CS 31: Introduction to Computer Systems

24-25: Race Conditions and Synchronization

April 23 - 25, 2019

SWARTHMORE COLLEGE

Recap

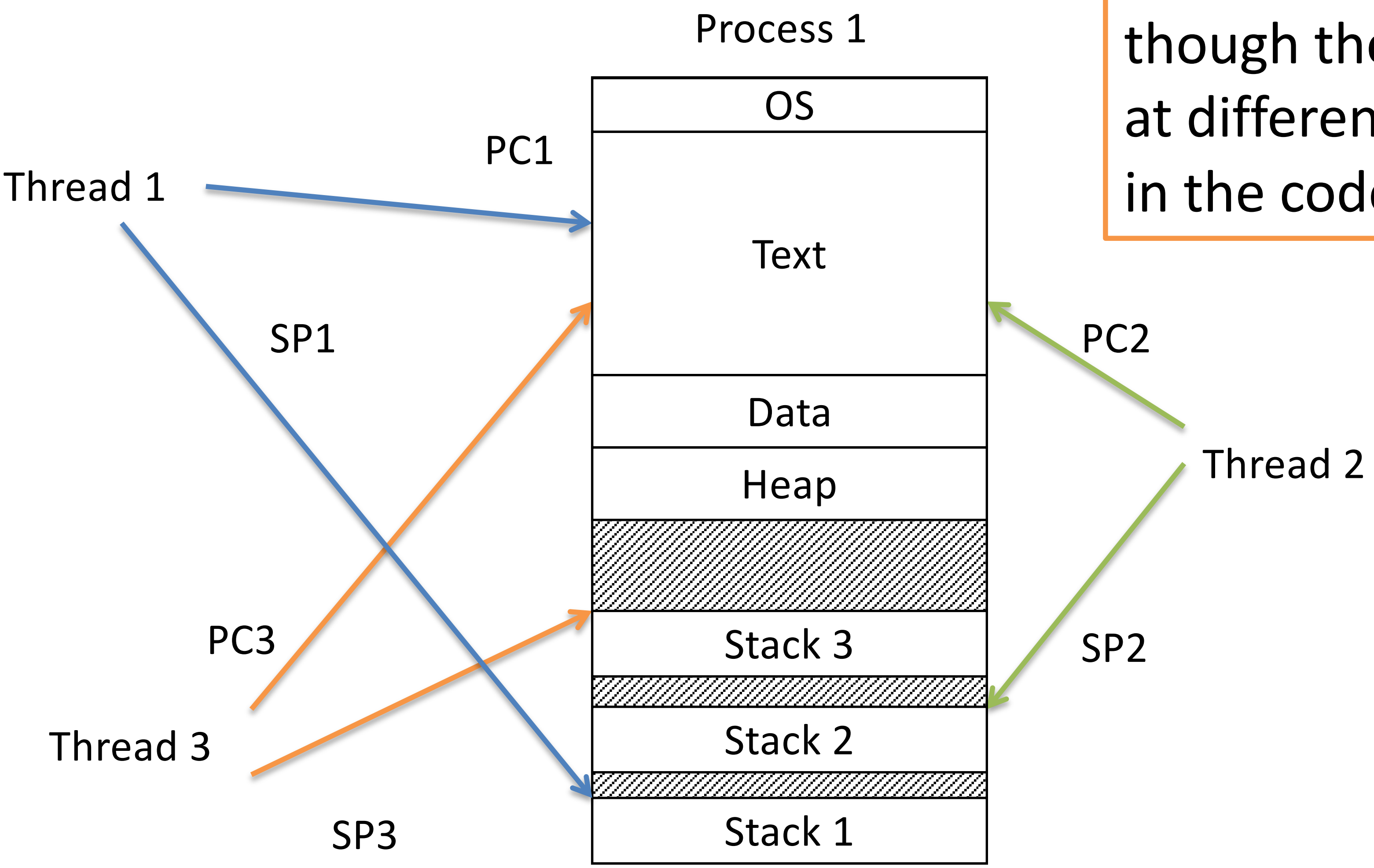
- To speed up a job, must divide it across multiple cores.

- Thread: abstraction for execution within process.
  - Threads share process memory.
  - Threads may need to communicate to achieve goal

- Thread communication:
  - To solve task (e.g., neighbor GOL cells)
  - To prevent bad interactions (synchronization)

Threads

They're all executing the same program (shared instructions in text), though they may be at different points in the code.

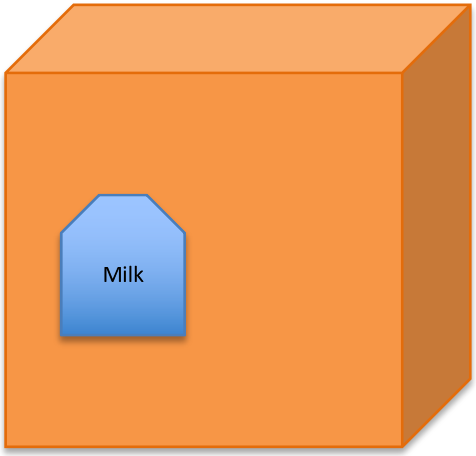


Synchronization

- Synchronize: to (arrange events to) happen such that two events do not overwrite each other's work.
- Thread synchronization
  - When one thread has to wait for another
  - Events in threads that occur "at the same time"
- Uses of synchronization
  - Prevent race conditions
  - Wait for resources to become available (only one thread has access at any time - deadlocks)

Synchronization:
Too Much Milk (TMM)

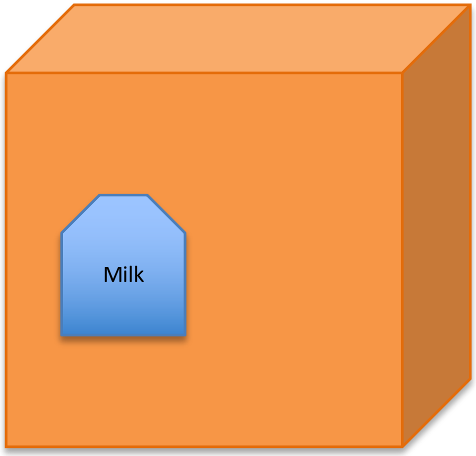
Time	You	Your Roommate
3.00	Arrive home	
3.05	Look in fridge, no milk	
3.10	Leave for the grocery store	
3.15		
3.20	Arrive at the grocery store	
3.25	Buy Milk	
3.30		
3.35	Arrive home, put milk in fridge	Arrive Home
3.40		Look in fridge, find milk
3.45		Cold Coffee (nom)



What mechanisms do we need for two independent threads to communicate and get a consistent view (computer state)?

How many cartons of milk can we have in this scenario? (Can we ensure this somehow?)

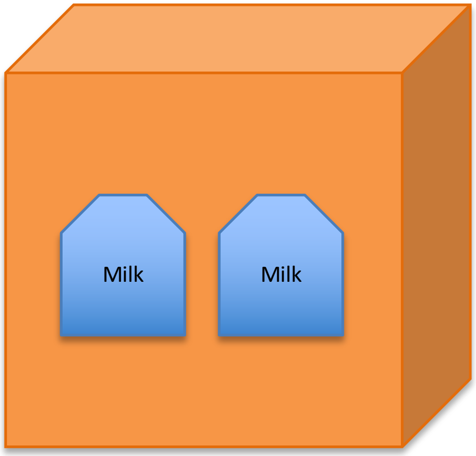
Time	You	Your Roommate
3.00	Arrive home	
3.05	Look in fridge, no milk	
3.10	Leave for the grocery store	
3.15		
3.20	Arrive at the grocery store	
3.25	Buy Milk	
3.30		
3.35	Arrive home, put milk in fridge	Arrive Home
3.40		Look in fridge, find milk
3.45		Cold Coffee (nom)



A. One carton (you)
B. Two cartons
C. No cartons
D. Something else

Synchronization:
Too Much Milk (TMM): One possible scenario

Time	You	Your Roommate
3.00	Arrive home	
3.05	Look in fridge, no milk	
3.10	Leave for grocery	Arrive Home
3.15		Look in fridge, no milk
3.20	Arrive at grocery	Leave for grocery
3.25	Buy Milk	
3.30		Arrive at grocery
3.35	Arrive home, put milk in fridge	
3.40		Arrive home, put milk in fridge
3.45		Oh No!



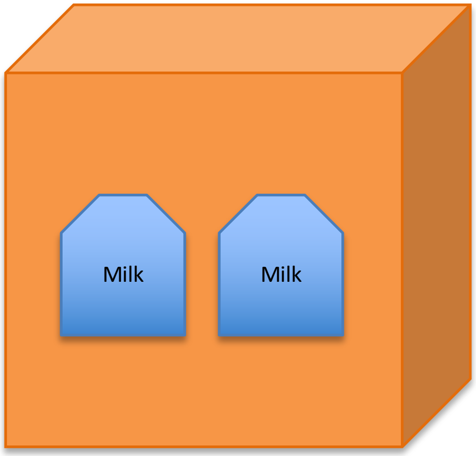
What mechanisms do we need for two independent threads to communicate and get a consistent view (computer state)?

Synchronization:

Threads get scheduled in an arbitrary manner:

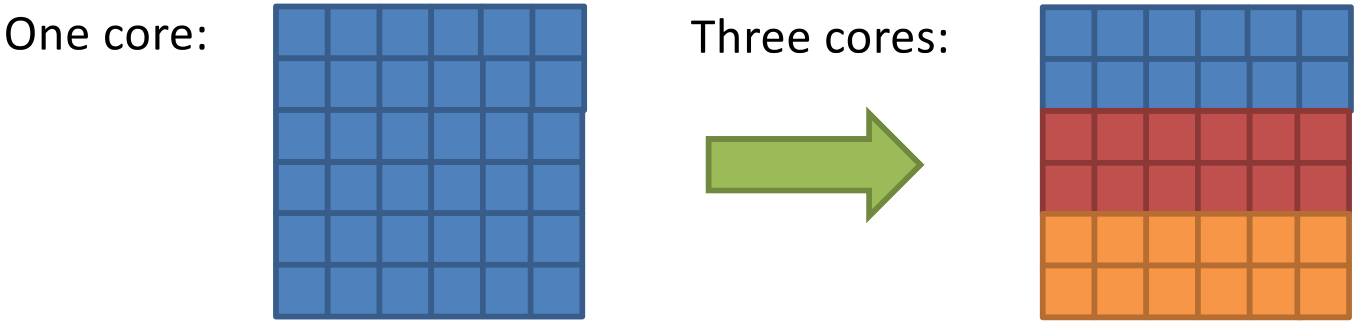
bad things may happen: ...or nothing may happen

Time	You	Your Roommate
3.00	Arrive home	
3.05	Look in fridge, no milk	
3.10	Leave for grocery	Arrive Home
3.15		Look in fridge, no milk
3.20	Arrive at grocery	Leave for grocery
3.25	Buy Milk	
3.30		Arrive at grocery
3.35	Arrive home, put milk in fridge	
3.40		Arrive home, put milk in fridge
3.45		Oh No!



What mechanisms do we need for two independent threads to communicate and get a consistent view (computer state)?

Synchronization Example



- Coordination required:
  - – Which thread goes first?
  - – Threads in different regions must work together to compute new value for boundary cells.
  - – Threads might not run at the same speed (depends on the OS scheduler). Can’t let one region get too far ahead.
  - – Context switches can happen at any time!

Thread Ordering

(Why threads require care. Humans aren't good at reasoning about this.)

- As a programmer you have *no idea* when threads will run. The OS schedules them, and the schedule will vary across runs.
- It might decide to context switch from one thread to another *at any time*.
- Your code must be prepared for this!
  - Ask yourself: "Would something bad happen if we context switched here?"
- hard to debug this problem if it is not reproducible

Example: The Credit/Debit Problem

- Say you have $1000 in your bank account
  - You deposit $100
  - You also withdraw $100

- How much should be in your account?

- What if your deposit and withdrawal occur at the same time, at different ATMs?

Credit/Debit Problem: Race Condition

Thread T_0

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

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

  PrintReceipt (b);
}
```

Thread T_1

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

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

  PrintReceipt (b);
}
```

Credit/Debit Problem: Race Condition

Say T_0 runs first

Read $1000 into b

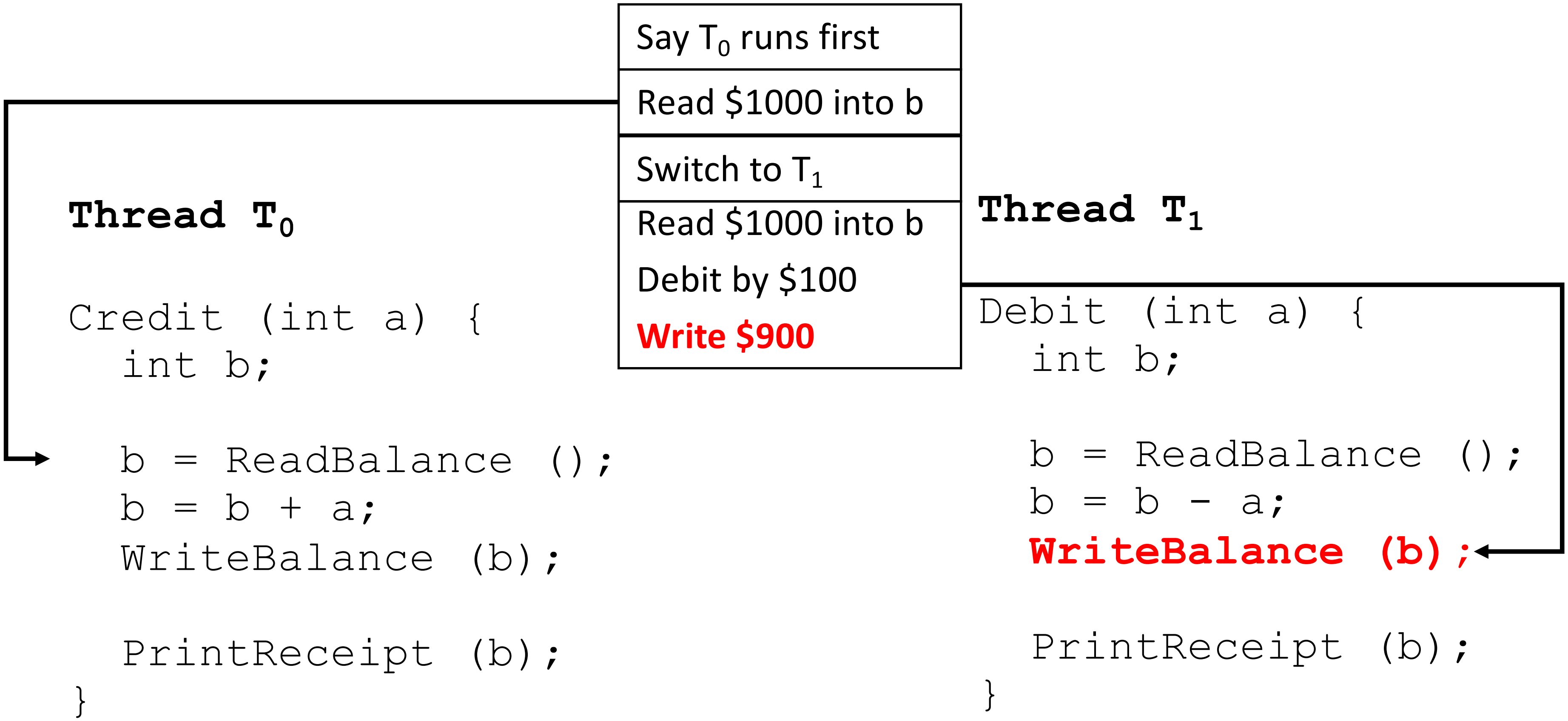
Thread T_0

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

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

  PrintReceipt (b);
}
```

Thread T_1

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

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

  PrintReceipt (b);
}
```

