

CS 31: Intro to Systems C Programming

L24-25: Synchronization and Race Conditions

Vasanta Chaganti & Kevin Webb

Swarthmore College

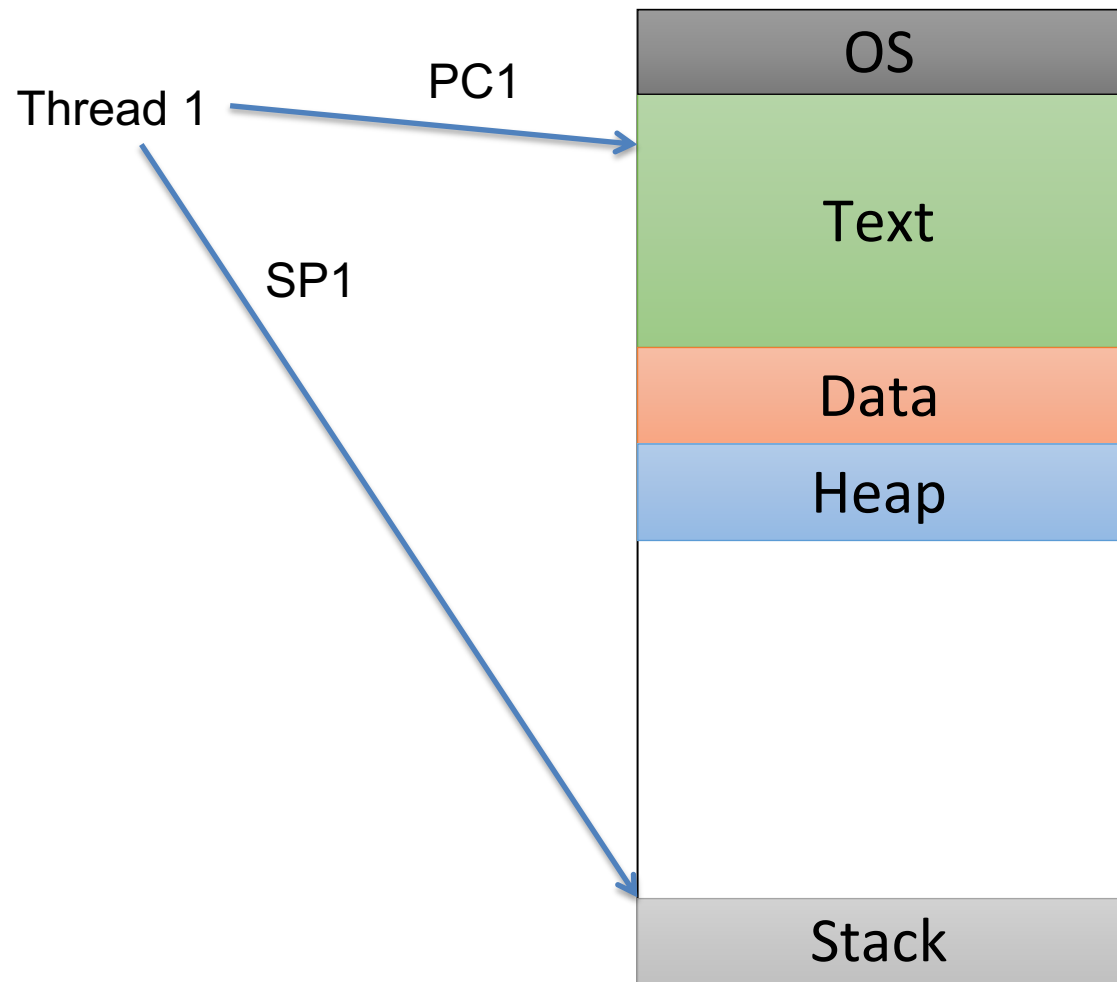
Dec 7, 12, 2023

Reading Quiz

Threads

- Modern OSes **separate the concepts of processes and threads.**
 - The process defines the address space and general process attributes (e.g., open files)
 - The thread **defines a sequential execution stream within a process** (PC, SP, registers)
- A thread is bound to a single process
 - Processes, however, can have multiple threads
 - **Each process has at least one thread (e.g. main)**

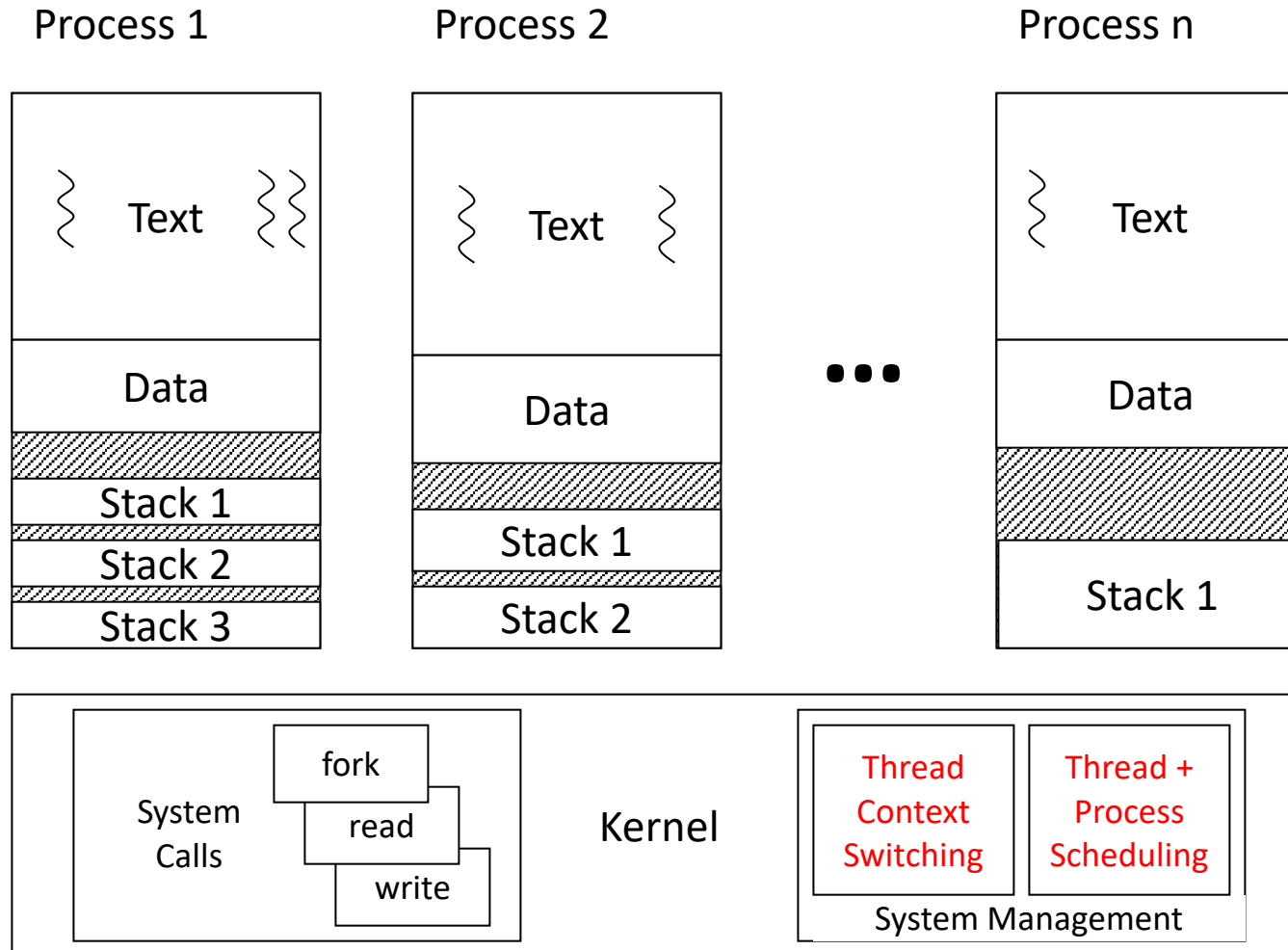
Threads



This is the picture we've been using all along:

A process with a single thread, which has execution state (registers) and a stack.

Kernel-Level Threads



Kernel Context switching over threads

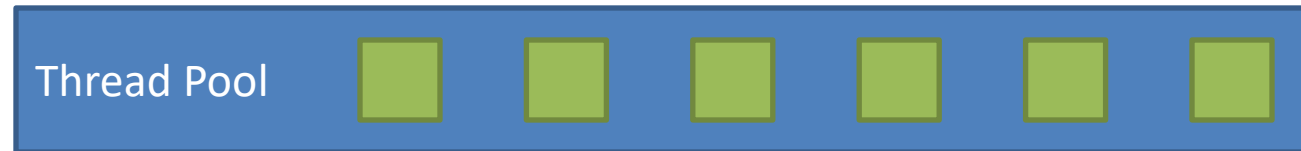
Each process has explicitly mapped regions for stacks

Common Thread Patterns

- Producer / Consumer (a.k.a. Bounded buffer)
- Thread pool (a.k.a. work queue)
- Thread per client connection

Thread Pool / Work Queue

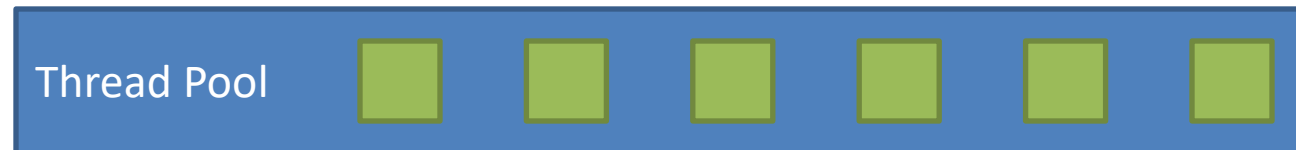
- Common way of structuring threaded apps:



Thread Pool / Work Queue

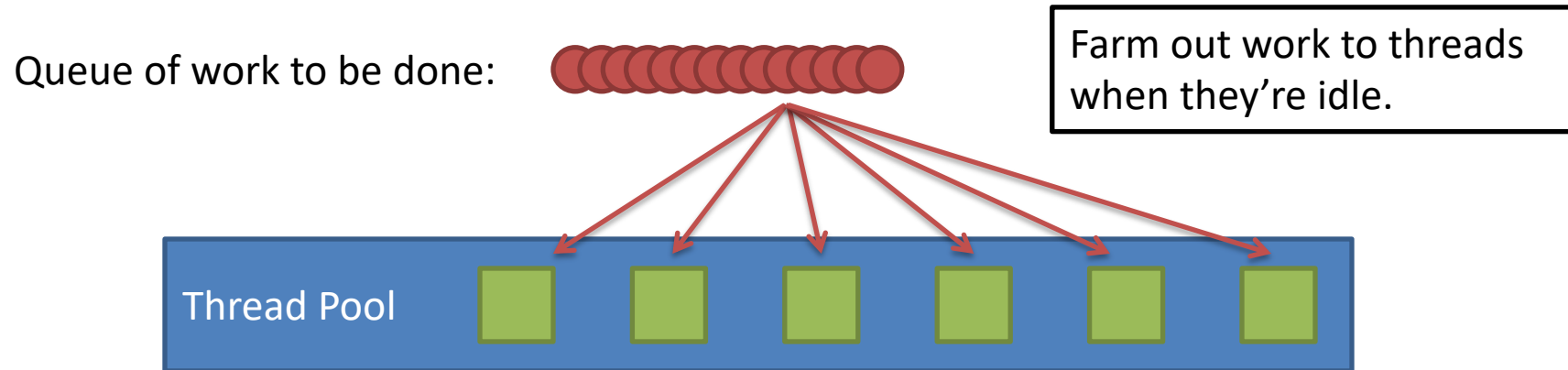
- Common way of structuring threaded apps:

Queue of work to be done: 




Thread Pool / Work Queue

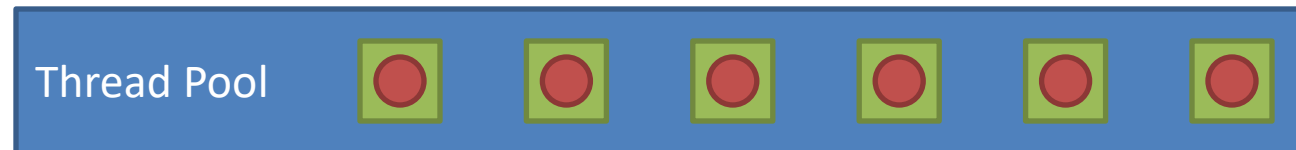
- Common way of structuring threaded apps:



Thread Pool / Work Queue

- Common way of structuring threaded apps:

Queue of work to be done: 



As threads finish work at their own rate, they grab the next item in queue.

