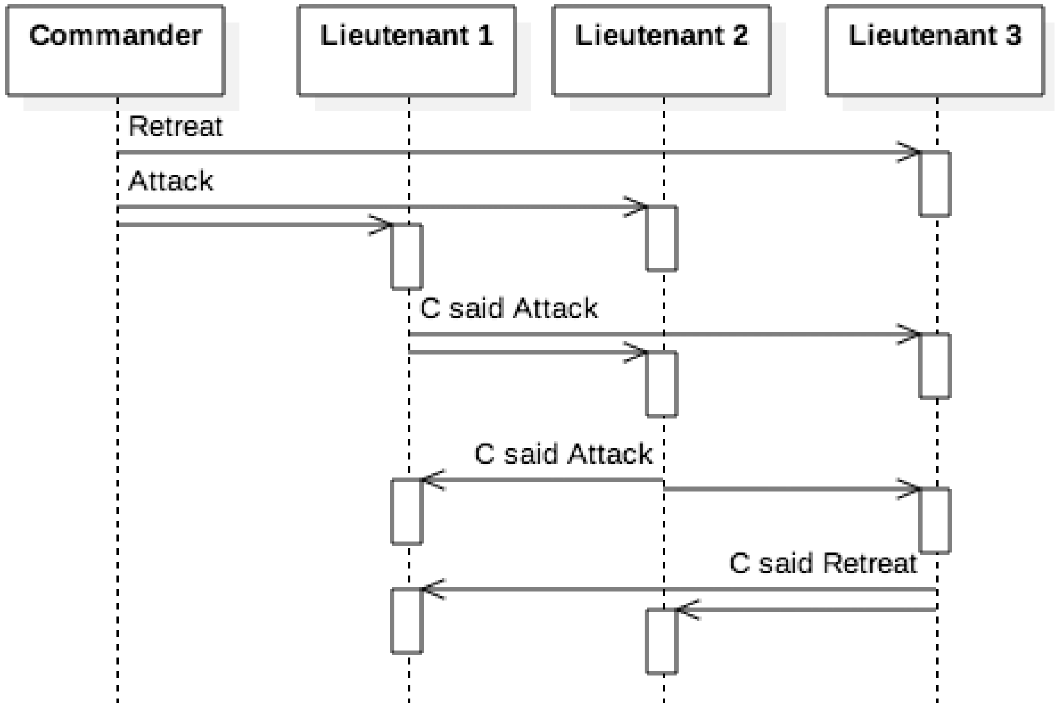
- A version of the Byzantine generals problem imagines:
  - One general is the commander who decides what to do
  - The other two are lieutenants who check with each other to make sure that they got the same message from the commander
- What if there's one bad general?
  - A bad commander could send retreat to one lieutenant and attack to the other
  - A bad lieutenant could receive attack from the commander but send retreat to the other lieutenant
  - A good lieutenant couldn't distinguish between those two situations

Limits on consensus

- Consensus is hard
  - Failing processes can mess things up for correct processes
  - Because there's limited information, a process can appear to be correct to one part of the system and failing to another
- A **Byzantine failure** is exactly this kind
  - There's conflicting information, and it's impossible to determine what's reliable

The 1/3 limit

- It's not an accident that the number of generals chosen is three
- If strictly less than 1/3 of the nodes are failing, it's possible to achieve consensus
- If we extend the problem to four generals (three lieutenants), then generals who are working can decide on a consensus using majority rule
- Even a bad commander who issues confusing orders won't mess up the system
- However, knowing what the consensus is doesn't tell us which nodes are failing
- Also, this 1/3 limit depends on synchronous communication



Blockchains

- Blockchains are a form of distributed ledger
- Unlike banks, which are centralized authorities for transactions that have occurred at the banks, blockchains try to record transactions in a distributed way with *no* central authority
- Although similar ideas existed before, blockchains as we know them were invented by Satoshi Nakamoto (real identity unknown) in 2008
- The original blockchain idea was intended to keep track of bitcoin transactions
- Now, most cryptocurrencies use some form of blockchain to track transactions

Double-spending

- Blockchains are distributed systems that can be used to record almost anything
- But their use has been dominated by cryptocurrency
- A central problem that any digital money faces is **double-spending**
  - What stops someone from spending a digital token more than once?
- Transactions are recorded in blockchains
- Two competing blockchains could record different transactions, but the longer chain is considered the valid one

Proof-of-work

- Blockchains are built by recording transactions along with other data that hashes in a specified pattern
  - Usually, a hash value with a certain number of zeroes at the beginning
- It's easy to check that the transaction has the right hash value
- But it's computationally difficult to generate data that has a hash with a certain number of zeroes at the beginning
- And that's what mining is: Trying random strings until something has the right hash value
- Since a large number of strings will have the right hash value, an entity with more than 50% of the computational power working on a blockchain network could outpace everyone else writing transactions, taking control of it