Credit/Debit Problem: Race Condition



Say T_0 runs first

Read $1000 into b

Switch to T_1

Read $1000 into b

Debit by $100

Write $900

Thread T_0

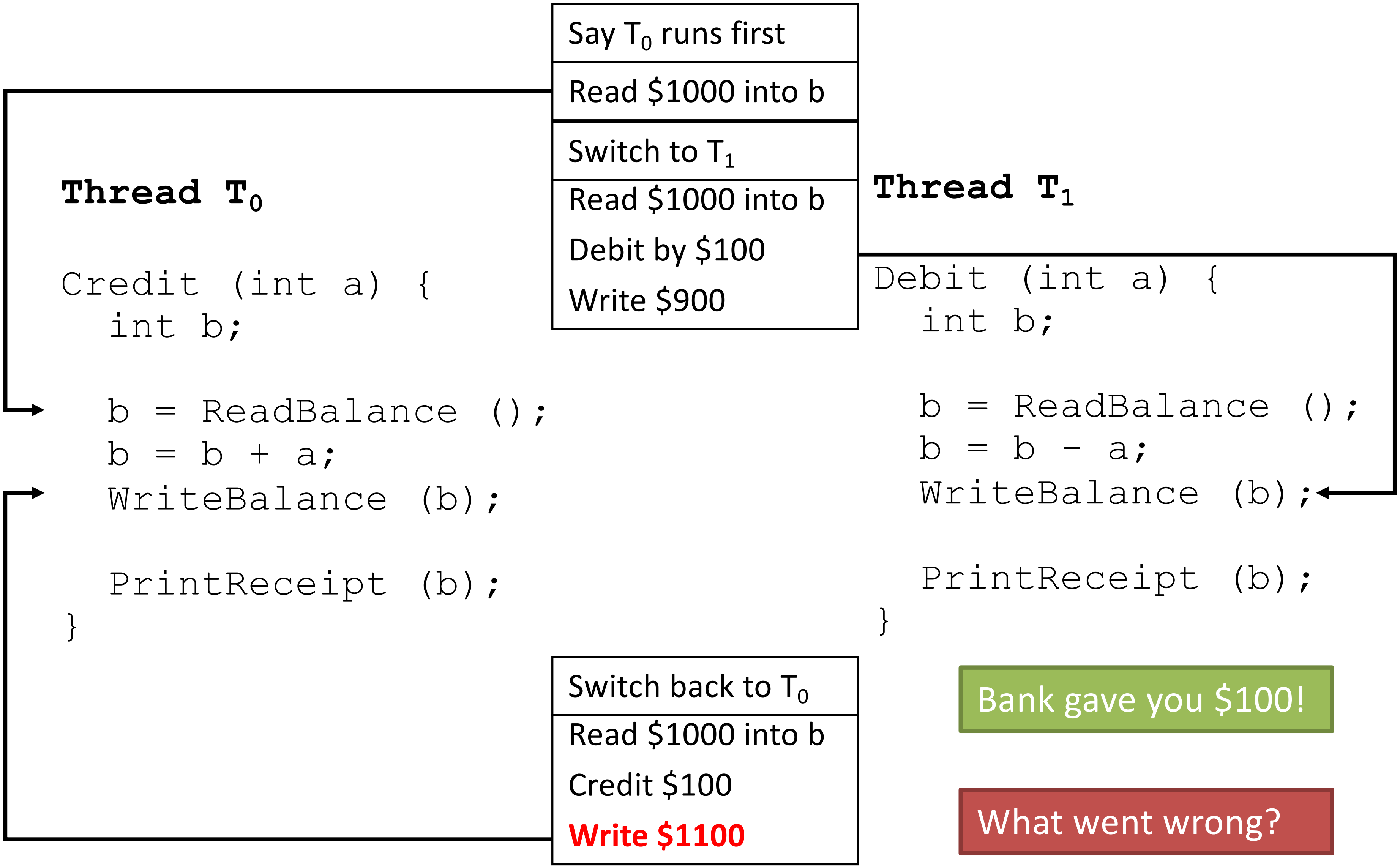
```
Credit (int a) {
  int b;

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

  PrintReceipt (b);
}
```

Thread T_1

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

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

  PrintReceipt (b);
}
```

CONTEXT SWITCH

Credit/Debit Problem: Race Condition



Say T_0 runs first

Read $1000 into b

Switch to T_1

Read $1000 into b

Debit by $100

Write $900

Thread T_0

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

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

  PrintReceipt (b);
}
```

Thread T_1

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

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

  PrintReceipt (b);
}
```

Switch back to T_0

Read $1000 into b

Credit $100

Write $1100

Bank gave you $100!

What went wrong?

“Critical Section”

Thread T_0

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

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

  PrintReceipt (b);
}
```

”Danger Will Robinson!

Badness
if context
switch
here!

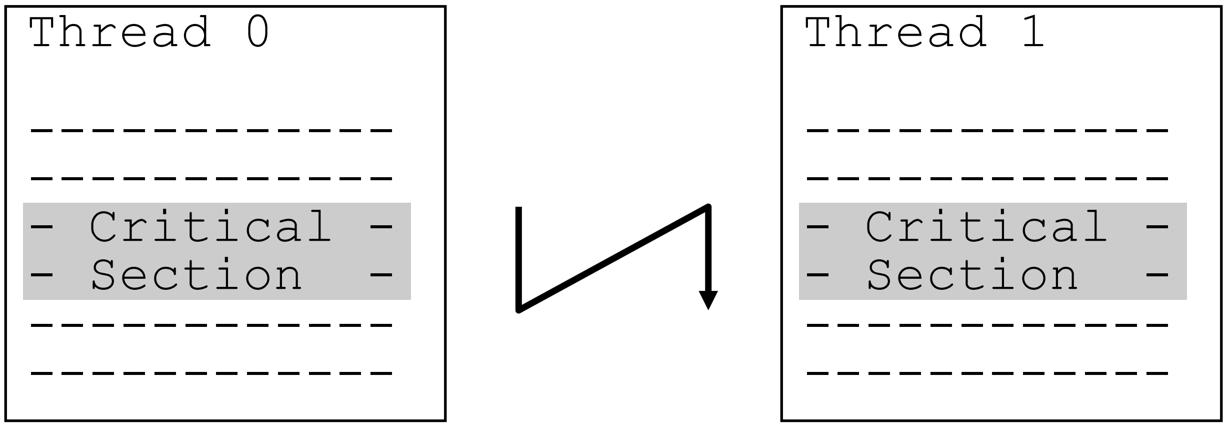
Thread T_1

```
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



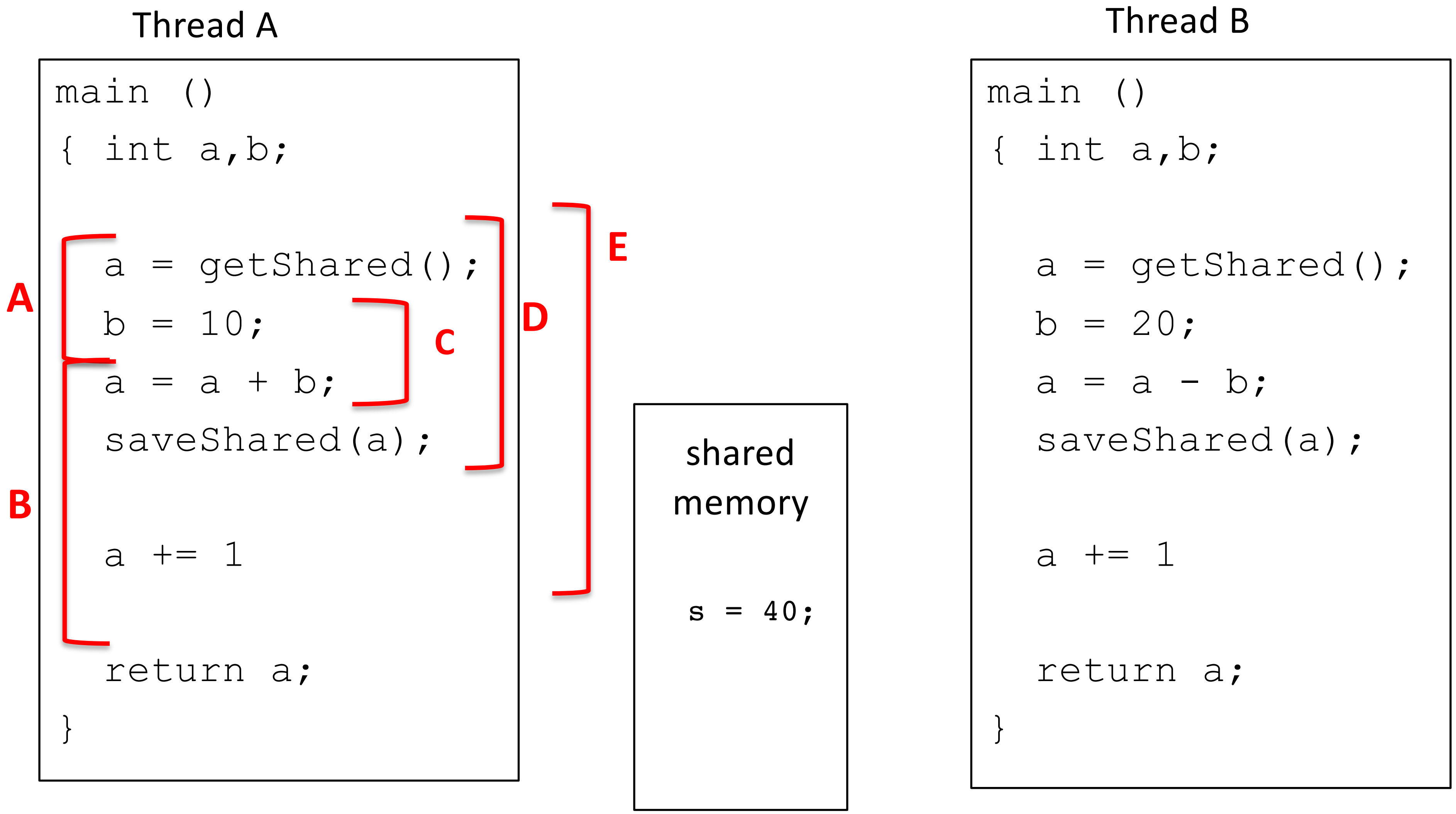
1. Identify critical sections

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

What Are Critical Sections?

- 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 unit or not at all. Cannot be divided into smaller parts.

Which code region is a critical section?



Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  a += 1

  return a;
}
```

shared memory

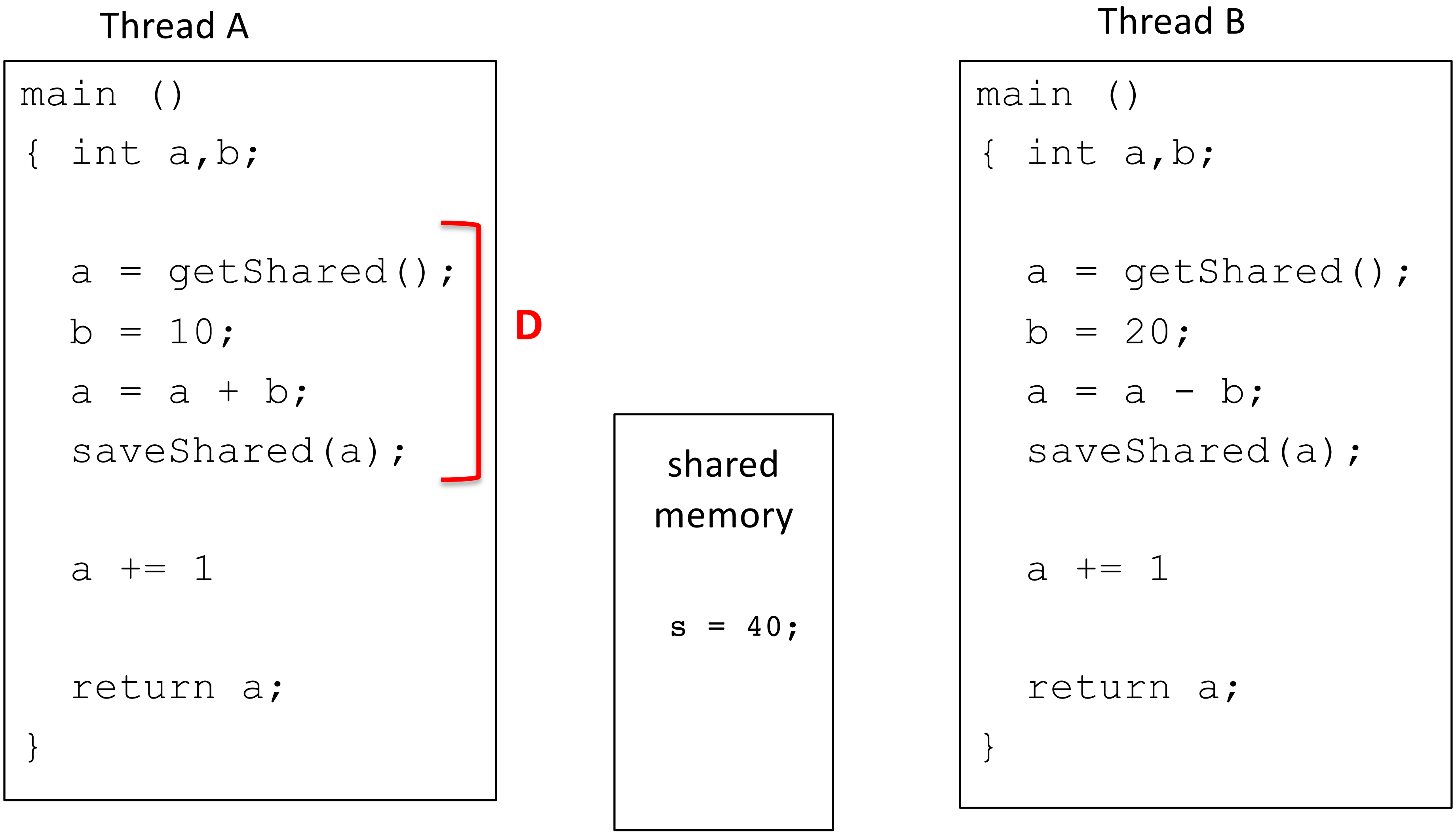
```
s = 40;
```

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  a += 1

  return a;
}
```

Which code region is a critical section?

read + modify + write of shared variable



Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  a += 1

  return a;
}
```

shared memory

```
s = 40;
```

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  a += 1

  return a;
}
```

Large enough for correctness + Small enough to minimize slow down

Which values might the shared s variable hold after both threads finish?

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  return a;
}
```

shared memory

```
s = 40;
```

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  return a;
}
```

If A runs first

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  return a;
}
```

shared memory

(s = 40)

s = 50

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  return a;
}
```

B runs after A Completes

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  return a;
}
```

shared memory

```
(s = 50)
 s = 30;
```

What about interleaving?