Common for “embarrassingly parallel” algorithms.


Works across the network too!

Thread Per Client

- Consider Web server:
 - Client connects
 - Client asks for a page:
 - <http://web.cs.swarthmore.edu/cs31>
 - Server looks through file system to find path (I/O)
 - Server sends back html for client browser (I/O)

What synchronization primitives do we need to serve each client with a dedicated thread?

Is there an advantage to using multiple threads in this example if we only have one CPU core?

Queue of work to be done: 

- Web server does this for MANY clients at once

Thread Pool



As threads finish work at their own rate, they grab the next item in queue.

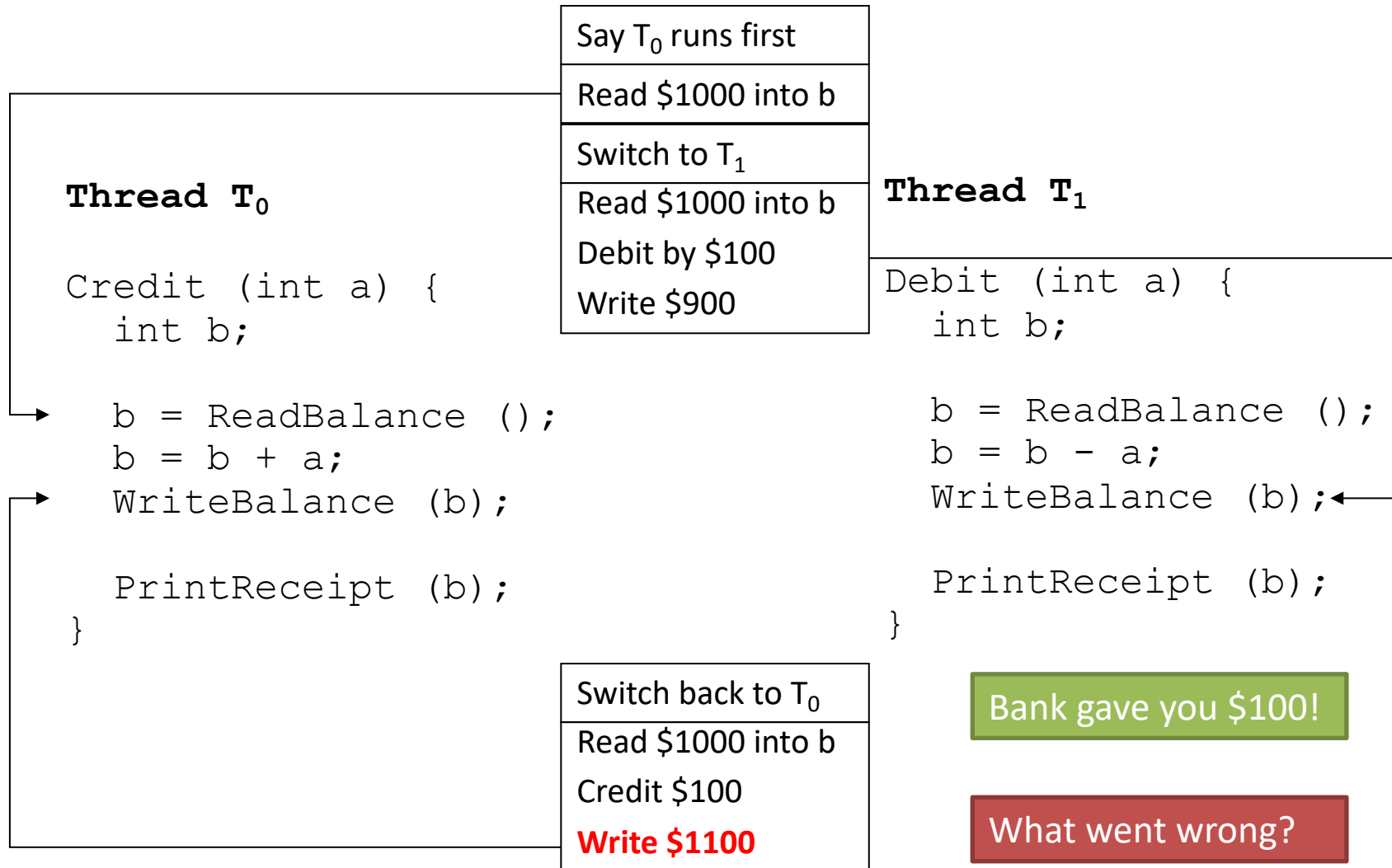
Common for “embarrassingly parallel” algorithms.

Works across the network too!

Thread Per Client

- Server “main” thread:
 - Wait for new connections
 - Upon receiving one, spawn new client thread
 - Continue waiting for new connections, repeat...
- Client threads:
 - Read client request, find files in file system
 - Send files back to client
 - Nice property: Each client is independent
 - Nice property: When a thread does I/O, it gets blocked for a while. OS can schedule another one.

Credit/Debit Problem: Race Condition



"Critical Section"

Thread T₀

```
Credit (int a) {  
    int b;
```

```
    b = ReadBalance ();  
    b = b + a;  
    WriteBalance (b);
```

```
    PrintReceipt (b);  
}
```

Danger Will Robinson!

Badness if
context
switch here!

Thread T₁

```
Debit (int a) {  
    int b;
```

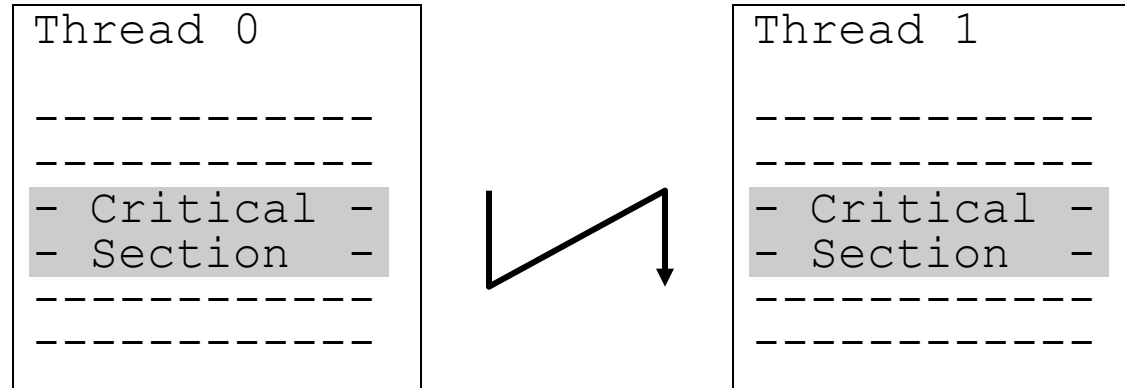
```
    b = ReadBalance ();  
    b = b - a;  
    WriteBalance (b);
```

```
    PrintReceipt (b);  
}
```

Bank gave you \$100!

What went wrong?

To Avoid Race Conditions

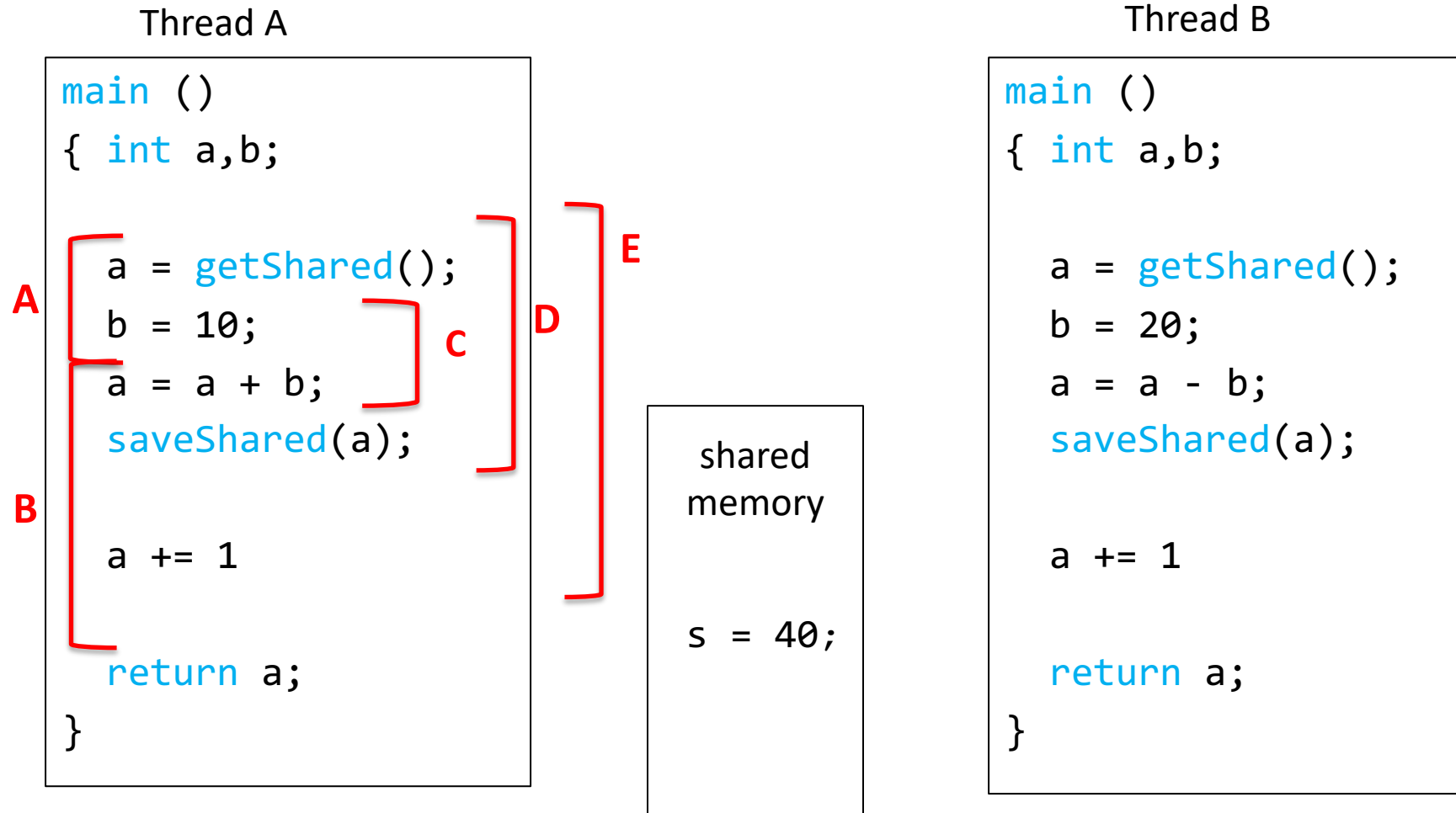


1. Identify critical sections
2. Use synchronization to **enforce mutual exclusion**
 - Only one thread active in a critical section