Libraries You Should Know

String manipulation

- String manipulation is annoying but necessary in C
- You should be able to use the following functions:
  - `char *strcat(char *dest, const char *src)`
    - Appends the string **src** to the end of the string pointed **dest**
  - `char *strncat(char *dest, const char *src, size_t n)`
    - Appends the string **src** to the end of the string **dest**, up to n characters
  - `char *strchr(const char *str, int c)`
    - Searches for the first occurrence of the character **c** (an unsigned char) in the string **str**
  - `int strcmp(const char *str1, const char *str2)`
    - Compares strings **str1** and **str2**
  - `int strncmp(const char *str1, const char *str2, size_t n)`
    - Compares at most the first **n** bytes of strings **str1** and **str2**
  - `char *strcpy(char *dest, const char *src)`
    - Copies the string **src** into **dest**
  - `char *strncpy(char *dest, const char *src, size_t n)`
    - Copies up to **n** characters from the string **src** into **dest**
  - `size_t strlen(const char *str)`
    - Computes the length of the string **str**
  - `char *strstr(const char *haystack, const char *needle)`
    - Finds the first occurrence of string **needle** in the string **haystack**
  - `char *strtok(char *str, const char *delim)`
    - Breaks string **str** into a series of tokens separated by **delim**

File I/O

- File I/O is the key abstraction for all I/O in Linux
- So you should be able to use the following functions:
  - `int open (char *path, int flags, int perms)`
    - Open the file specified by `path`
    - Possible flags: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, `O_APPEND`, `O_TRUNC`
    - Permissions: Only needed when creating a file, and octal values are the easiest
  - `int close(int fd)`
    - Close the file given by file descriptor `fd`
  - `int read (int fd, char *buffer, int size)`
    - Read from file descriptor `fd` into `buffer` a maximum of `size` bytes
    - Returns the number of bytes successfully read
  - `int write(int fd, char *buffer, int size)`
    - Write from `buffer` into file descriptor `fd` a maximum of `size` bytes
    - Returns the number of bytes successfully written

Process management functions

- You should know these pretty well:
  - `pid_t fork (void)`
    - Fork a new version of the current process at exactly the same point in the program
  - `int execl(char *path, char *arg0, ..., NULL)`
  - `int execle(char *path, char *arg0, ..., NULL, char* envp[])`
  - `int execlp(char *file, char *arg0, ..., NULL)`
  - `int execv(char *path, char *argv[])`
  - `int execve(char *path, char *argv[], char *envp[])`
  - `int execvp(char *file, char *argv[])`
    - Execute a process (replacing the current process) with the given arguments and environment variables
  - `pid_t wait(int *stat_loc)`
    - Wait for all child processes to finish
  - `int pipe(int pipefd[2])`
    - Create a pipe where `pipefd[0]` is the reading end of the pipe and `pipefd[1]` is the writing end
  - `int dup2(int actual, int replaced)`
    - All reads from and writes to `replaced` will actually be read from or written to `actual`

Networking

- You don't need to have these functions memorized, but you should be familiar enough to read them in code and understand them
  - `int socket (int domain, int type, int protocol)`
    - Create a socket, which will work like a file descriptor
  - `int bind (int socket, const struct sockaddr *address, socklen_t address_len)`
    - Bind the socket to a port
  - `int listen (int socket, int backlog)`
    - Set up the socket for listening
  - `int accept (int socket, struct sockaddr *address, socklen_t *address_len)`
    - Accept an incoming connection
  - `int connect (int socket, const struct sockaddr *address, socklen_t address_len)`
    - Connect to a listening socket

Threads

- You don't need to have these functions memorized, but you should be familiar enough to read them in code and understand them:
  - `int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void(*start_routine)(void*), void *arg)`
    - Create a new thread
  - `void pthread_exit (void *value_ptr)`
    - Exit from the current thread, possibly returning a result
  - `void pthread_join (pthread_t thread, void *value_ptr)`
    - Wait for a thread to end (getting a pointer to its result, if any)

Synchronization

- You don't need to have these functions memorized, but you should be familiar enough to read them in code and understand them:
  - `sem_t *sem_open (const char *name, int flag, /* mode_t mode, unsigned int value */ )`
    - Return (and possibly create) a named semaphore, using the usual **`flag`** and **`mode`** constants
    - **`value`** determines the initial value of the semaphore (often 0)
  - `int sem_wait (sem_t *sem)`
    - Block if the semaphore's value is 0, decrement after continuing
  - `int sem_post (sem_t *sem)`
    - Increment the semaphore's value, unblocking a process if the value is 0
  - `int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)`
    - Create a mutex with the specified attributes
  - `int pthread_mutex_lock (pthread_mutex_t *mutex)`
    - Acquire a mutex, blocking until you succeed
  - `int pthread_mutex_unlock (pthread_mutex_t *mutex)`
    - Release the mutex

Upcoming

Next time…

- There is no next time!

Reminders

- Finish Assignment 8
  - **Due tonight before midnight!**
- **Study for the final exam:**
  - **Wednesday, April 30, 2025**
  - **8:00 – 10:00 a.m.**