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  return a;
}
```

shared
memory

```
s = 40;
```

One of the threads will overwrite the other’s changes.

Is there a race condition?

Suppose `count` is a global variable (shared amongst threads), multiple threads increment it: `count++;`

A. Yes, there's a race condition (`count++` is a critical section).
B. No, there's no race condition (`count++` is not a critical section).
C. Cannot be determined

How about if compiler implements it as:

```
movl  (%edx), %eax    // read count value
addl  $1, %eax        // modify value
movl  %eax, (%edx)    // write count
```

How about if compiler implements it as:

```
incl  (%edx)          // increment value
```

Is there a race condition?

Suppose `count` is a global variable (shared amongst threads), multiple threads increment it: `count++;`

A. Yes, there's a race condition (`count++` is a critical section).

B. No, there's no race condition (`count++` is not a critical section).

C. Cannot be determined

How about if compiler implements it as:

```
movl  (%edx), %eax    // read count value
addl  $1, %eax        // modify value
movl  %eax, (%edx)    // write count
```

How about if compiler implements it as:

```
incl  (%edx)           // increment value
```

Neither of these instructions are implemented necessarily as atomic instruction!

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.

Thread Ordering

(Why threads require care. Humans aren't good at reasoning about this.)

- As a programmer you have *no idea* when threads will run. The OS schedules them, and the schedule will vary across runs.
- It might decide to context switch from one thread to another *at any time*.
- Your code must be prepared for this!
  - Ask yourself: "Would something bad happen if we context switched here?"
- hard to debug this problem if it is not reproducible

How to Achieve Mutual Exclusion?

```
< entry code >
< critical section >
< exit code >
```

```
< entry code >
< critical section >
< exit code >
```

- Surround critical section with entry/exit code
- Entry code should act as a gate
  - If another thread is in critical section, block
  - Otherwise, allow thread to proceed
- Exit code should release other entry gates

Possible Solution: Spin Lock?

```
shared int lock = OPEN;
```

T_0

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

T_1

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

- Lock indicates whether any thread is in critical section.

Note: While loop has no body. Keeps checking the condition as quickly as possible until it becomes false. (It “spins”)

Possible Solution: Spin Lock?

```
shared int lock = OPEN;
```

T_0

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

T_1

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

- Lock indicates whether any thread is in critical section.
- Is there a problem here?
  - A: Yes, this is broken.
  - B: No, this ought to work.

Possible Solution: Spin Lock?

```
shared int lock = OPEN;
```

T_0

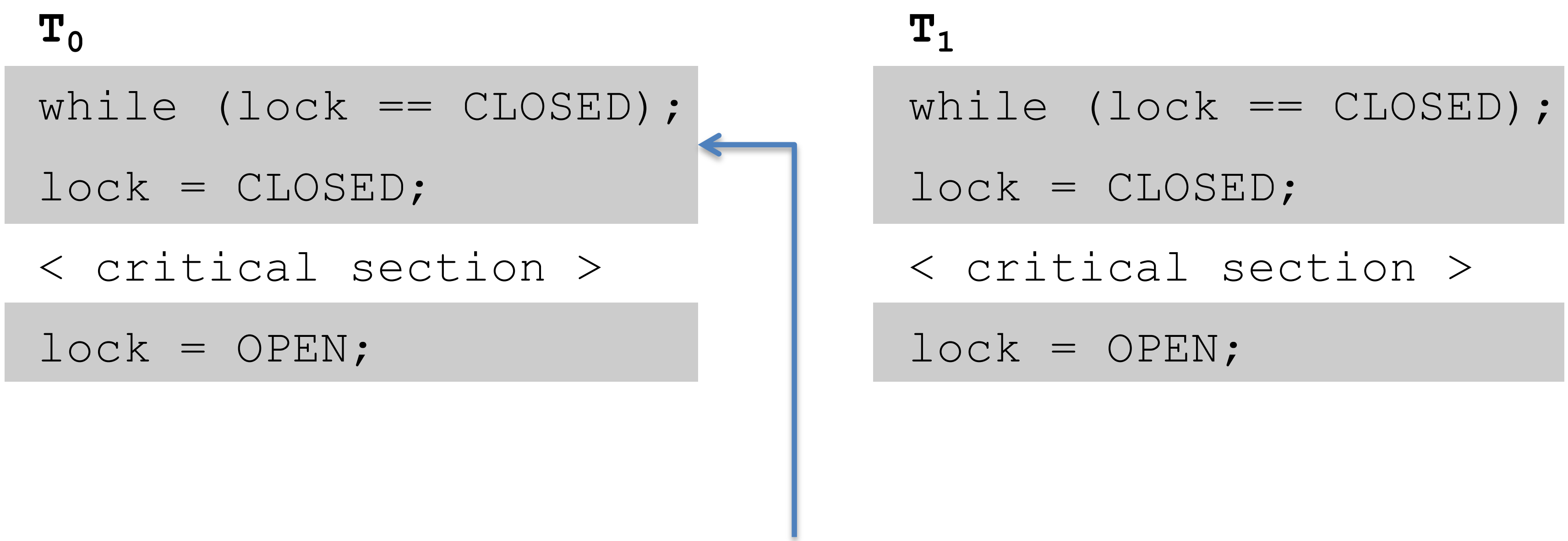
```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

T_1

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

- Lock indicates whether any thread is in critical section.
- Is there a problem here?
  - A: Yes, this is broken.
  - B: No, this ought to work.

Possible Solution: Spin Lock?



```
shared int lock = OPEN;
```

T_0

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

T_1

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

- What if a context switch occurs at this point?

Two statements: while lock is closed and setting of lock are not happening atomically. Race condition on updating the lock.

Possible Solution: Take Turns?

```
shared int turn = T0;
```

T_0

```
while (turn != T0);
< critical section >
turn = T1;
```

T_1

```
while (turn != T1);
< critical section >
turn = T0;
```

- Alternate which thread can enter critical section
- Is there a problem?
  - – A: Yes, this is broken.
  - – B: No, this ought to work.

Possible Solution: Take Turns?

```
shared int turn = T0;
```

T_0

```
while (turn != T0);
< critical section >
turn = T1;
```

T_1

```
while (turn != T1);
< critical section >
turn = T0;
```

- Alternate which thread can enter critical section
- Is there a problem?
  - A: Yes, this is broken.
  - B: No, this ought to work.

Possible Solution: Take Turns?

```
shared int turn = T0;
```

T_0

```
while (turn != T0);
< critical section >
turn = T1;
```

T_1

```
while (turn != T1);
< critical section >
turn = T0;
```

- Gives us the correctness of Mutual Exclusion (Rule 1)
- Breaks Rule #2: No thread outside its critical section may prevent others from entering their critical sections.
- Gets worse with more threads.

Possible Solution: State Intention?

```
shared boolean flag[2] = {FALSE, FALSE};
```

T_0

```
flag[T0] = TRUE;
while (flag[T1]);
< critical section >
flag[T0] = FALSE;
```

T_1

```
flag[T1] = TRUE;
while (flag[T0]);
< critical section >
flag[T1] = FALSE;
```

- Each thread states it wants to enter critical section
- Is there a problem?
  - A: Yes, this is broken.
  - B: No, this ought to work.

Possible Solution: State Intention?

```
shared boolean flag[2] = {FALSE, FALSE};
```

T_0

```
flag[T0] = TRUE;
while (flag[T1]);
< critical section >
flag[T0] = FALSE;
```

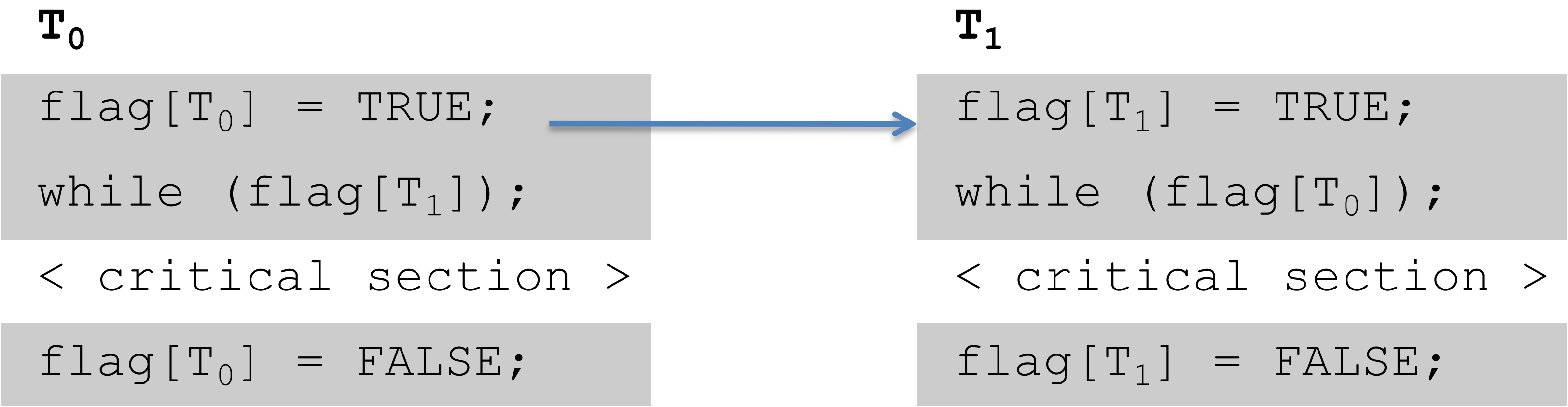
T_1

```
flag[T1] = TRUE;
while (flag[T0]);
< critical section >
flag[T1] = FALSE;
```

- Each thread states it wants to enter critical section
- Is there a problem?
  - A: Yes, this is broken.
  - B: No, this ought to work.

Possible Solution: State Intention?

```
shared boolean flag[2] = {FALSE, FALSE};
```



T_0

```
flag[T0] = TRUE;
while (flag[T1]);
< critical section >
flag[T0] = FALSE;
```

T_1

```
flag[T1] = TRUE;
while (flag[T0]);
< critical section >
flag[T1] = FALSE;
```

- What if threads context switch between these two lines?
- Rule #3: No thread should have to wait forever to enter its critical section (deadlock: neither thread makes progress)

Peterson’s Solution

```
shared int turn;
shared boolean flag[2] = {FALSE, FALSE};
```

T_0

```
flag[T0] = TRUE;
turn = T1;
while (flag[T1] && turn==T1);
< critical section >
flag[T0] = FALSE;
```

T_1

```
flag[T1] = TRUE;
turn = T0;
while (flag[T0] && turn==T0);
< critical section >
flag[T1] = FALSE;
```

- If there is competition, take turns; otherwise, enter
- Is there a problem?
    - A: Yes, this is broken.
    - B: No, this ought to work.

Peterson's Solution

```
shared int turn;
shared boolean flag[2] = {FALSE, FALSE};
```

T_0

```
flag[T0] = TRUE;
turn = T1;
while (flag[T1] && turn==T1);
< critical section >
flag[T0] = FALSE;
```

T_1

```
flag[T1] = TRUE;
turn = T0;
while (flag[T0] && turn==T0);
< critical section >
flag[T1] = FALSE;
```

- If there is competition, take turns; otherwise, enter
- Is there a problem?
  - A: Yes, this is broken.
  - B: No, this ought to work.

Peterson’s Solution

```
shared int turn;
shared boolean flag[2] = {FALSE, FALSE};
```

T_0

```
flag[T0] = TRUE;
turn = T1;
while (flag[T1] && turn==T1);
< critical section >
flag[T0] = FALSE;
```

T_1

```
flag[T1] = TRUE;
turn = T0;
while (flag[T0] && turn==T0);
< critical section >
flag[T1] = FALSE;
```

- Do we like this solution? Are there problems we would like to avoid?
  - A: Yes
  - B: No

Peterson’s Solution

```
shared int turn;
shared boolean flag[2] = {FALSE, FALSE};
```

T_0

```
flag[T0] = TRUE;
turn = T1;
while (flag[T1] && turn==T1);
< critical section >
flag[T0] = FALSE;
```

T_1

```
flag[T1] = TRUE;
turn = T0;
while (flag[T0] && turn==T0);
< critical section >
flag[T1] = FALSE;
```

- Do we like this solution? Are there problems we would like to avoid?
  - Complexity of the solution increases with the number of threads you have.
  - while loop – using CPU doing nothing.

Spinlocks are Wasteful

- If a thread is spinning on a lock, it’s using the CPU without making progress.
  - Single-core system, prevents lock holder from executing.
  - Multi-core system, waste core time when something else could be running.

- Ideal: thread can’t enter critical section? Schedule something else. Consider it *blocked*.

Railroad Semaphore

- track at any given time

Spinlocks are Wasteful

- If a thread is spinning on a lock, it's using the CPU without making progress.
  - Single-core system, prevents lock holder from executing.
  - Multi-core system, waste core time when something else could be running.

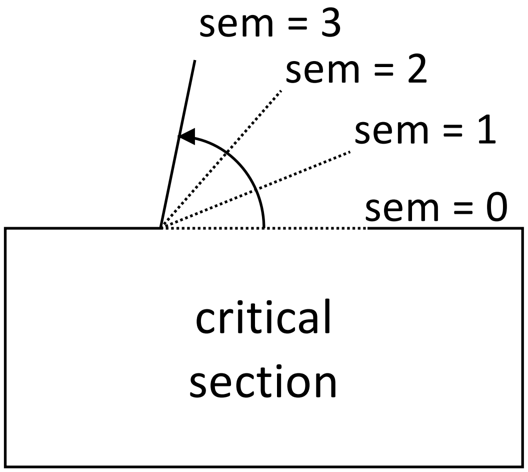
- Ideal: thread can't enter critical section? Schedule something else. Consider it *blocked*.

Atomicity

- How do we get away from having to know about all other interested threads?
- 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

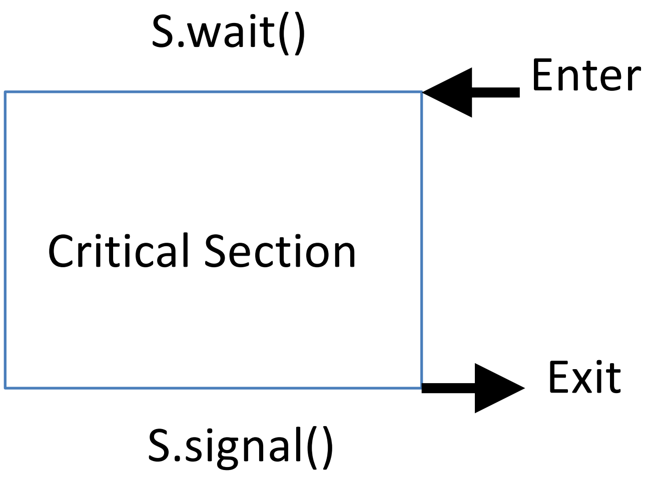
Semaphores

- 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



Semaphores

- Associated with each semaphore (S) is a queue of waiting threads
- When wait() is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- Then signal() opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread



S.wait()

Enter

Critical Section

Exit

S.signal()

Semaphore Operations

```
sem s = n;    //initialize: num. copies of resource
```

Executes atomically

```
wait (sem s)
   decrement s;
   if s < 0, block thread (and associate with s);
```

Executes atomically

```
signal (sem s)
   increment s;
   if blocked threads, unblock (any) one of them;
```

Semaphore: an integer variable that can be updated only using two special atomic instructions (test/set, compare/swap)

Semaphore Operations

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

Executes atomically

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

Executes atomically

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

Based on what you know about semaphores, should a process be able to check beforehand whether wait(s) will cause it to block?