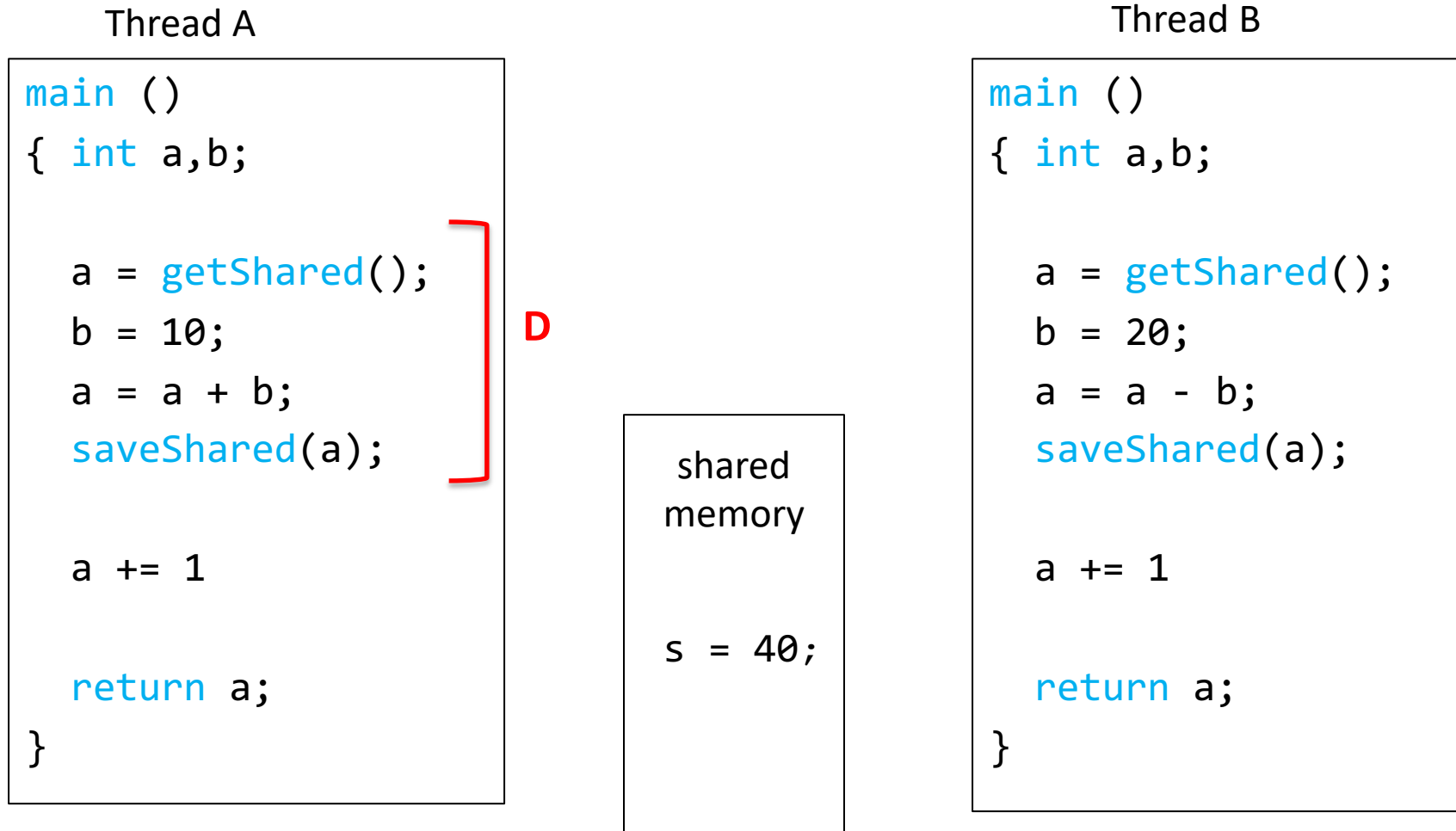
Critical Section and Atomicity

- Sections of code executed by multiple threads
 - **Access shared variables**, often making local copy
 - Places where order of execution or thread interleaving will affect the outcome
 - Follows: **read + modify + write** of shared variable
- Must run atomically with respect to each other
 - Atomicity: **runs as an entire instruction or not at all**. Cannot be divided into smaller parts.

Which code region is a critical section?



Which code region is a critical section? read + modify + write of shared variable



Large enough for correctness + Small enough to minimize slow down

Atomicity

- The implementation of acquiring/releasing critical section must be atomic.
 - An atomic operation is one which executes as though it could not be interrupted
 - Code that executes “all or nothing”
- How do we make them atomic?
 - Atomic instructions (e.g., test-and-set, compare-and-swap)
 - Allows us to build “semaphore” OS abstraction

Four Rules for Mutual Exclusion

1. No two threads can be inside their critical sections at the same time (**one of many but not more than one**).
2. No thread outside its critical section may prevent others from entering their critical sections.
3. **No thread should have to wait forever** to enter its critical section.
(Starvation)
4. No assumptions can be made about speeds or number of CPU's.



Railroad Semaphore

- Help trains figure out which track to be on at any given time.



Railroad Semaphore

- Help trains figure out which track to be on at any given time.

O.S. Semaphore:

- Construct that the OS provides to processes.
- Make system calls to modify their value

Mutual Exclusion with Semaphores

```
mutex = 1; //lock and unlock mutex atomically.
```

T₀

```
lock (mutex);
```

```
< critical section >
```

```
unlock (mutex);
```

T₁

```
lock (mutex);
```

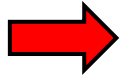
```
< critical section >
```

```
unlock (mutex);
```

Atomicity: run the entire instruction without interruption.

Mutual Exclusion with Semaphores

```
mutex = 1; //unlocked.
```



T₀

```
lock (mutex);
```

```
< critical section >
```

```
unlock (mutex);
```

T₁

```
lock (mutex);
```

```
< critical section >
```

```
unlock (mutex);
```

Atomicity: run the entire instruction without interruption.

T₀: Wants to execute the critical section

T₀: Reads the value of mutex,
Changes the value of mutex = 0 (acquires lock)
Enters critical section.

Mutual Exclusion with Semaphores

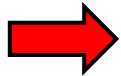
```
mutex = 0; //locked.
```

T₀

```
lock (mutex);
```

```
< critical section >
```

```
unlock (mutex);
```



T₁

```
lock (mutex);
```

```
< critical section >
```

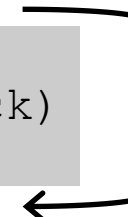
```
unlock (mutex);
```

Atomicity: run the entire instruction without interruption.

T₀: Wants to execute the critical section

```
T0: Reads the value of mutex,  
Changes the value of mutex = 0 (acquires lock)  
Enters critical section.
```

Atomic Execution



Mutual Exclusion with Semaphores

```
mutex = 0; //locked.
```

T₀

```
lock (mutex);
```

```
< critical section >
```

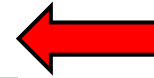
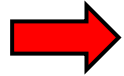
```
unlock (mutex);
```

T₁ (blocked)

```
lock (mutex);
```

```
< critical section >
```

```
unlock (mutex);
```



Atomicity: run the entire instruction without interruption.

T₀: In the critical section

T₁: Wants to enter the critical section.

 Reads the value of mutex (mutex = 0)

 Cannot enter critical section.

 Blocked.

Mutual Exclusion with Semaphores

```
mutex = 0; //locked.
```

T₀

```
lock (mutex);
```

```
< critical section >
```

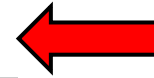
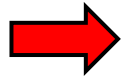
```
unlock (mutex);
```

T₁ (blocked)

```
lock (mutex);
```

```
< critical section >
```

```
unlock (mutex);
```



Atomicity: run the entire instruction without interruption.

T₀: Completes execution of critical section
Updates mutex value = 1. (release lock)

Mutual Exclusion with Semaphores

```
mutex = 1; //unlocked.
```

T₀

```
lock (mutex);
```

< critical section >

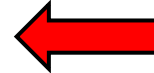
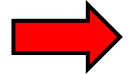
```
unlock (mutex);
```

T₁ (blocked)

```
lock (mutex);
```

< critical section >

```
unlock (mutex);
```



Atomicity: run the entire instruction without interruption.

T₀: Completes execution of critical section
Updates mutex value = 1. (release lock)



Atomic Execution

Mutual Exclusion with Semaphores

```
mutex = 1; //locked.
```

T₀

```
lock (mutex);
```

```
< critical section >
```

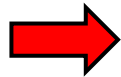
```
unlock (mutex);
```

T₁

```
lock (mutex);
```

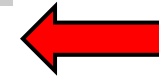
```
< critical section >
```

```
unlock (mutex);
```



Atomicity: run the entire instruction without interruption.

T₁: Can now acquire lock atomically and
Enter the critical section



Mutual Exclusion with Semaphores

```
mutex = 1; //lock and unlock mutex atomically.
```

T₀

```
lock (mutex);
```

```
< critical section >
```

```
unlock (mutex);
```

T₁

```
lock (mutex);
```

```
< critical section >
```

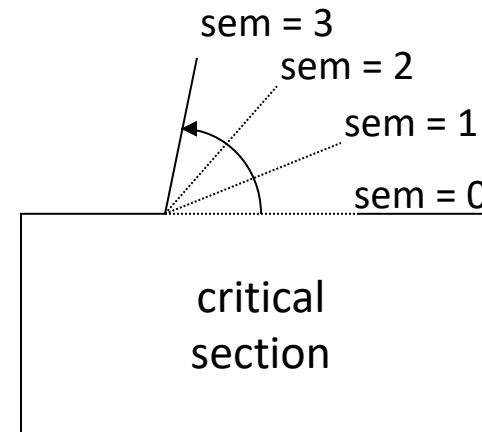
```
unlock (mutex);
```

- Use a “mutex” semaphore initialized to 1
- **Only one thread can enter critical section at a time.**
- Simple, works for any number of threads

Atomicity: runs as an entire instruction or not at all.

Semaphores

- Semaphore: OS synchronization variable
 - Has integer value
 - List of waiting threads
- Works like a gate
- If $sem > 0$, gate is open
 - Value equals number of threads that can enter
- Else, gate is closed
 - Possibly with waiting threads



Semaphore Operations

```
sem s = n; // declare and initialize

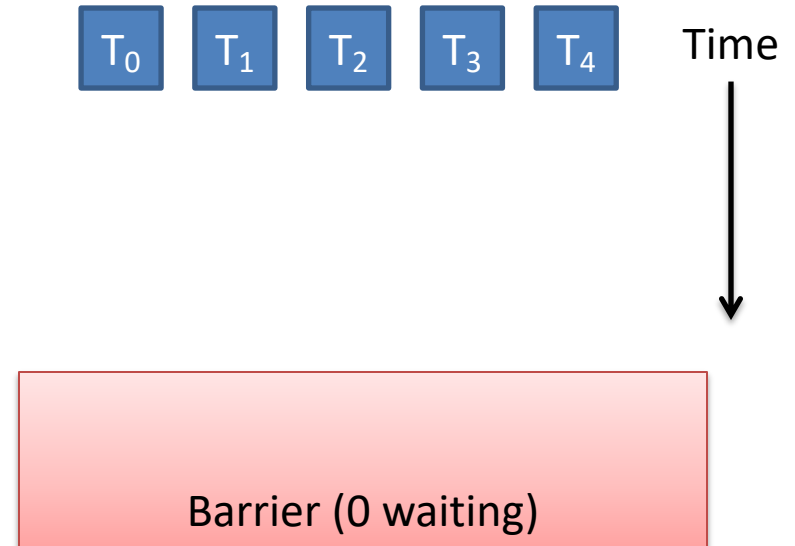
wait (sem s) // Executes atomically(*)
  decrement s;
  if s < 0:
    block thread (and associate with s);

signal (sem s) // Executes atomically(*)
  increment s;
  if blocked threads:
    unblock (any) one of them;
```

(*) With help from special hardware instructions.