A. Yes, it should be able to check.

B. No, it should not be able to check.

Semaphore Operations

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

Executes atomically

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

Executes atomically

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

Based on what you know about semaphores, should a process be able to check beforehand whether wait(s) will cause it to block?

A. Yes, it should be able to check.

B. No, it should not be able to check.

Semaphore Operations

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

Executes atomically

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

Executes atomically

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

- No other operations allowed
- In particular, semaphore’s value can’t be tested!
  - No thread can tell the value of semaphore s

Mutual Exclusion with Semaphores

```
sem mutex = 1;
```

T_0

```
wait (mutex);
< critical section >
signal (mutex);
```

T_1

```
wait (mutex);
< critical section >
signal (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
- Is there any busy-waiting?

Locking Abstraction

- One way to implement critical sections is to “lock the door” on the way in, and unlock it again on the way out
  - Typically exports “nicer” interface for semaphores in user space

- A lock is an object in memory providing two operations
  - acquire()/lock(): before entering the critical section
  - release()/unlock(): after leaving a critical section

- Threads pair calls to acquire() and release()
  - Between acquire()/release(), the thread holds the lock
  - acquire() does not return until any previous holder releases
  - What can happen if the calls are not paired?

Using Locks

Thread A

```
main ()
{ int a,b;


  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);


  return a;
}
```

shared memory

```
s = 40;
```

Thread B

```
main ()
{ int a,b;


  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);


  return a;
}
```

Using Locks

Thread A

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

```
s = 40;
Lock l;
```

Lock Held by:
Nobody

Using Locks

Thread A

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared
memory

```
s = 40;
Lock l;
```

Lock held by:
Thread A

Using Locks

Thread A

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

```
s = 40;
Lock l;
```

Lock held by:
Thread A

Using Locks

Thread A

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

```
s = 40;
Lock l;
```

Lock held by:
Thread A

Using Locks

Thread A

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

```
s = 40;
Lock l;
```

Lock Held by:
Nobody

Using Locks

Thread A

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

```
s = 40;
Lock l;
```

Lock held by:
Thread B

Using Locks

Thread A

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

```
s = 40;
Lock l;
```

Lock Held by:
Nobody

Thread B

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

Using Locks

Thread A

```
main  ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

```
s = 40;
Lock l;
```