Barrier Example, N Threads

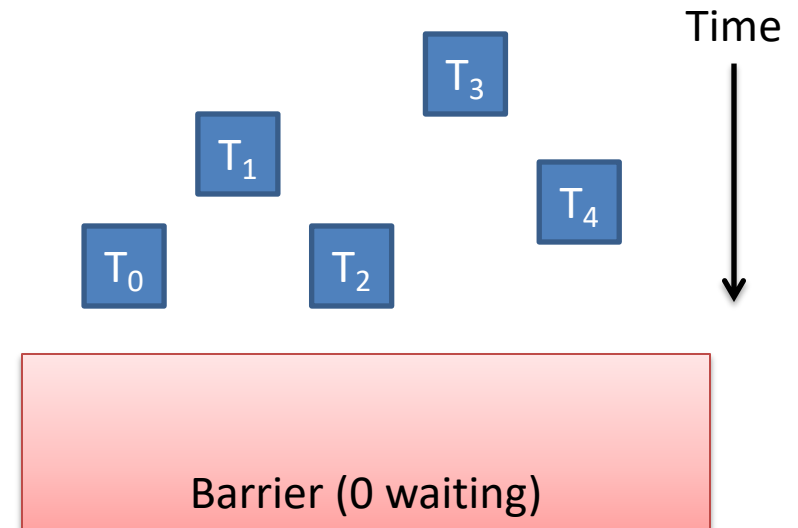
```
shared barrier b;  
  
init_barrier(&b, N);  
  
create_threads(N, func);  
  
void *func(void *arg) {  
    while (...) {  
        compute_sim_round()  
        barrier_wait(&b)  
    }  
}
```



Barrier Example, N Threads

```
shared barrier b;  
  
init_barrier(&b, N);  
  
create_threads(N, func);  
  
void *func(void *arg) {  
    while (...) {  
        compute_sim_round()  
        barrier_wait(&b)  
    }  
}
```

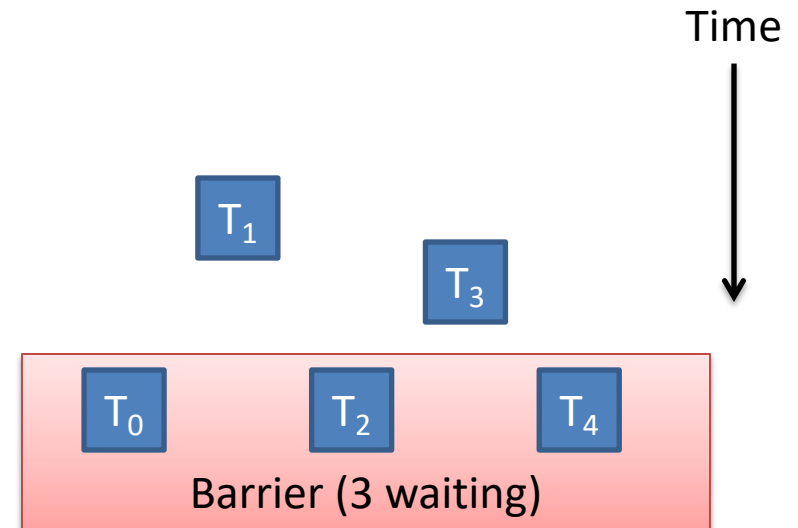
Threads make progress computing current round at different rates.



Barrier Example, N Threads

```
shared barrier b;  
  
init_barrier(&b, N);  
  
create_threads(N, func);  
  
void *func(void *arg) {  
    while (...) {  
        compute_sim_round()  
        barrier_wait(&b)  
    }  
}
```

Threads that make it to barrier must wait for all others to get there.



Barrier Example, N Threads

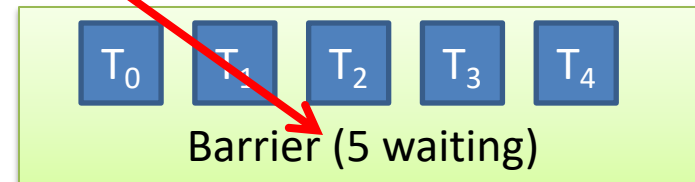
```
shared barrier b;  
  
init_barrier(&b, N);  
create_threads(N, func);  
  
void *func(void *arg) {  
    while (...) {  
        compute_sim_round()  
        barrier_wait(&b)  
    }  
}
```

Barrier allows threads to pass when N threads reach it.

Time



Matches

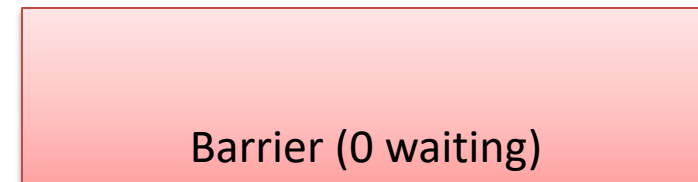


Barrier Example, N Threads

```
shared barrier b;  
  
init_barrier(&b, N);  
  
create_threads(N, func);  
  
void *func(void *arg) {  
    while (...) {  
        compute_sim_round()  
        barrier_wait(&b)  
    }  
}
```

Threads compute next round, wait on barrier again, repeat...

Time

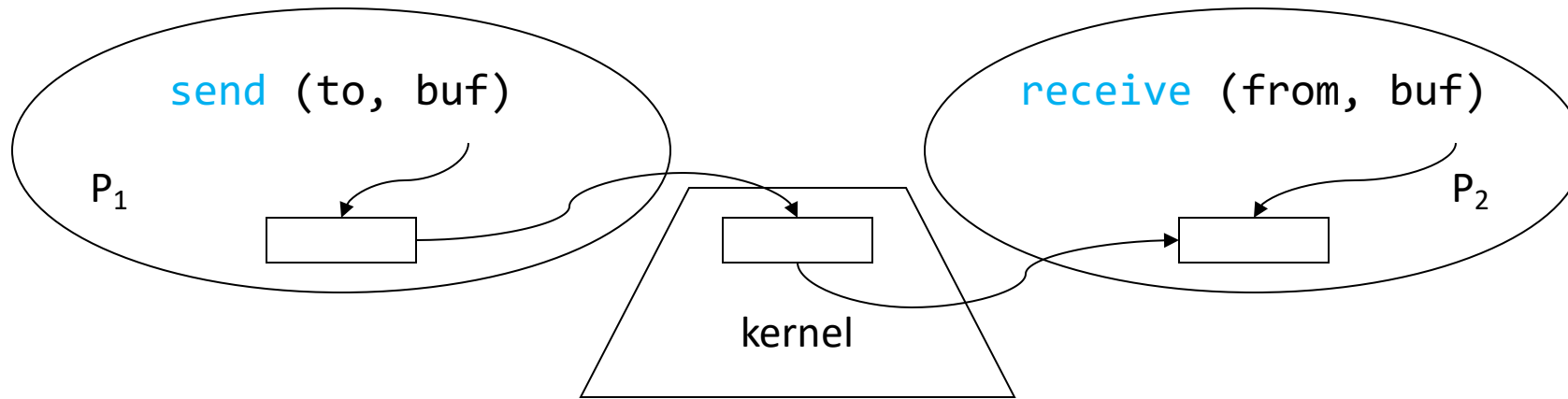


Synchronization: More than Mutexes

- I want all my threads to sync up at the same point.
 - Barrier: wait for everyone to catch up.
- I want to block a thread until something specific happens.
 - Condition variable: wait for a condition to be true
- I want my threads to share a critical section when they're reading, but still safely write.
 - Readers/writers lock: distinguish how lock is used

Synchronization: Beyond Mutexes

Message Passing

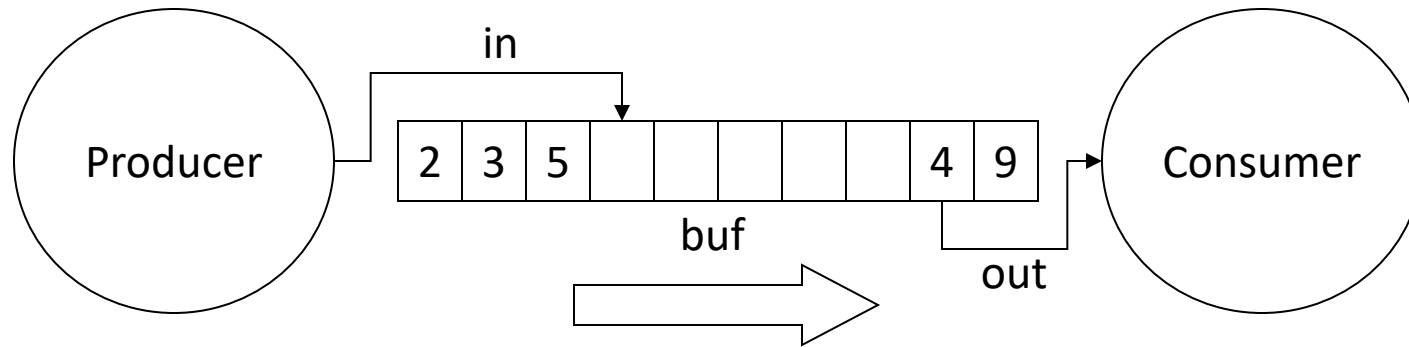


- Operating system mechanism for IPC
 - `send (destination, message_buffer)`
 - `receive (source, message_buffer)`
- Data transfer: in to and out of kernel message buffers
- Synchronization: can't receive until message is sent

Common Thread Patterns

- Producer / Consumer (a.k.a. Bounded buffer)
- Thread pool (a.k.a. work queue)
- Thread per client connection

The Producer/Consumer Problem



- Producer produces data, places it in shared buffer
- Consumer consumes data, removes from buffer
- Cooperation: Producer feeds Consumer
 - How does data get from Producer to Consumer?
 - How does Consumer wait for Producer?

Producer/Consumer: Shared Memory

```
shared int buf[N], in = 0, out = 0;
```

Producer

```
while (TRUE) {  
    buf[in] = Produce();  
    in = (in + 1)%N;  
}
```

Consumer

```
while (TRUE) {  
    Consume(buf[out]);  
    out = (out + 1)%N;  
}
```

Data transferred in shared memory buffer.

Producer/Consumer: Shared Memory

```
shared int buf[N], in = 0, out = 0;
```

Producer

```
while (TRUE) {  
    buf[in] = Produce();  
    in = (in + 1)%N;  
}
```

Consumer

```
while (TRUE) {  
    Consume(buf[out]);  
    out = (out + 1)%N;  
}
```

Data transferred in shared memory buffer.

- Is there a problem with this code?
 - A. Yes, this is broken.
 - B. No, this ought to be fine.

Adding Semaphores

```
shared int buf[N], in = 0, out = 0;  
shared sem filledslots = 0, emptyslots = N;
```

Producer

```
while (TRUE) {  
    wait (X);  
    buf[in] = Produce ();  
    in = (in + 1) % N;  
    signal (Y);  
}
```

Consumer

```
while (TRUE) {  
    wait (Z);  
    Consume (buf[out]);  
    out = (out + 1) % N;  
    signal (W);  
}
```

- Recall semaphores:
 - wait(): decrement sem and block if sem value < 0
 - signal(): increment sem and unblock a waiting process (if any)

Suppose we now have two semaphores to protect our array. Where do we use them?

```
shared int buf[N], in = 0, out = 0;  
shared sem filledslots = 0, emptyslots = N;
```

Producer

```
while (TRUE) {  
    wait (X);  
    buf[in] = Produce ();  
    in = (in + 1) % N;  
    signal (Y);  
}
```

Consumer

```
while (TRUE) {  
    wait (Z);  
    Consume (buf[out]);  
    out = (out + 1) % N;  
    signal (W);  
}
```

Answer choice	X	Y	Z	W
A.	emptyslots	emptyslots	filledslots	filledslots
B.	emptyslots	filledslots	filledslots	emptyslots
C.	filledslots	emptyslots	emptyslots	filledslots

Add Semaphores for Synchronization

```
shared int buf[N], in = 0, out = 0;  
shared sem filledslots = 0, emptyslots = N;
```

Producer

```
while (TRUE) {  
    wait (emptyslots);  
    buf[in] = Produce();  
    in = (in + 1) % N;  
    signal (filledslots);  
}
```

Consumer

```
while (TRUE) {  
    wait (filledslots);  
    Consume(buf[out]);  
    out = (out + 1) % N;  
    signal (emptyslots);  
}
```

- Buffer empty, Consumer waits
- Buffer full, Producer waits
- Don't confuse synchronization with mutual exclusion

Synchronization: More than Mutexes

- “I want to block a thread until something specific happens.”
 - Condition variable: wait for a condition to be true

Condition Variables

- In the pthreads library:
 - `pthread_cond_init`: Initialize CV
 - `pthread_cond_wait`: Wait on CV
 - `pthread_cond_signal`: Wakeup one waiter
 - `pthread_cond_broadcast`: Wakeup all waiters
- Condition variable is associated with a mutex:
 1. Lock mutex, realize conditions aren't ready yet
 2. Temporarily give up mutex until CV signaled
 3. Reacquire mutex and wake up when ready

Condition Variable Pattern

```
cond_one, cond_two
while (TRUE) {
    //independent code

    lock(m); //acquire the lock
    while (conditions bad)
        wait(m, cond_one); //implicitly release lock to other thread
                               while waiting for cond_one to be true.

    //proceed knowing that conditions are now good (as signaled by
    other thread)

    signal (cond_two); //your turn to signal a thread
                        waiting on cond_two
    unlock(m); //release the lock
}
```

Condition Variable Example

```
shared int buf[N], in = 0, out = 0;
shared int count = 0; // # of items in buffer
shared mutex m;
shared cond notempty, notfull;
```

Producer

```
while (TRUE) {
    item = Produce();

    lock(m);
    while (count == N)
        wait(m, notfull);

    buf[in] = item;
    in = (in + 1)%N;
    count += 1;

    signal (notempty);
    unlock(m);
}
```

Consumer

```
while (TRUE) {
    lock(m);
    while (count == 0)
        wait(m, notempty);

    item = buf[out];
    out = (out + 1)%N;
    count -= 1;

    signal (notfull);
    unlock(m);

    Consume(item);
}
```

Synchronization: More than Mutexes

- “I want to block a thread until something specific happens.”
 - Condition variable: wait for a condition to be true
- “I want all my threads to sync up at the same point.”
 - Barrier: wait for everyone to catch up.
- “I want my threads to share a critical section when they’re reading, but still safely write.”
 - Readers/writers lock: distinguish how lock is used

Summary

- Many ways to solve the same classic problems
 - Producer/Consumer: semaphores, CVs, messages
- There's more to synchronization than just mutual exclusion!
 - CVs, barriers, RWlocks, and others.

“Deadly Embrace”

- *The Structure of the THE-Multiprogramming System* (Edsger Dijkstra, 1968)
- Also introduced semaphores
- Deadlock is as old as synchronization

What is Deadlock?

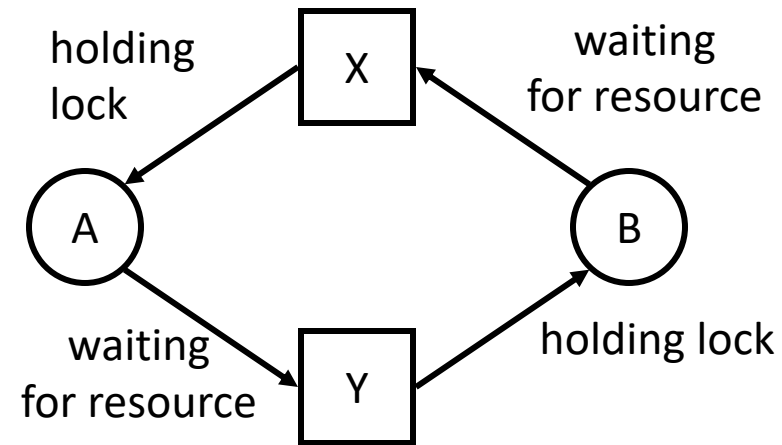
- Deadlock is a problem that can arise:
 - When processes compete for access to limited resources
 - When threads are incorrectly synchronized
- Definition:
 - Deadlock exists among a set of threads **if every thread is waiting for an event that can be caused only by another thread** in the set.

What is Deadlock?

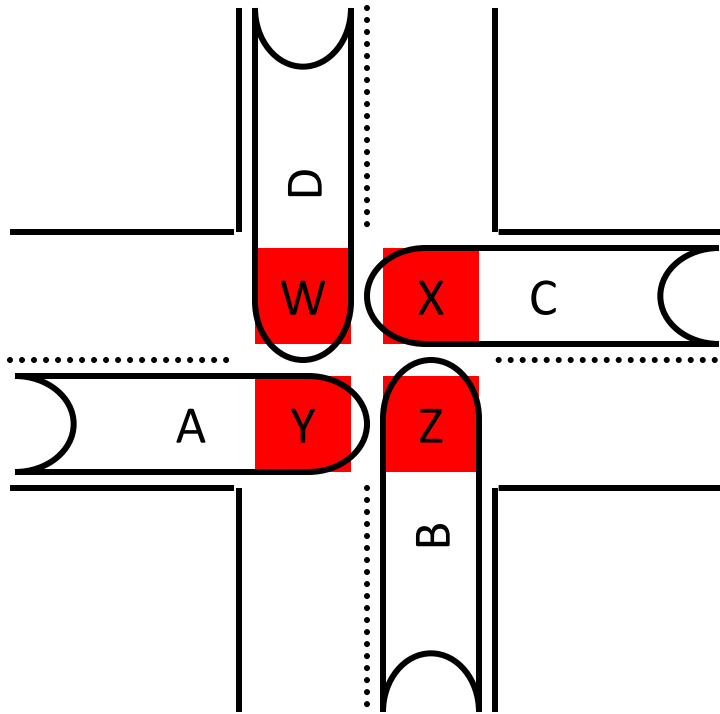
- Set of threads are permanently blocked
 - Unblocking of one relies on progress of another
 - But none can make progress!

- Example

- Threads A and B
- Resources X and Y
- A holding X, waiting for Y
- B holding Y, waiting for X
- Each is waiting for the other; will wait forever



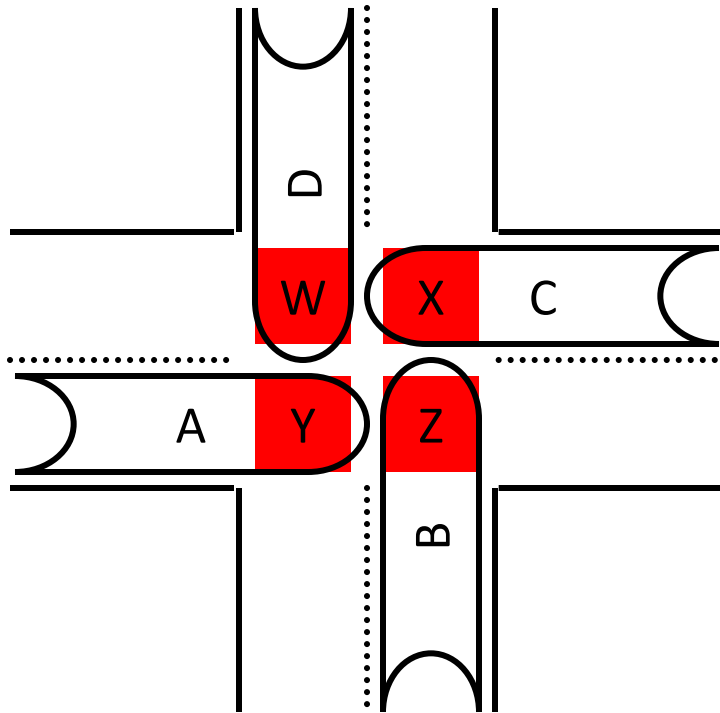
Traffic Jam as Example of Deadlock



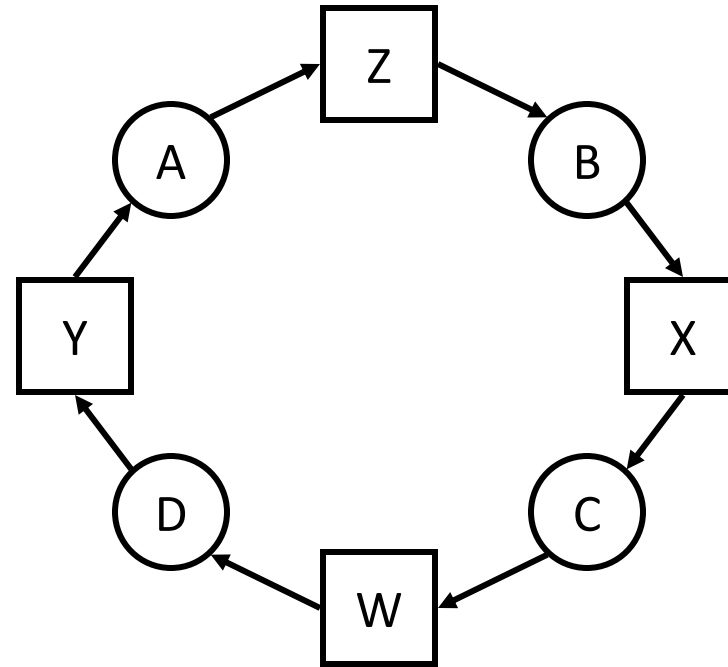
Cars deadlocked
in an intersection

- Cars A, B, C, D
- Road W, X, Y, Z
- Car A holds road space Y, waiting for space Z
- “Gridlock”

Traffic Jam as Example of Deadlock



Cars deadlocked
in an intersection



Resource Allocation
Graph

Four Conditions for Deadlock

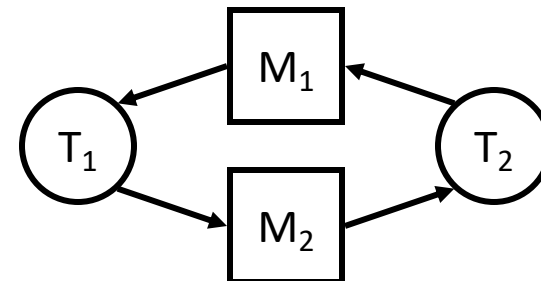
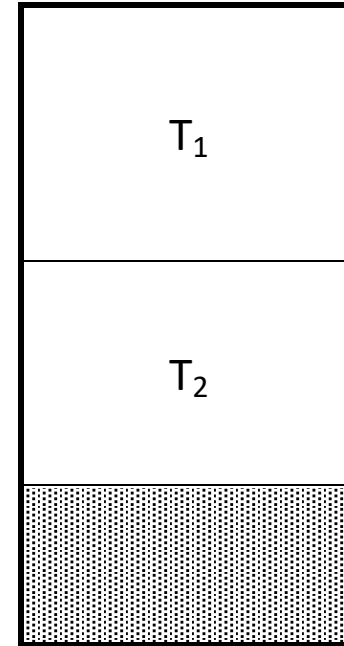
1. Mutual Exclusion
 - Only one thread may use a resource at a time.
2. Hold-and-Wait
 - Thread holds resource while waiting for another.
3. No Preemption
 - Can't take a resource away from a thread.
4. Circular Wait
 - The waiting threads form a cycle.

Four Conditions for Deadlock

1. Mutual Exclusion
 - Only one thread may use a resource at a time.
2. Hold-and-Wait
 - Thread holds resource while waiting for another.
3. No Preemption
 - Can't take a resource away from a thread.
4. Circular Wait
 - The waiting threads form a cycle.

Examples of Deadlock

- Memory (a reusable resource)
 - total memory = 200KB
 - T_1 requests 80KB
 - T_2 requests 70KB
 - T_1 requests 60KB (wait)
 - T_2 requests 80KB (wait)
- Messages (a consumable resource)
 - T_1 : receive M_2 from P_2
 - T_2 : receive M_1 from P_1



Banking, Revisited

```
struct account {  
    mutex lock;  
    int balance;  
}
```

```
Transfer(from_acct, to_acct, amt) {  
    lock(from_acct.lock);  
    lock(to_acct.lock);  
  
    from_acct.balance -= amt;  
    to_acct.balance += amt;  
  
    unlock(to_acct.lock);  
    unlock(from_acct.lock);  
}
```

If multiple threads are executing this code, is there a race? Could a deadlock occur?

```
struct account {  
    mutex lock;  
    int balance;  
}
```

```
Transfer(from_acct, to_acct, amt) {  
    lock(from_acct.lock);  
    lock(to_acct.lock);  
  
    from_acct.balance -= amt;  
    to_acct.balance += amt;  
  
    unlock(to_acct.lock);  
    unlock(from_acct.lock);  
}
```

If there's potential for a race/deadlock, what execution ordering will trigger it?

Clicker Choice	Potential Race?	Potential Deadlock?
A	No	No
B	Yes	No
C	No	Yes
D	Yes	Yes

Common Deadlock

Thread 0

```
Transfer(acctA, acctB, 20);
```

```
Transfer(...) {  
    lock(acctA.lock);  
    lock(acctB.lock);
```

Thread 1

```
Transfer(acctB, acctA, 40);
```

```
Transfer(...) {  
    lock(acctB.lock);  
    lock(acctA.lock);
```

Common Deadlock

Thread 0

```
Transfer(acctA, acctB, 20);
```

```
Transfer(...) {
```

```
    lock(acctA.lock);
```

T₀ gets to here

```
    lock(acctB.lock);
```

Thread 1

```
Transfer(acctA, acctB, 40);
```

```
Transfer(...) {
```

```
    lock(acctB.lock);
```

T₁ gets to here

```
    lock(acctA.lock);
```

T₀ holds A's lock, will make no progress until it can get B's.
T₁ holds B's lock, will make no progress until it can get A's.

How to solve the Deadlock Problem

What should your OS do to help you?

- Deadlock Prevention
 - Make deadlock impossible by removing a condition
- Deadlock Avoidance (“Banker’s Algorithm”)
 - Avoid getting into situations that lead to deadlock
- Deadlock Detection
 - Don’t try to stop deadlocks
 - Rather, if they happen, detect and resolve

Which type of deadlock-handling scheme would you expect to see in a modern OS (Linux/Windows/OS X) ?