Thread B

```
main  ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

Lock Held by:
Nobody

- No matter how we order threads or when we context switch, result will always be 30, like we expected (and probably wanted).

“Deadly Embrace”

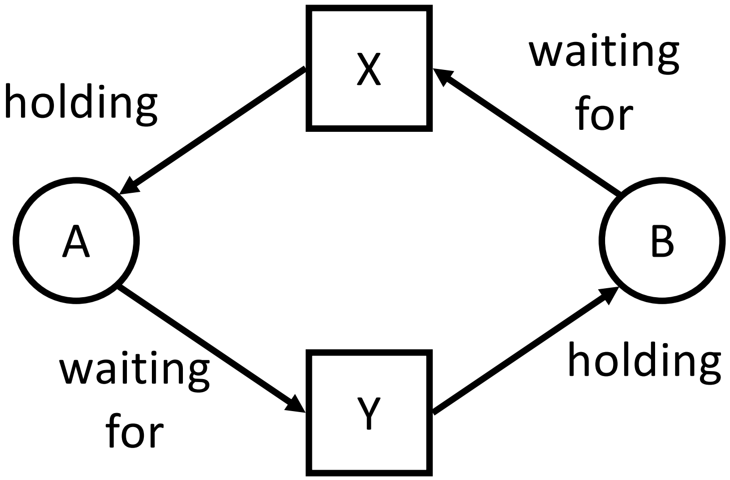
- *The Structure of the THE-Multiprogramming System* (Edsger Dijkstra, 1968)
- Also introduced semaphores
- Deadlock is as old as synchronization

What is Deadlock?

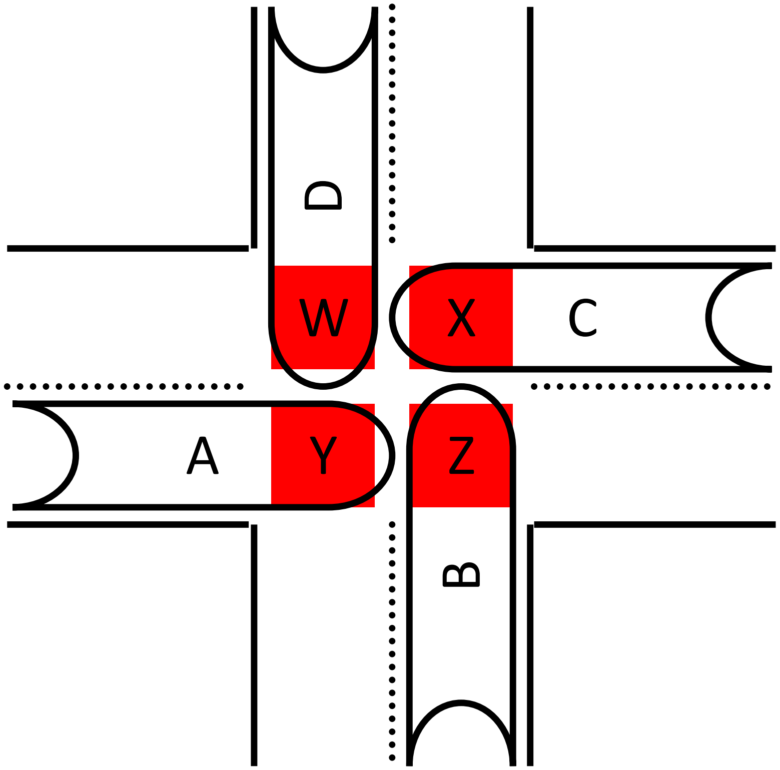
- Deadlock is a problem that can arise:
  - When processes compete for access to limited resources
  - When threads are incorrectly synchronized
- Definition:
  - Deadlock exists among a set of threads if every thread is waiting for an event that can be caused only by another thread in the set.

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



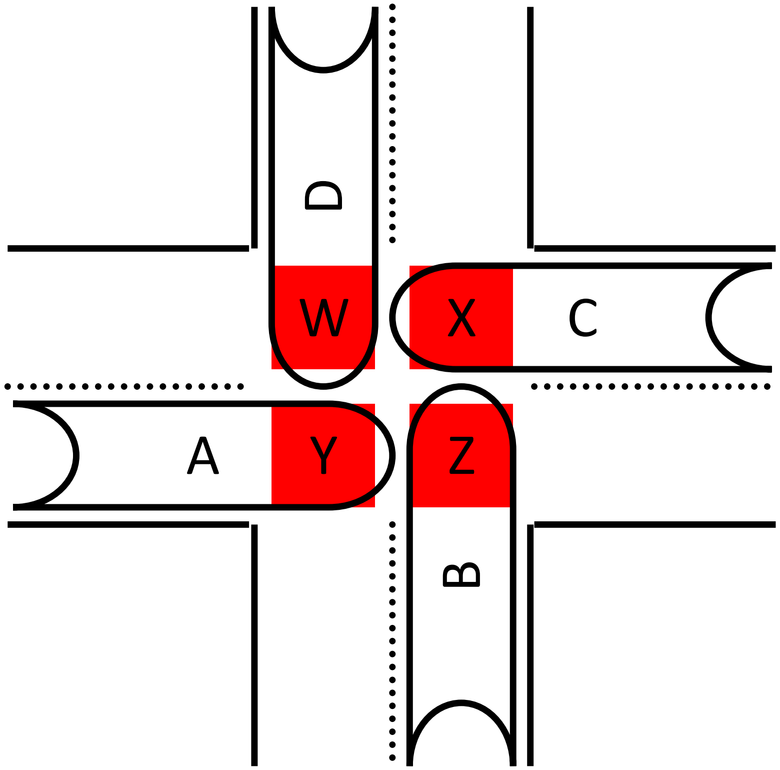
Traffic Jam as Example of Deadlock



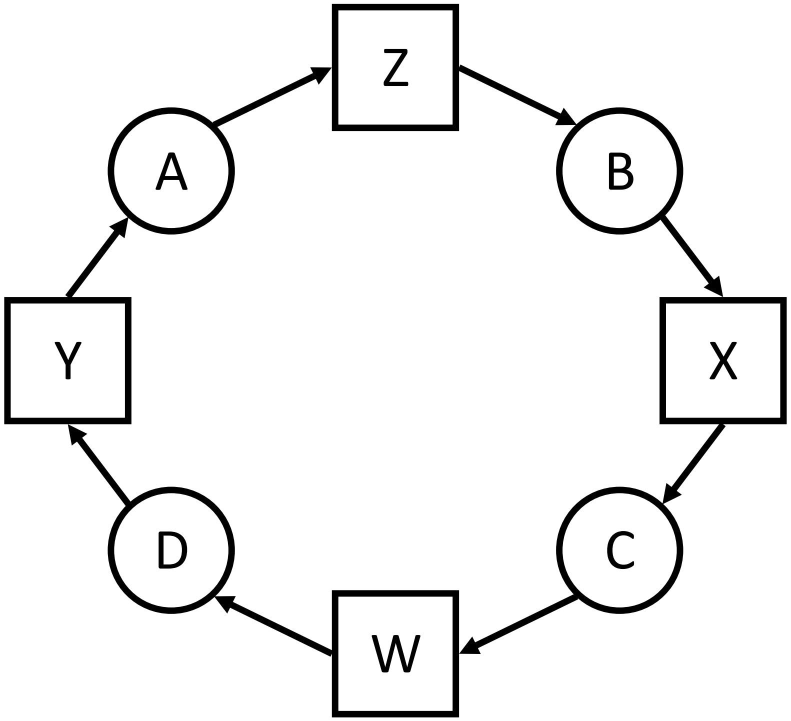
Cars deadlocked
in an intersection

- 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

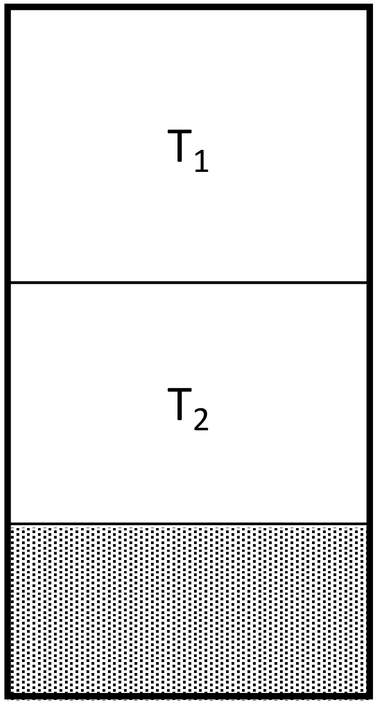
1. Mutual Exclusion
   - Only one thread may use a resource at a time.
2. Hold-and-Wait
   - Thread holds resource while waiting for another.
3. No Preemption
   - Can’t take a resource away from a thread.
4. Circular Wait
   - The waiting threads form a cycle.

Four Conditions for Deadlock

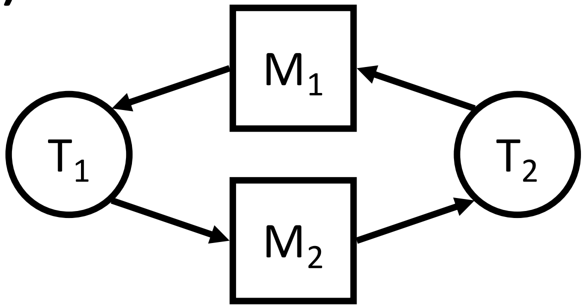
1. Mutual Exclusion
   - Only one thread may use a resource at a time.
2. Hold-and-Wait
   - Thread holds resource while waiting for another.
3. No Preemption
   - Can’t take a resource away from a thread.
4. Circular Wait
   - The waiting threads form a cycle.

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



Four Conditions for Deadlock

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

How to Attack the Deadlock Problem

- What should your OS do to help you?

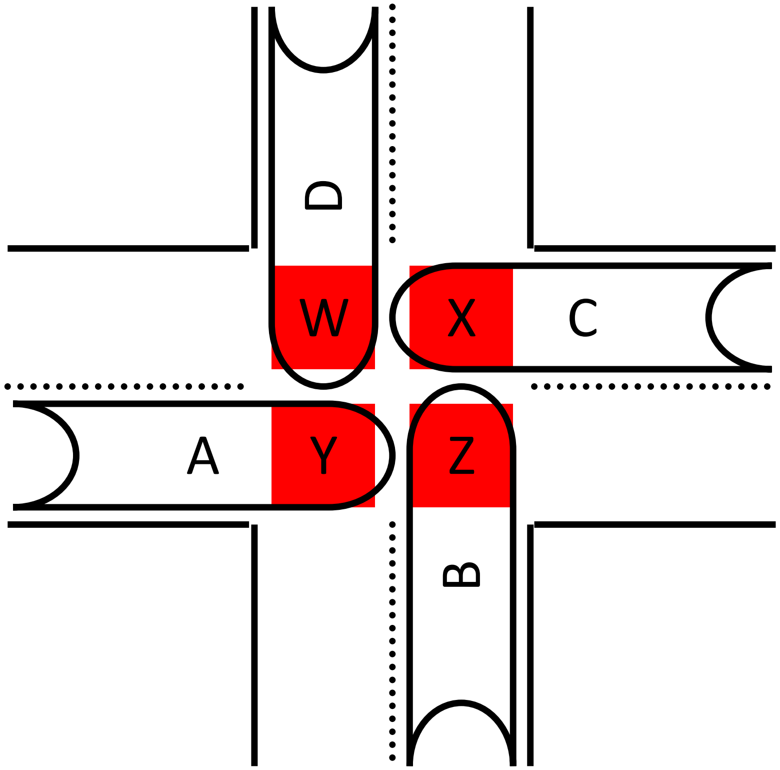
- Deadlock Prevention
  - Make deadlock impossible by removing a condition
- Deadlock Avoidance
  - Avoid getting into situations that lead to deadlock
- Deadlock Detection
  - Don’t try to stop deadlocks
  - Rather, if they happen, detect and resolve

How to Attack the Deadlock Problem

- What should your OS do to help you?

- 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

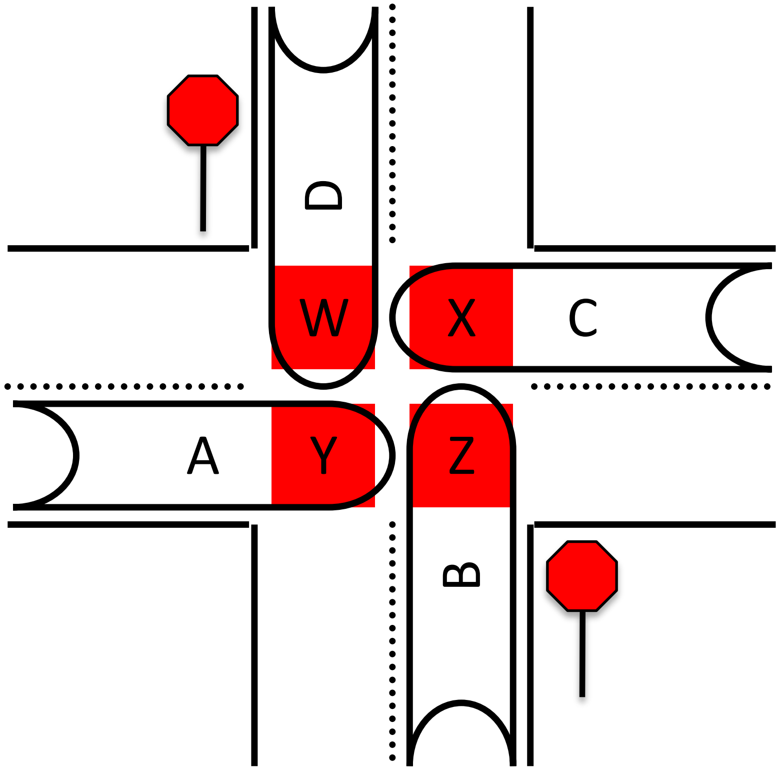
How Can We Prevent a Traffic Jam?



Cars deadlocked in an intersection

- Do intersections usually look like this one?
- We have road infrastructure (mechanisms)
- We have road rules (policies)

Suppose we add <u>north/south stop signs.</u> Which condition would that eliminate?



A. Mutual exclusion

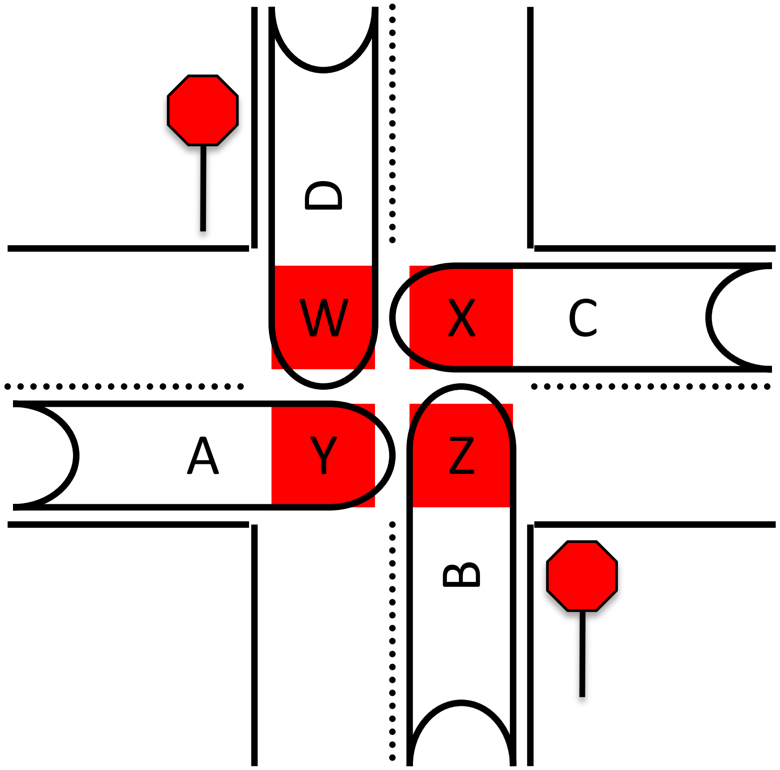
B. Hold and wait

C. No preemption

D. Circular wait

E. More than one (which?)

Suppose we add north/south stop signs. Which condition would that eliminate?



A. Mutual exclusion

B. Hold and wait

C. No preemption

D. Circular wait

E. More than one (which?)

Deadlock Prevention

- Simply prevent any single condition for deadlock

1. Mutual exclusion
   - Make all resources sharable
2. Hold-and-wait
   - Get all resources simultaneously (wait until all free)
   - Only request resources when it has none

Deadlock Prevention

- Simply prevent any single condition for deadlock

3. No preemption
   - Allow resources to be taken away (at any time)
4. Circular wait
   - Order all the resources, force ordered acquisition

Which of these conditions is easiest to give up to prevent deadlocks?

A. Mutual exclusion (make everything sharable)

B. Hold and wait (must get all resources at once)

C. No preemption (resources can be taken away)

D. Circular wait (total order on resource requests)

E. I’m not willing to give up any of these!

Which of these conditions is easiest to give up to prevent deadlocks?

A. Mutual exclusion (make everything sharable)

B. Hold and wait (must get all resources at once)

C. No preemption (resources can be taken away)

D. Circular wait (total order on resource requests)

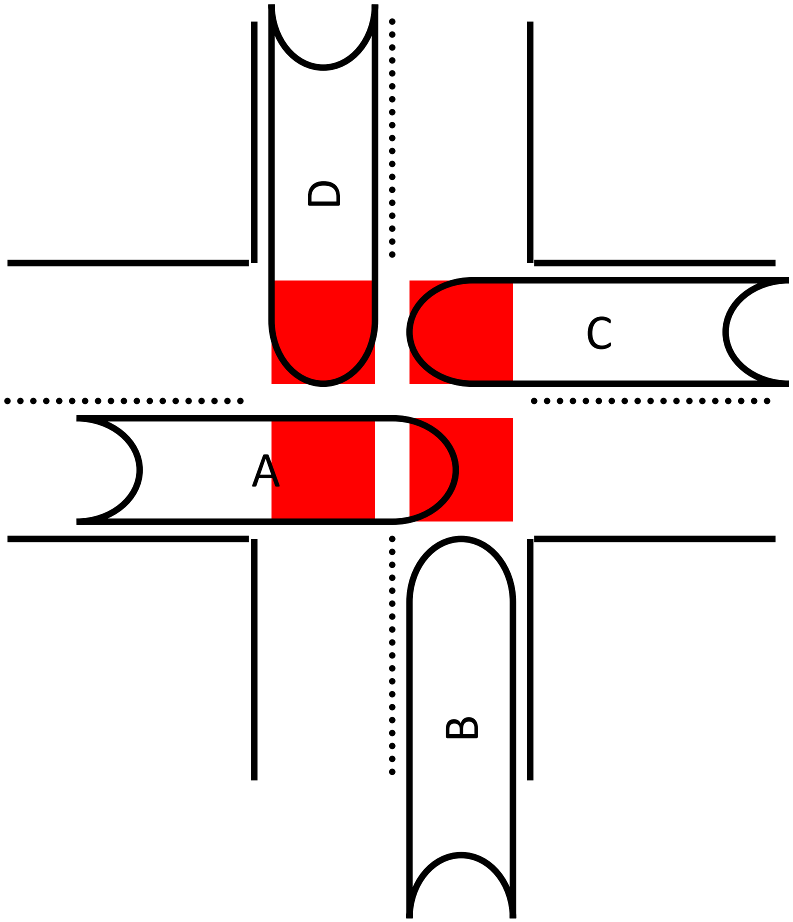
E. I’m not willing to give up any of these!

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

How Can We Avoid a Traffic Jam?

- What are the incremental resources?
- Safe* state:
  - No possibility of deadlock
  - <= 3 cars in intersection
- Unsafe state:
  - Deadlock possible, don’t allow



*Don’t try this while driving…

Deadlock Avoidance

- Eliminates deadlock

- Must know max resource usage in advance
  - Do we always know resources at compile time?
  - Do we specify resources at run time? Could we?

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

Deadlock Detection and Recovery

- Do nothing special to prevent/avoid deadlocks
  - If they happen, they happen
  - Periodically, try to detect if a deadlock occurred
  - Do something to resolve it
- Reasoning
  - Deadlocks rarely happen (hopefully)
  - Cost of prevention or avoidance not worth it
  - Deal with them in special way (may be very costly)

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

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

- <u>These all have major drawbacks…</u>

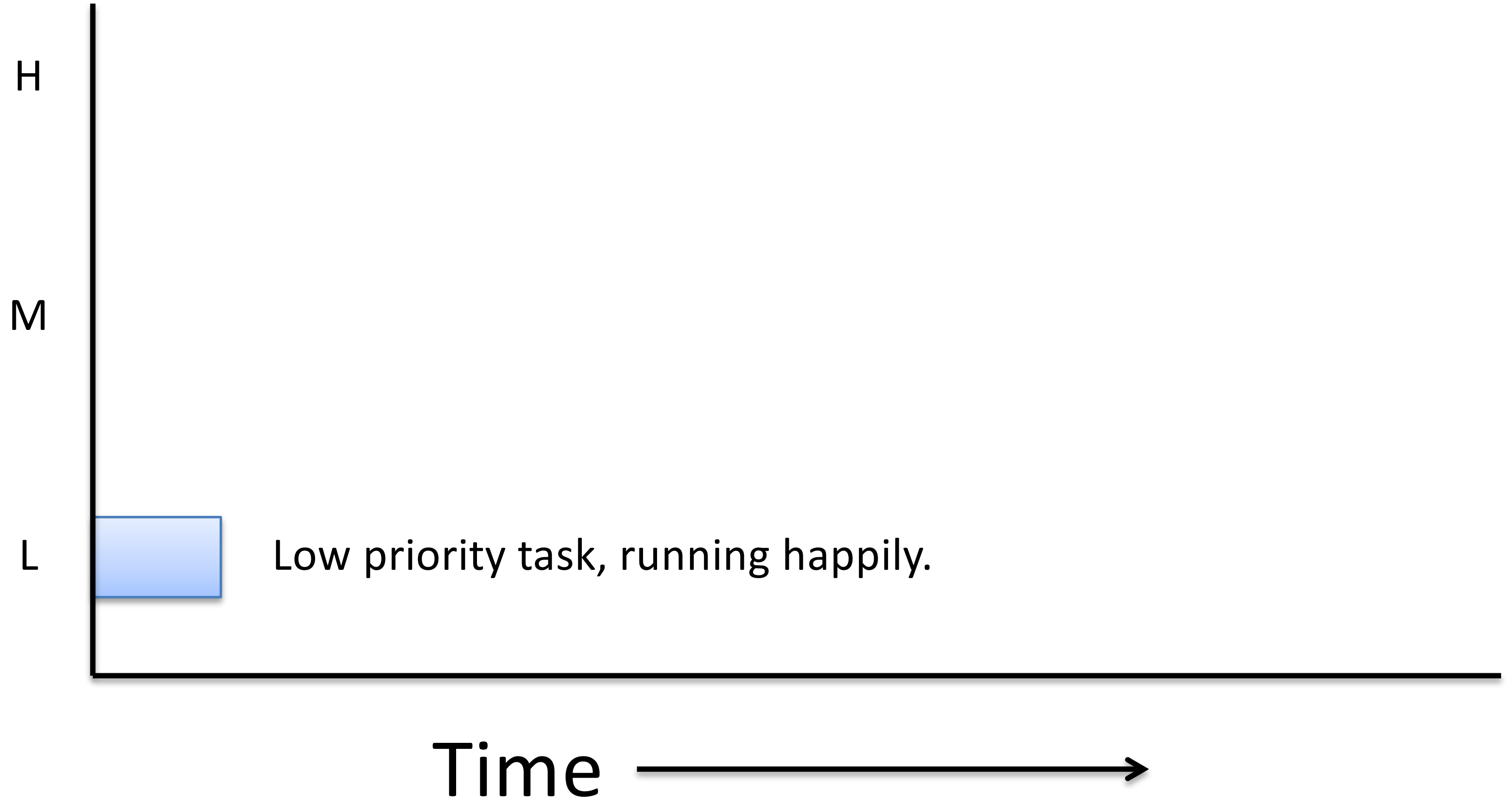
Other Thread Complications

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

Priority Inversion

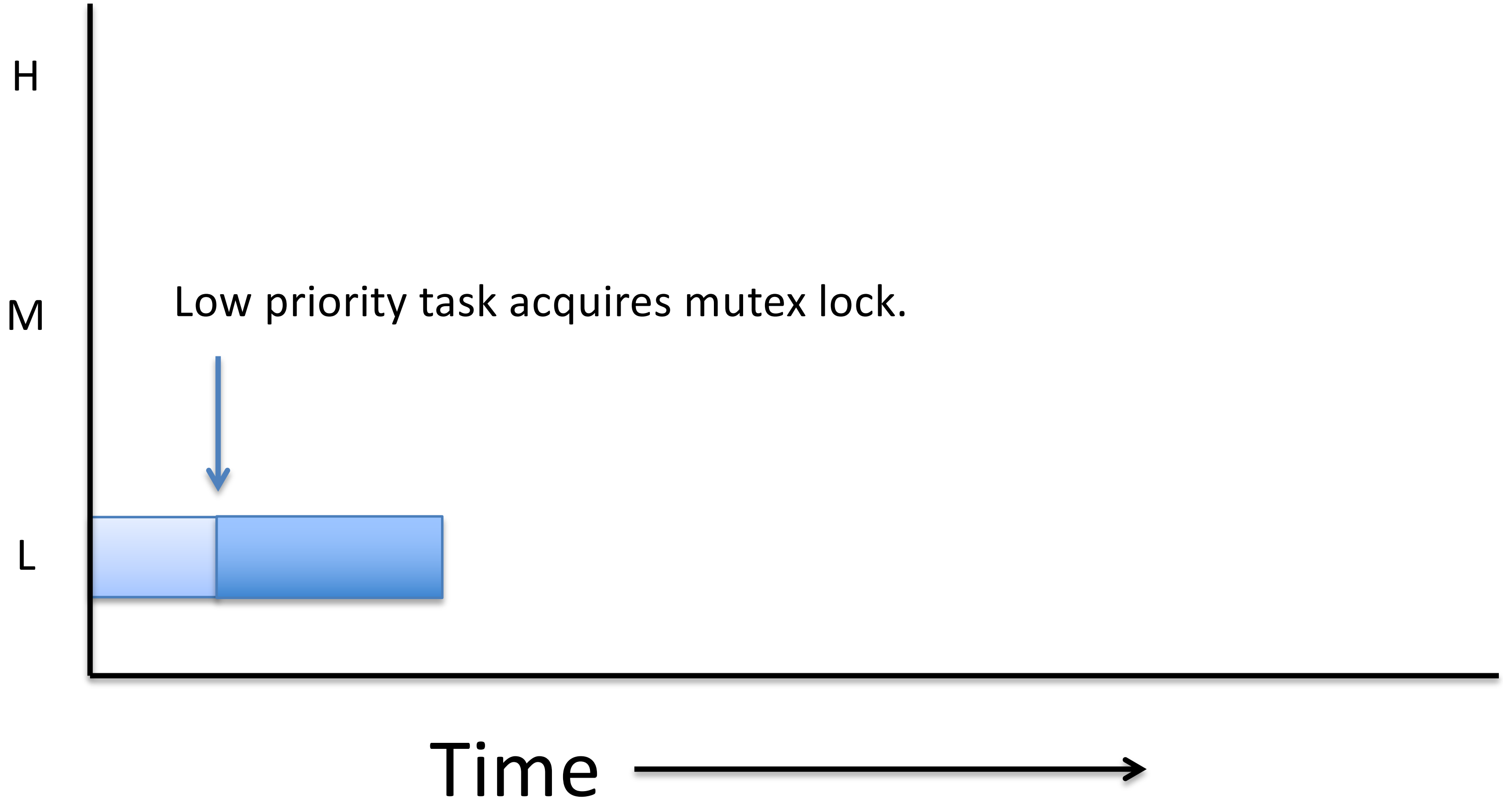