- A. Deadlock prevention
- B. Deadlock avoidance
- C. Deadlock detection/recovery
- D. Something else



“Ostrich Algorithm”

How to Attack the Deadlock Problem

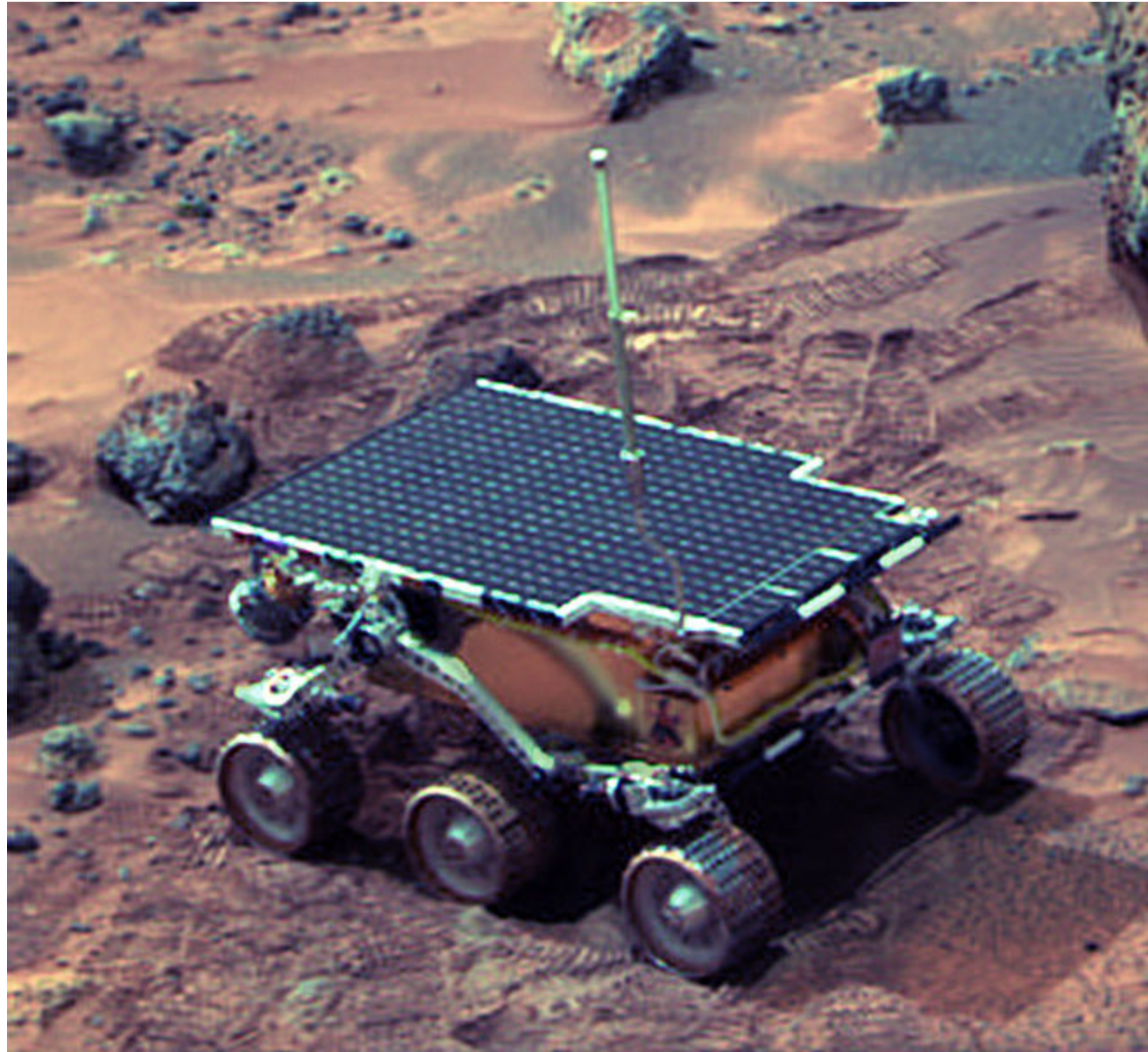
- Deadlock Prevention
 - Make deadlock impossible by removing a condition
- Deadlock Avoidance
 - Avoid getting into situations that lead to deadlock
- Deadlock Detection
 - Don't try to stop deadlocks
 - Rather, if they happen, detect and resolve
- These all have major drawbacks...

Other Thread Complications

- Deadlock is not the only problem
- Performance: too much locking?
- Priority inversion
- ...

Priority Inversion

- Problem: Low priority thread holds lock, high priority thread waiting for lock.
 - What needs to happen: boost low priority thread so that it can finish, release the lock
 - What sometimes happens in practice: low priority thread not scheduled, can't release lock
- Example: Mars Pathfinder (1997)



Sojourner Rover on Mars

Mars Rover

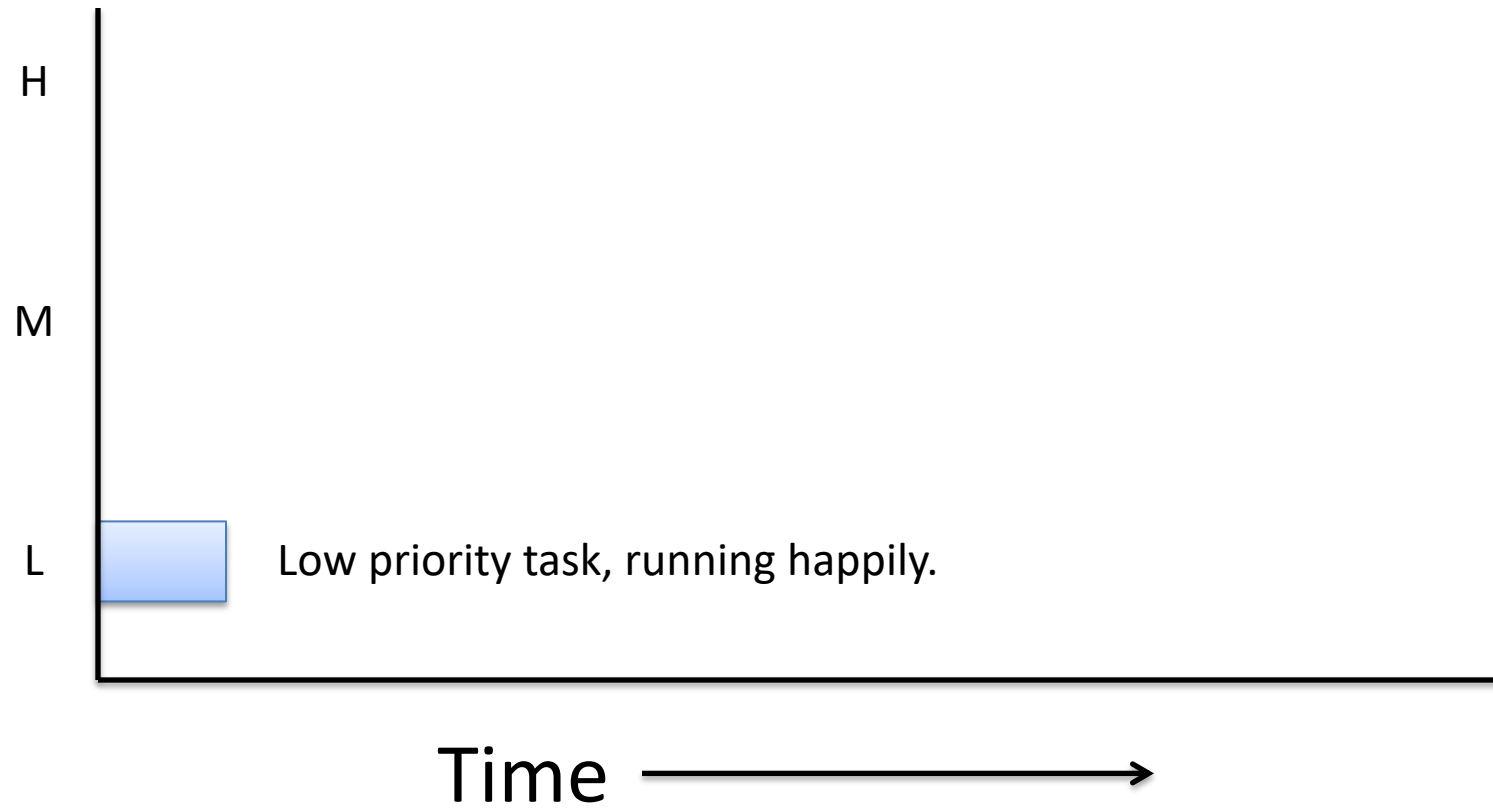
- Three periodic tasks:
 1. Low priority: collect meteorological data
 2. Medium priority: communicate with NASA
 3. High priority: data storage/movement
- Tasks 1 and 3 require exclusive access to a hardware bus to move data.
 - Bus protected by a mutex.

Mars Rover

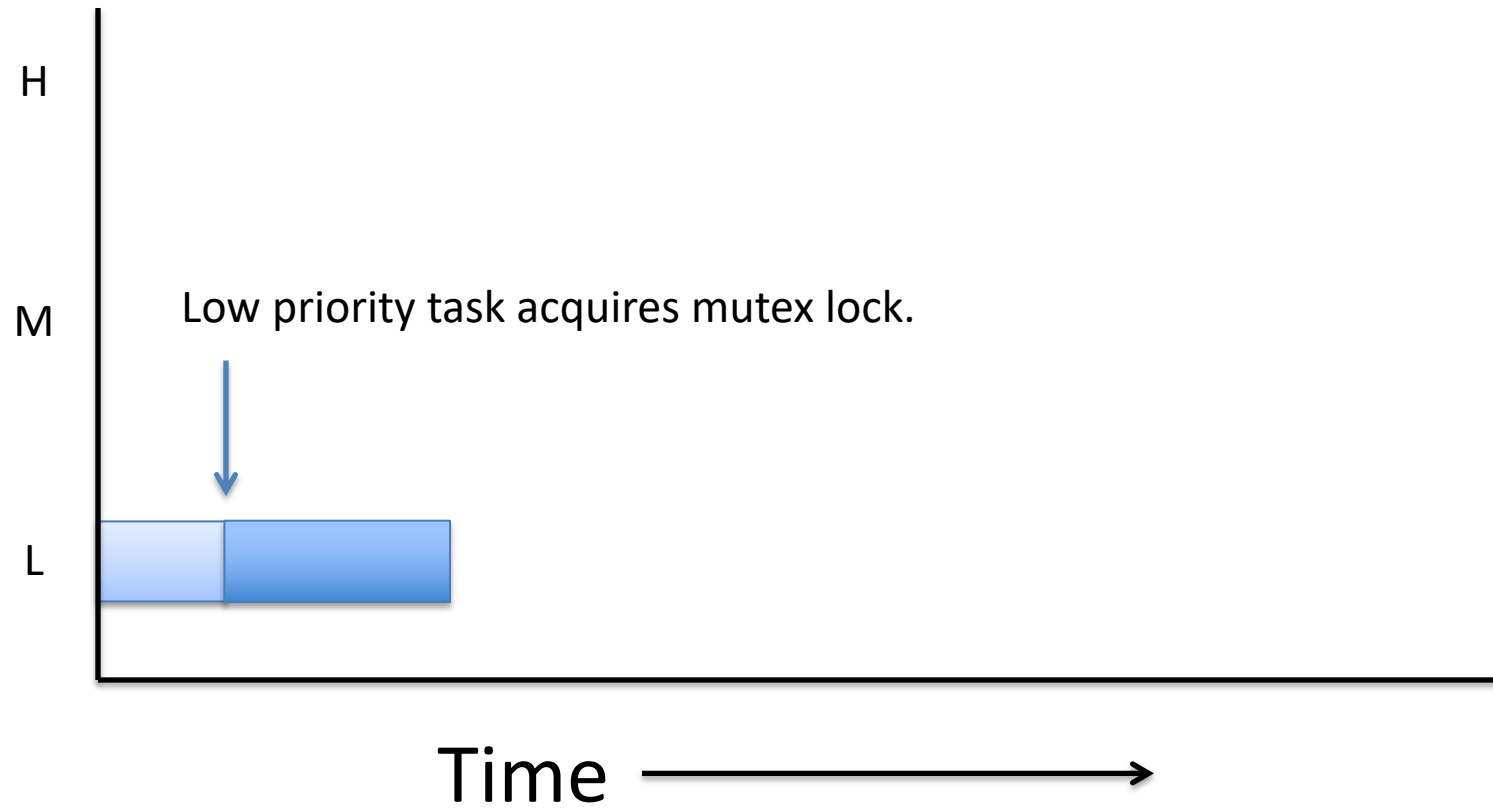
- Failsafe timer (watchdog): if high priority task doesn't complete in time, reboot system
- Observation: uh-oh, this thing seems to be rebooting a lot, we're losing data...