- Problem: Low priority thread holds lock, high priority thread waiting for lock.
  - What needs to happen: boost low priority thread so that it can finish, release the lock
  - What sometimes happens in practice: low priority thread not scheduled, can’t release lock
- Example: Mars Pathfinder (1997)

What Happened: Priority Inversion



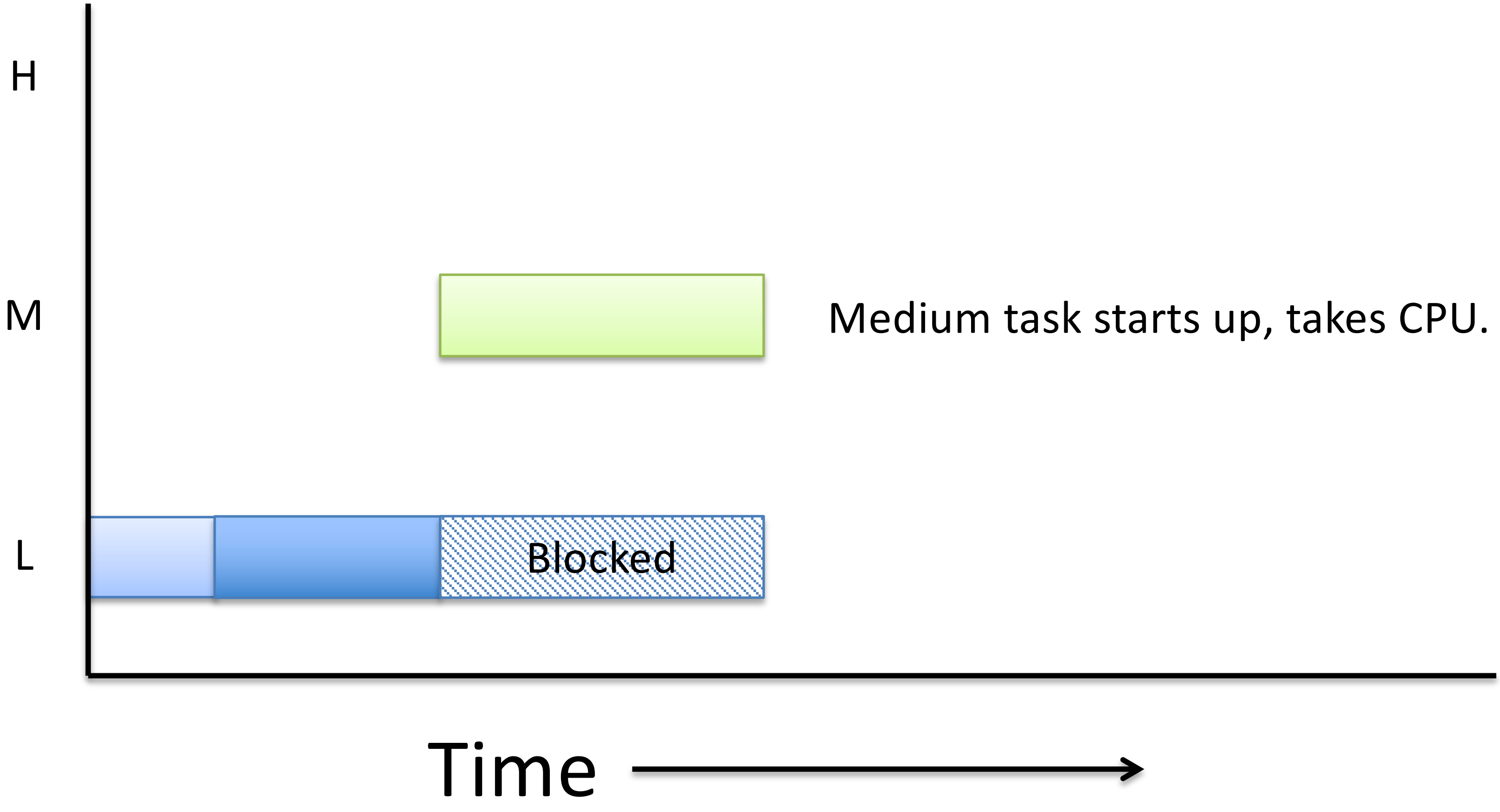
Low priority task, running happily.

What Happened: Priority Inversion



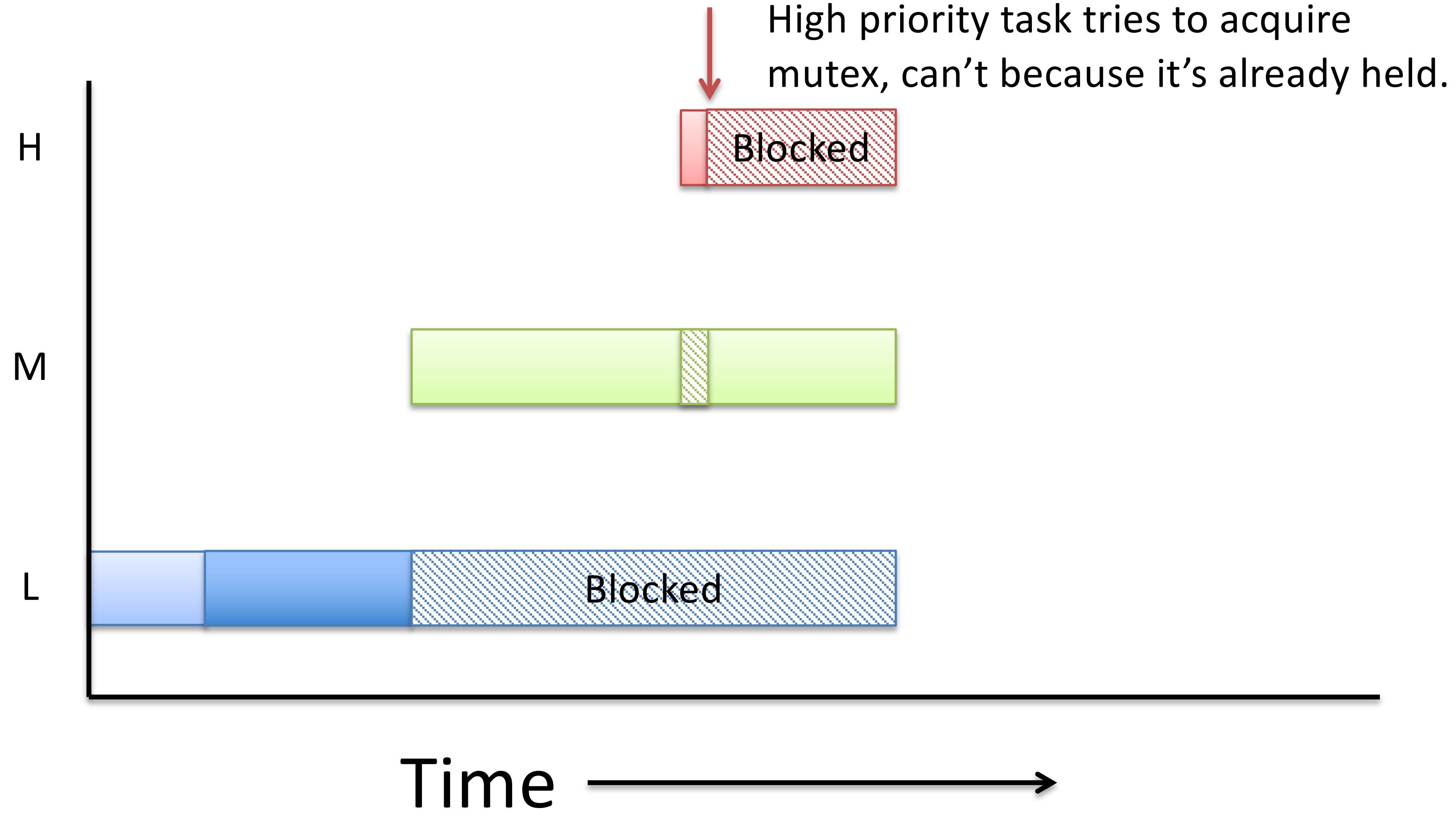
Low priority task acquires mutex lock.

What Happened: Priority Inversion

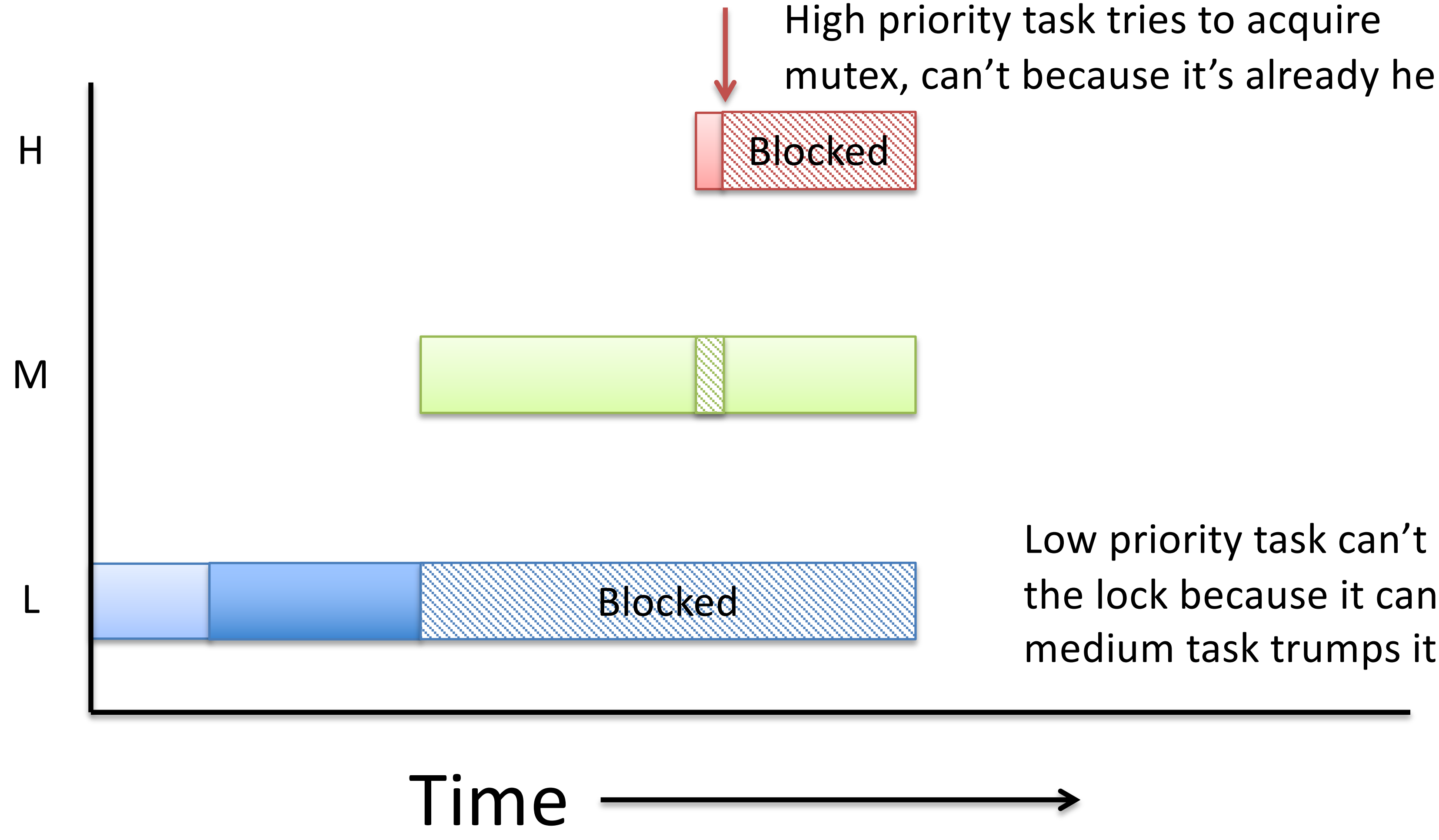


Medium task starts up, takes CPU.

What Happened: Priority Inversion


High priority task tries to acquire mutex, can't because it's already held.

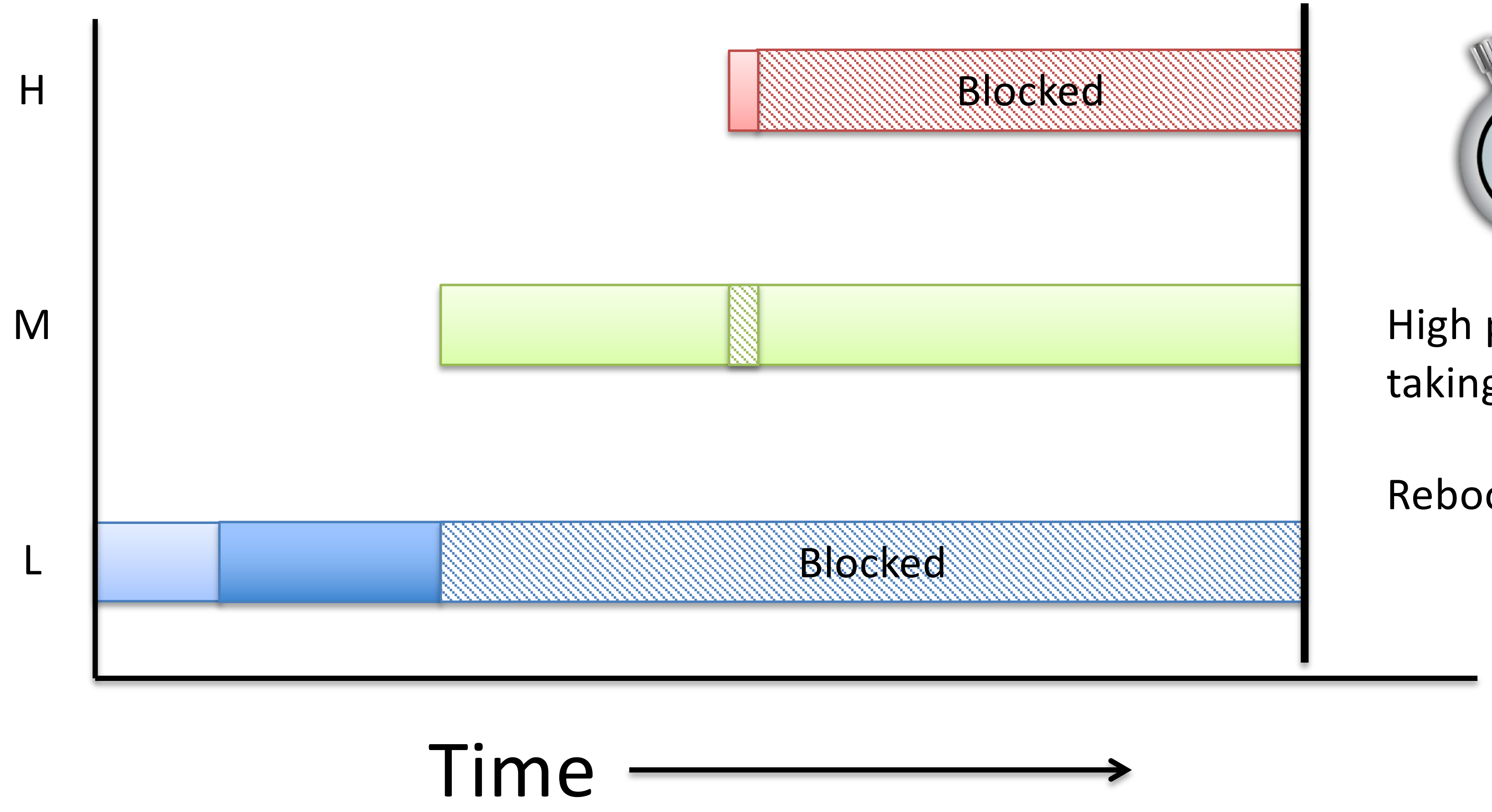
What Happened: Priority Inversion



High priority task tries to acquire mutex, can't because it's already held.

Low priority task can't give up the lock because it can't run - medium task trumps it.

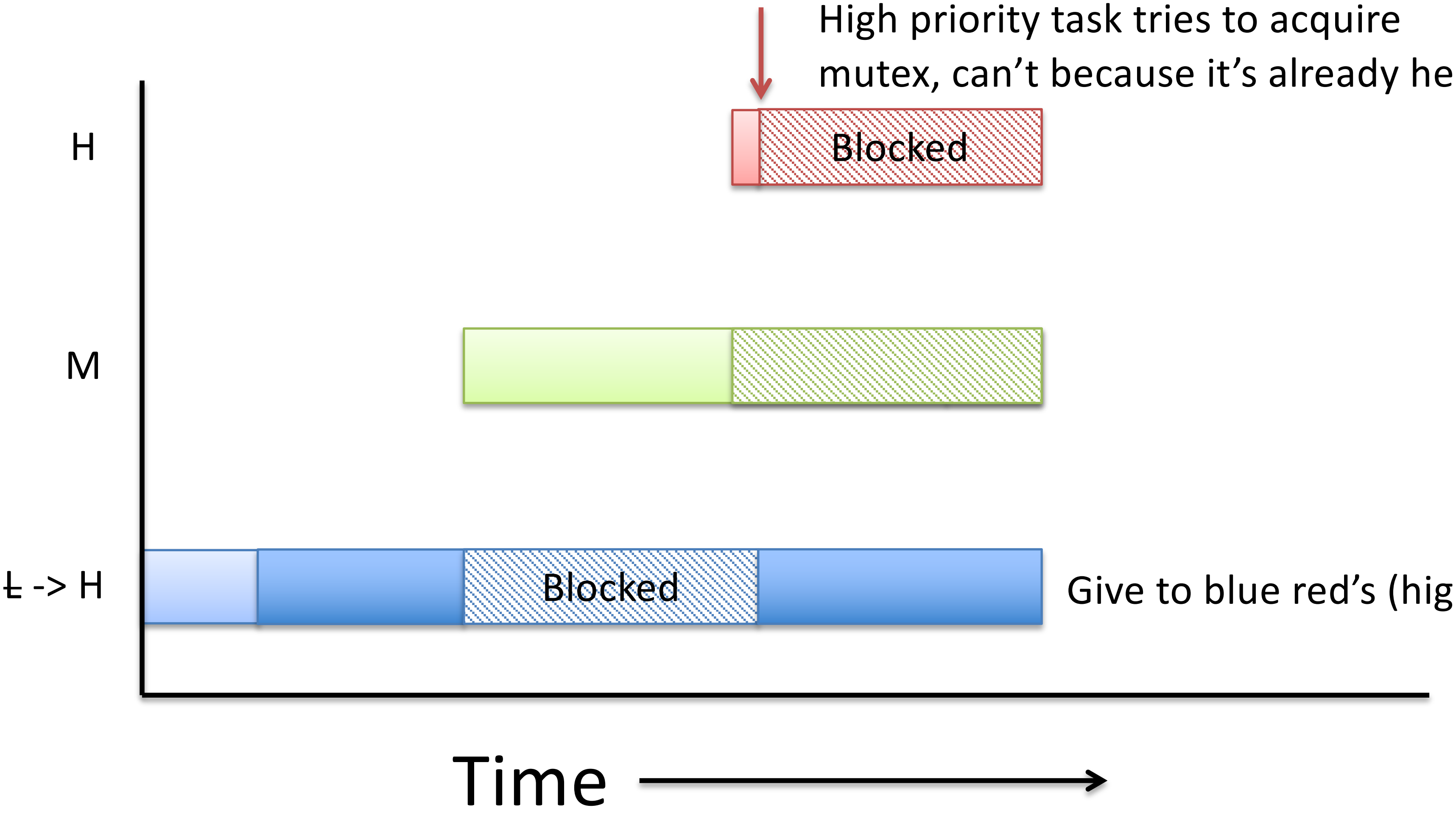
What Happened: Priority Inversion



High priority is taking too long.

Reboot!

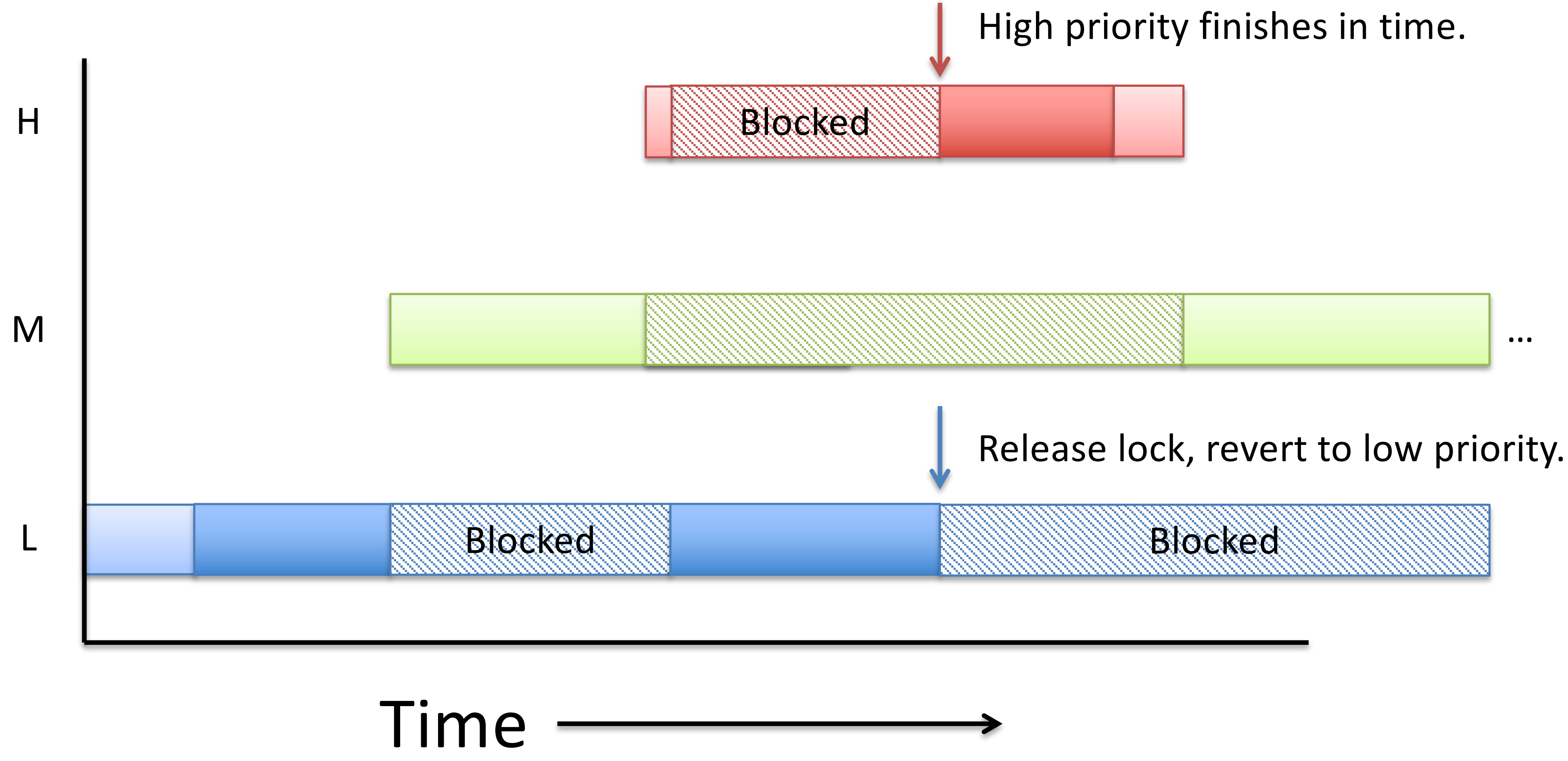
Solution: Priority Inheritance

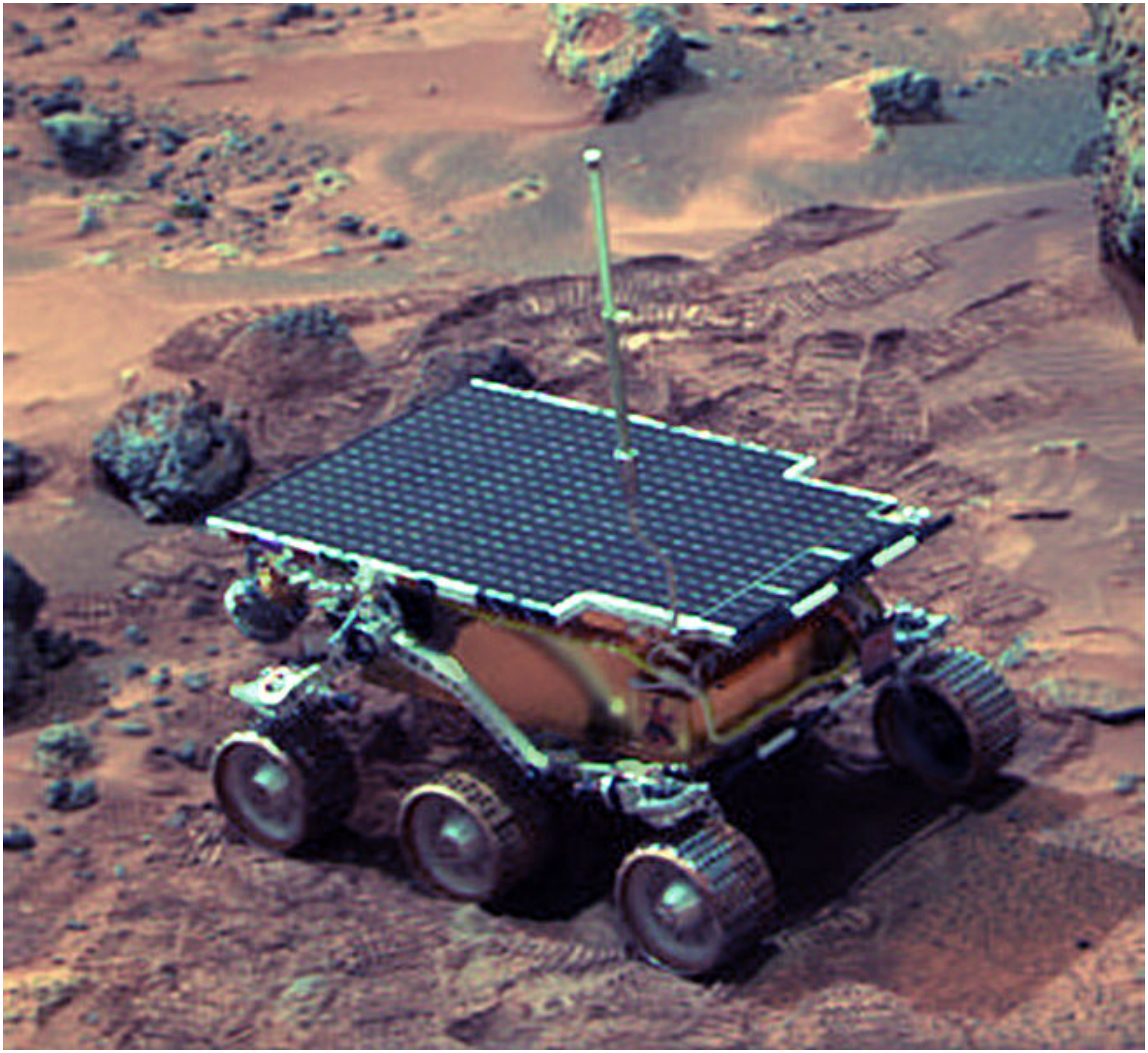


High priority task tries to acquire mutex, can’t because it’s already held.

Give to blue red’s (higher) priority!

Solution: Priority Inheritance





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

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".

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

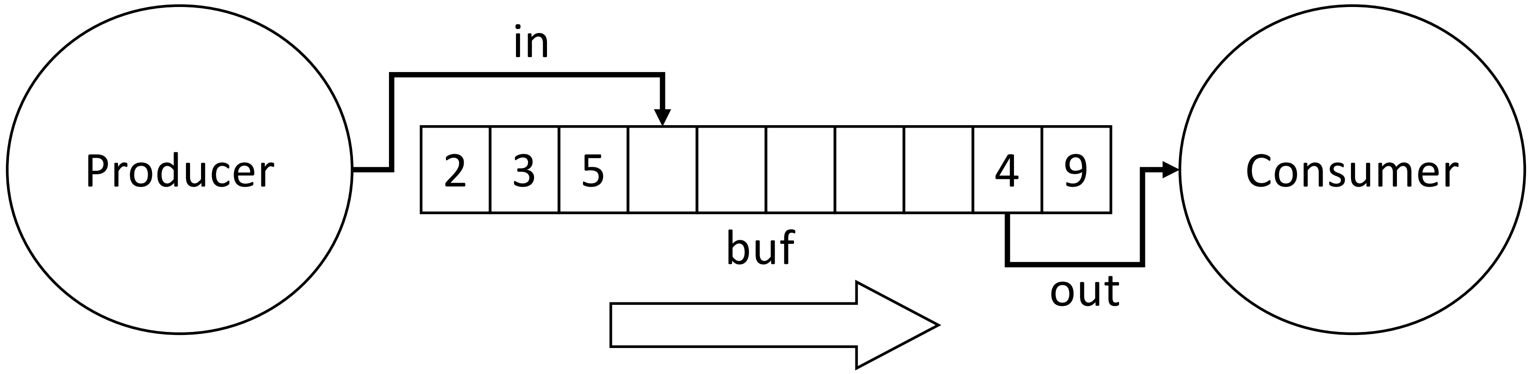
Agenda

- Classic thread patterns
- Pthreads primitives and examples of other forms of synchronization:
  - Condition variables
  - Barriers
  - RW locks
  - Message passing
- Message passing: alternative to shared memory

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	**Consumer**

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

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 (producer overwrites existing data in the buffer, or consumer tries to consume from an empty buffer).
  B. No, this ought to be fine.

Adding Semaphores

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

Producer                              Consumer
while (TRUE) {                        while (TRUE) {
    wait (X);                             wait (Z);
    buf[in] = Produce ();                 Consume (buf[out]);
    in = (in + 1)%N;                      out = (out + 1)%N;
    signal (Y);                           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?

Wait = decrements, blocks at zero
Signal = increment and unblocking semaphore