JPL engineers later confessed that one or two system resets had occurred in their months of pre-flight testing. They had never been reproducible or explainable, and so the engineers, in a very human-nature response of denial, decided that they probably weren't important, using the rationale "it was probably caused by a hardware glitch".

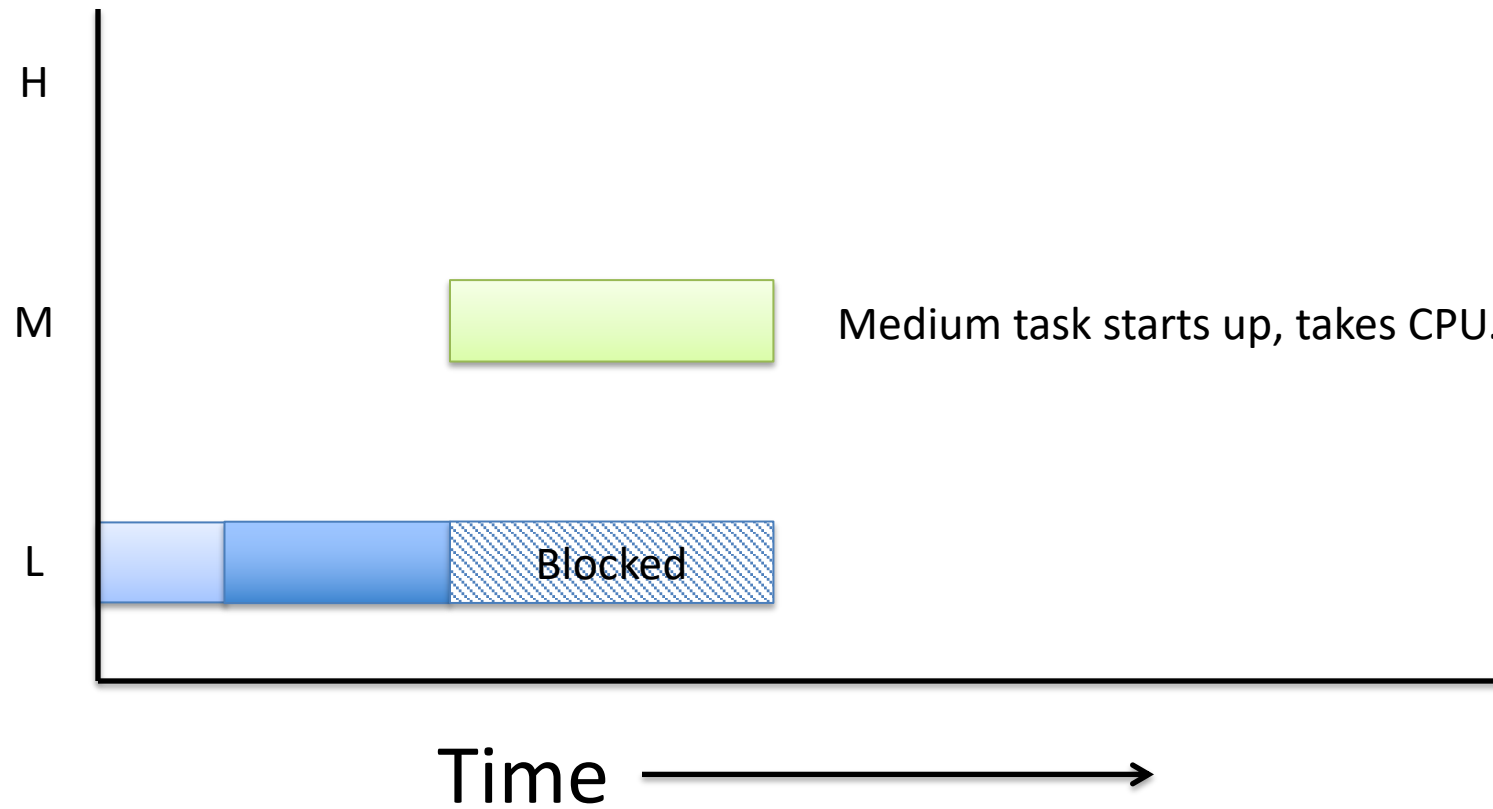
What Happened: Priority Inversion



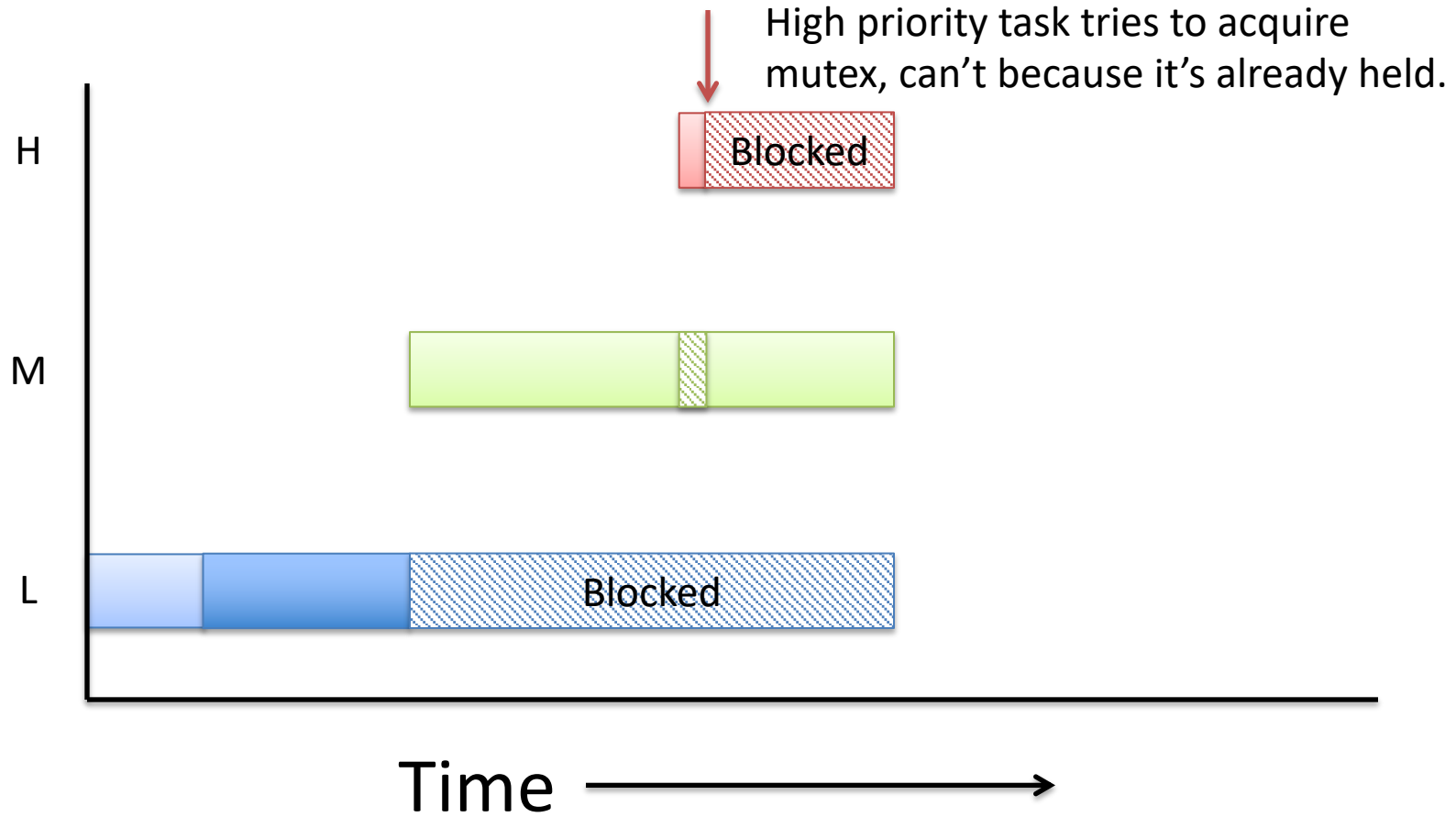
What Happened: Priority Inversion



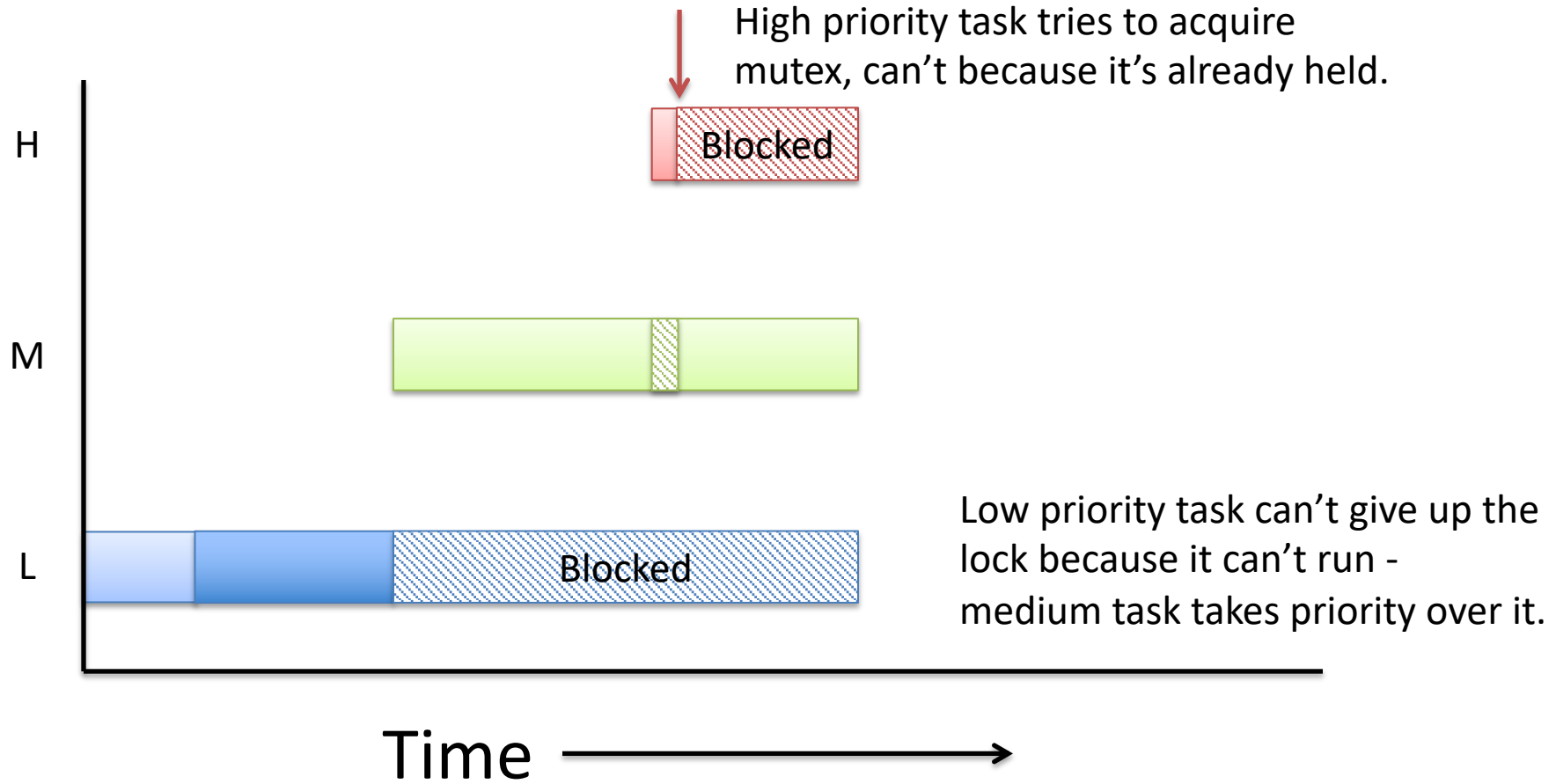
What Happened: Priority Inversion



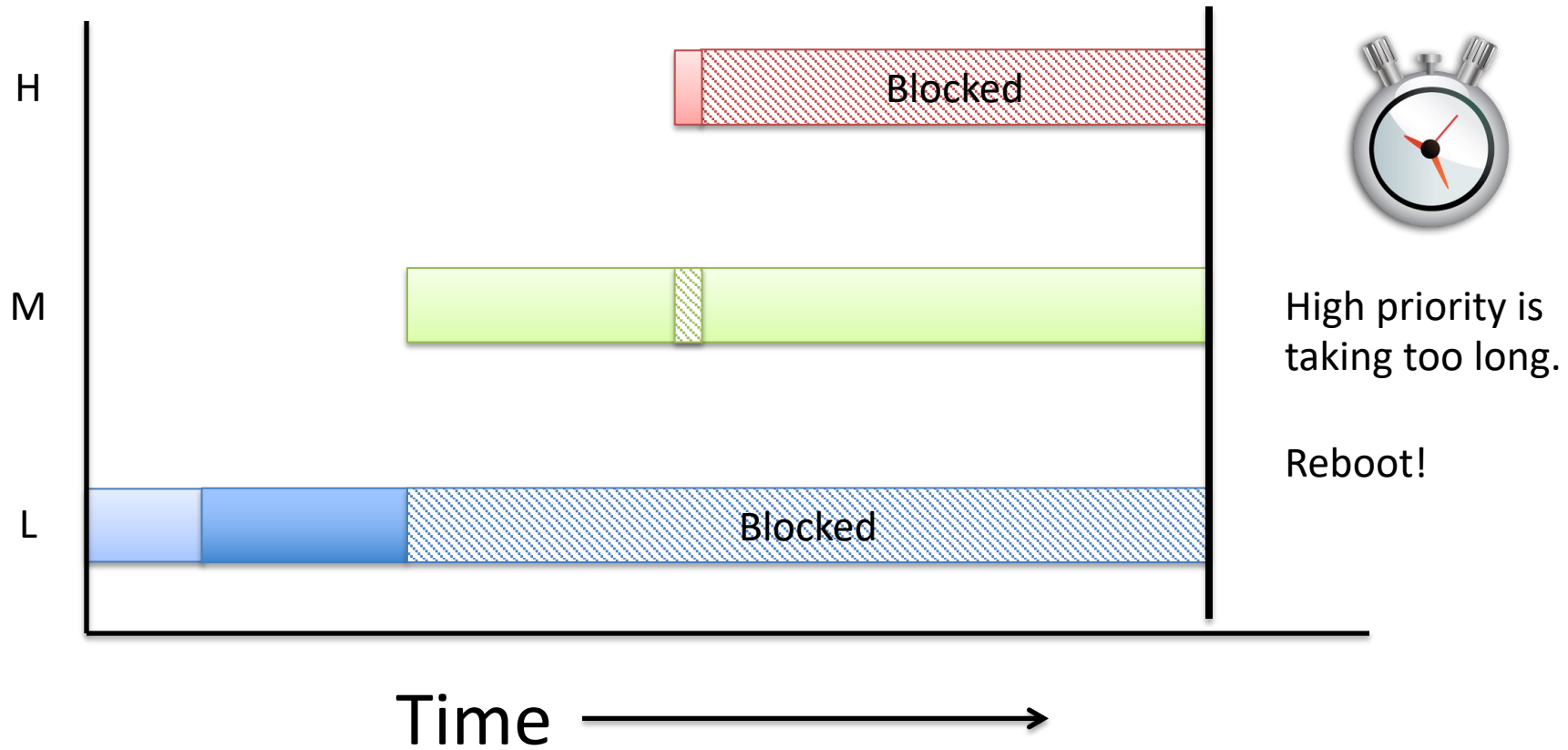
What Happened: Priority Inversion



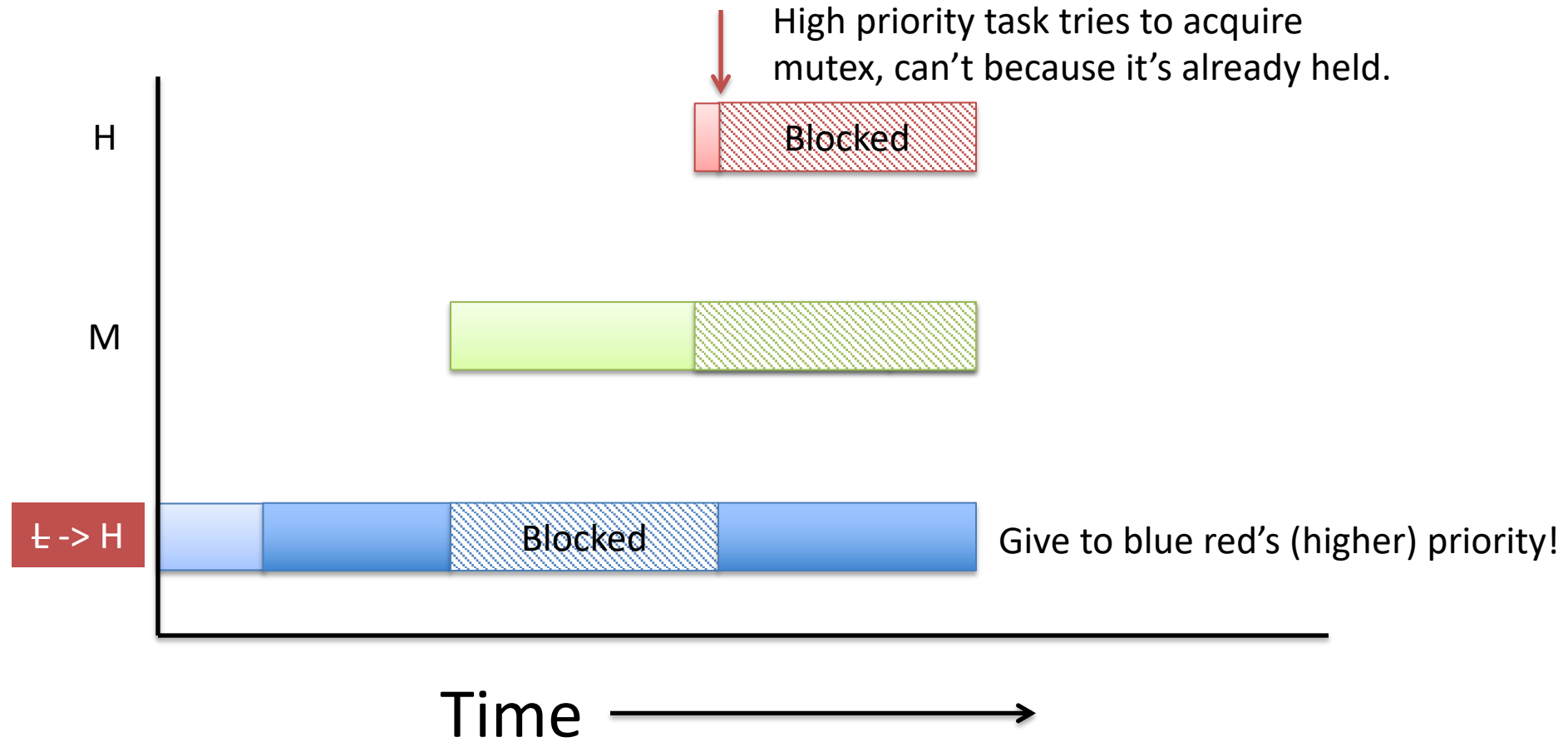
What Happened: Priority Inversion



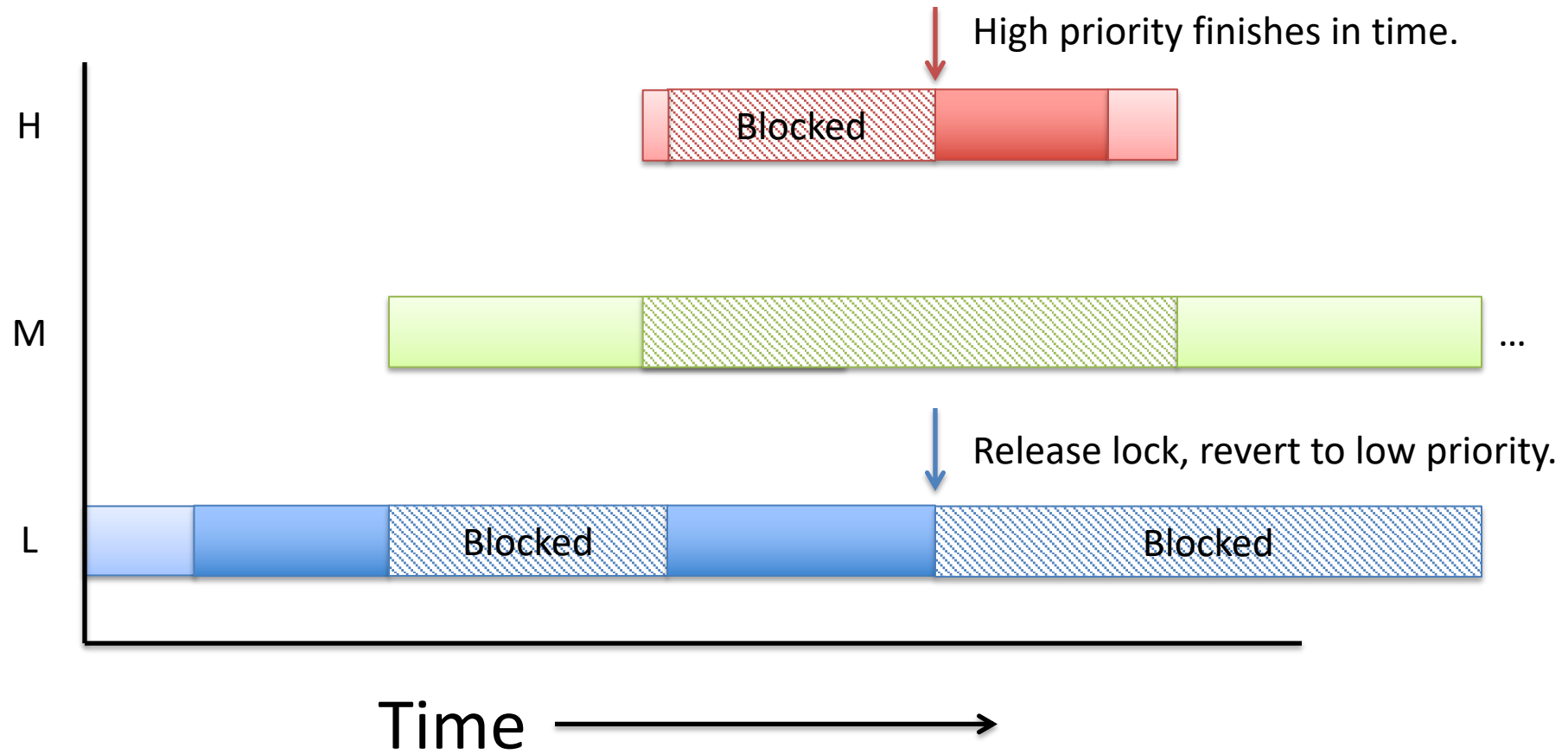
What Happened: Priority Inversion



Solution: Priority Inheritance



Solution: Priority Inheritance



Deadlock Summary

- Deadlock occurs when threads are waiting on each other and cannot make progress.
- Deadlock requires four conditions:
 - Mutual exclusion, hold and wait, no resource preemption, circular wait
- Approaches to dealing with deadlock:
 - Ignore it – Living life on the edge (most common!)
 - Prevention – Make one of the four conditions impossible
 - Avoidance – Banker's Algorithm (control allocation)
 - Detection and Recovery – Look for a cycle, preempt/abort

The image features a central graphic consisting of several concentric circles. The innermost circle is a dark blue color. Surrounding it are several rings of varying shades of red, from a deep, dark red to a lighter, more vibrant red. The text "That's all Folks!" is written in a white, elegant cursive script across the middle of the graphic, overlapping the blue circle and the red rings. The background of the entire image is a solid dark red color.

That's all Folks!