```
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

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

Wait = decrements, blocks at zero

Signal = increment and unblocking semaphore

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

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

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
  - pthread_cond_destroy: free resources

- 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

```
while (TRUE) {
   //independent code

   lock(m); //lock mutex
   while (conditions bad)
      wait(cond, m); //pass in mutex, because
      //atomically waiting on cond. var. and
      // unlocking mutex someone else can make
      // cond. good.

   //upon waking up, proceed knowing that
conditions are now good

   signal (other_cond);  // Let other thread
know that you finished your work.
   unlock(m);
}
```

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.

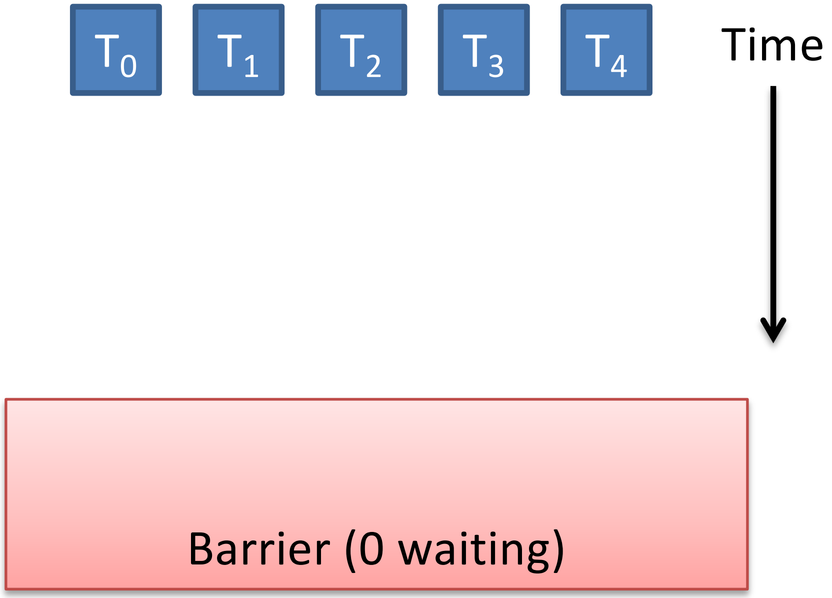
Barriers

- Used to coordinate threads, but also other forms of concurrent execution.

- Often found in simulations that have discrete rounds. (e.g., game of life)

Barrier Example, N Threads

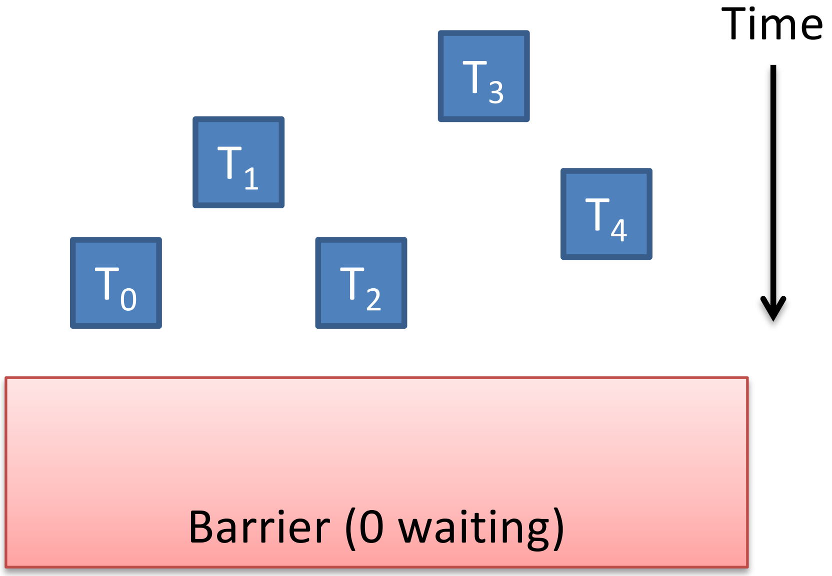
```
shared barrier b;

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)
  }
}
```

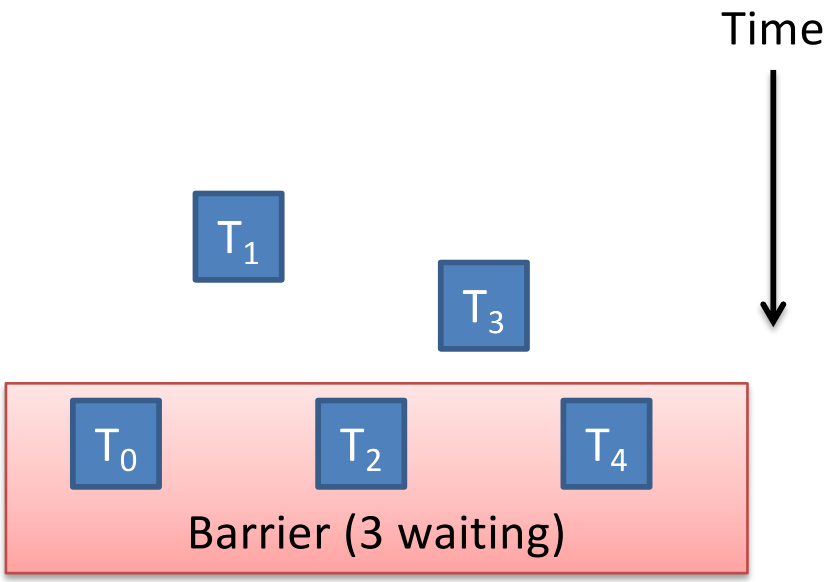
Threads make progress computing current round at different rates.



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)
  }
}
```

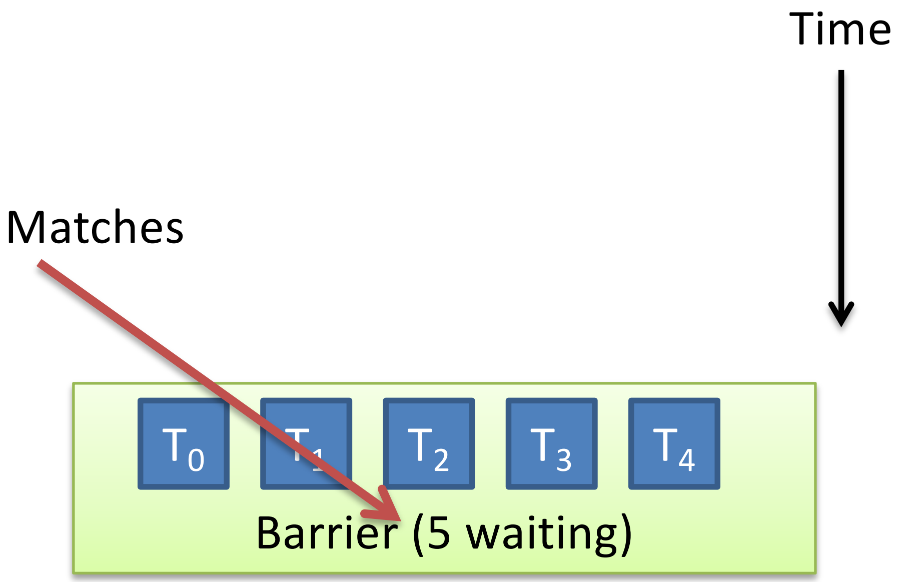
Threads that make it to barrier must wait for all others to get there.



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

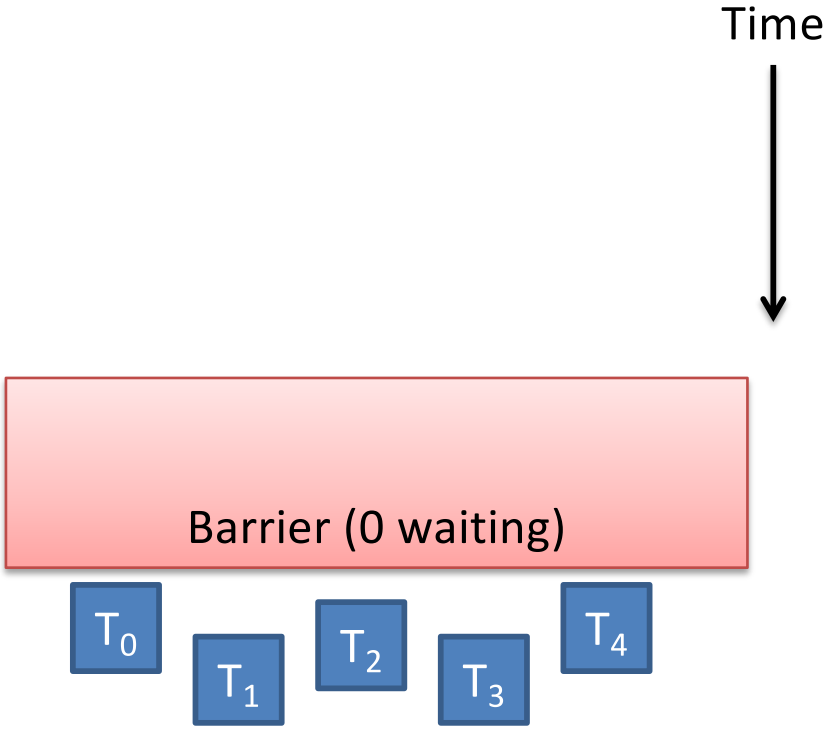
T_0 T_1 T_2 T_3 T_4

Barrier (5 waiting)

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…



Synchronization: More than Mutexes

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

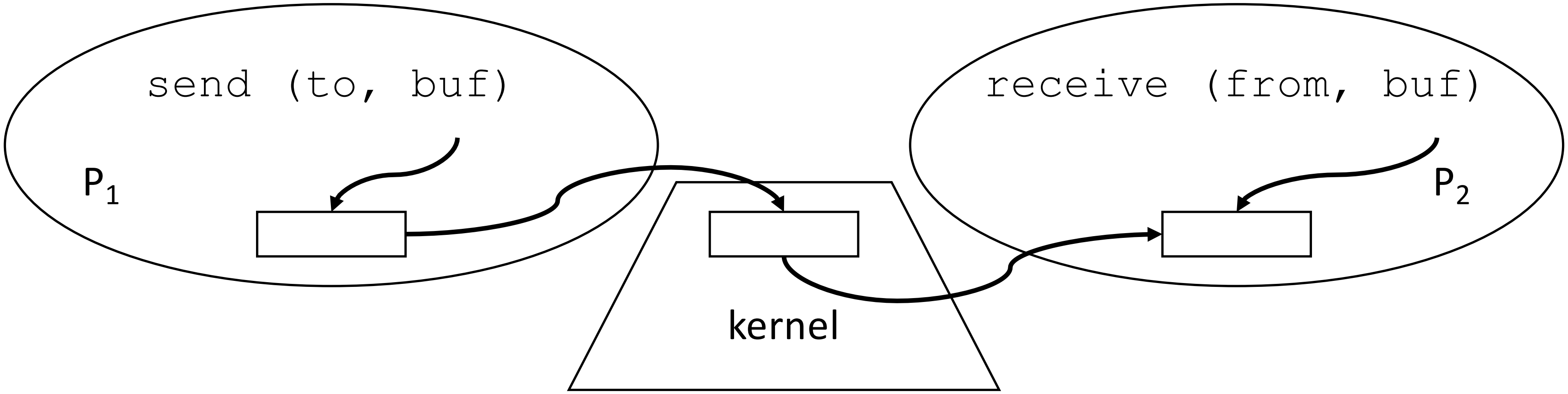
- “I want all my threads to sync up at the same point.”
  - Barrier: wait for everyone to catch up.

- “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

Readers/Writers

- Readers/Writers Problem:
  - An object is shared among several threads
  - Some threads only read the object, others only write it
  - We can safely allow multiple readers
  - But only one writer
- pthread_rwlock_t:
  - pthread_rwlock_init: initialize rwlock
  - pthread_rwlock_rdlock: lock for reading
  - pthread_rwlock_wrlock: lock for writing

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

Suppose we're using message passing, will this code operate correctly?

```
/* NO SHARED MEMORY */

Producer                             Consumer
int item;                            int item;

while (TRUE) {                       while (TRUE) {
    item = Produce ();                   receive (Producer, &item);
    send (Consumer, &item);              Consume (item);
}                                    }
```

A. No, there is a race condition.

B. No, we need to protect *item*.

C. Yes, this code is correct.

Suppose we're using message passing, will this code operate correctly?

```
/* NO SHARED MEMORY */

Producer                    Consumer
int item;                   int item;

while (TRUE) {              while (TRUE) {
    item = Produce ();          receive (Producer, &item);
    send (Consumer, &item);     Consume (item);
}                           }
```

A. No, there is a race condition.

B. No, we need to protect *item*.

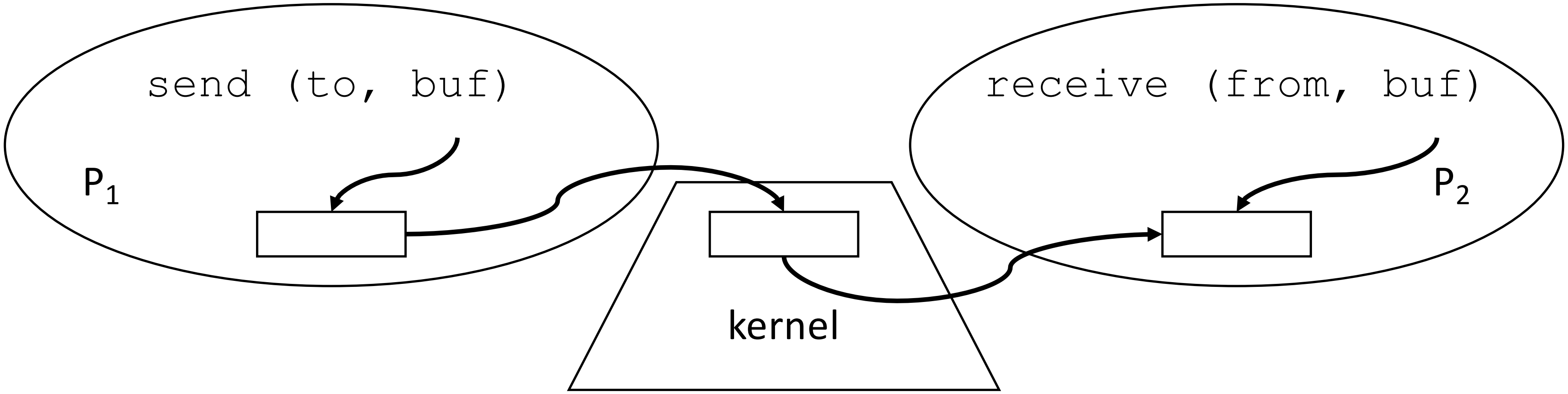
C. Yes, this code is correct.

This code is correct and relatively simple. Why don’t we always just use message passing (vs semaphores, etc.)?

```
/* NO SHARED MEMORY */

Producer                   Consumer
int item;                  int item;

while (TRUE) {             while (TRUE) {
   item = Produce ();         receive (Producer, &item);
   send (Consumer, &item);    Consume (item);
}                          }
```

A. Message passing copies more data.
B. Message passing only works across a network.
C. Message passing is a security risk.
D. We usually do use message passing!

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

Issues with Message Passing

- Who should messages be addressed to?
  - ports (mailboxes) rather than processes/threads
- What if it wants to receive from anyone?
  - `pid = receive (*, msg)`
- Synchronous (blocking) vs. asynchronous (non-blocking)
- Kernel buffering: how many sends w/o receives?
- Good paradigm for IPC over networks

Summary

- Many ways to solve the same classic problems
  - Producer/Consumer: semaphores, CVs, messages

- There's more to synchronization than just mutual exclusion!
  - Condition variables, barriers, RWlocks, and others.

- Message passing doesn't require shared mem.
  - Useful for "threads" on different machines.