Deadlock

12/1/16

Two topics today

- Deadlock:
  - What it is.
  - How it can happen.
  - How to deal with it.

- Assembly support for atomicity:
  - Test-and-set
  - Compare-and-swap

What is Deadlock?

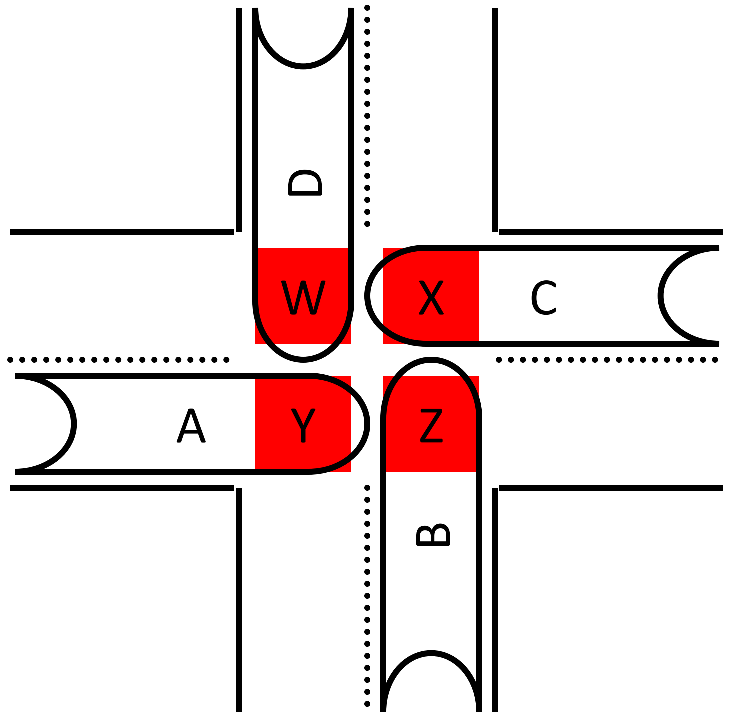
- Deadlock is a problem that can arise…
  - when processes compete for access to limited system resources.
  - when threads are incorrectly synchronized.

not necessarily our fault

our fault (as the programmer)

- 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.

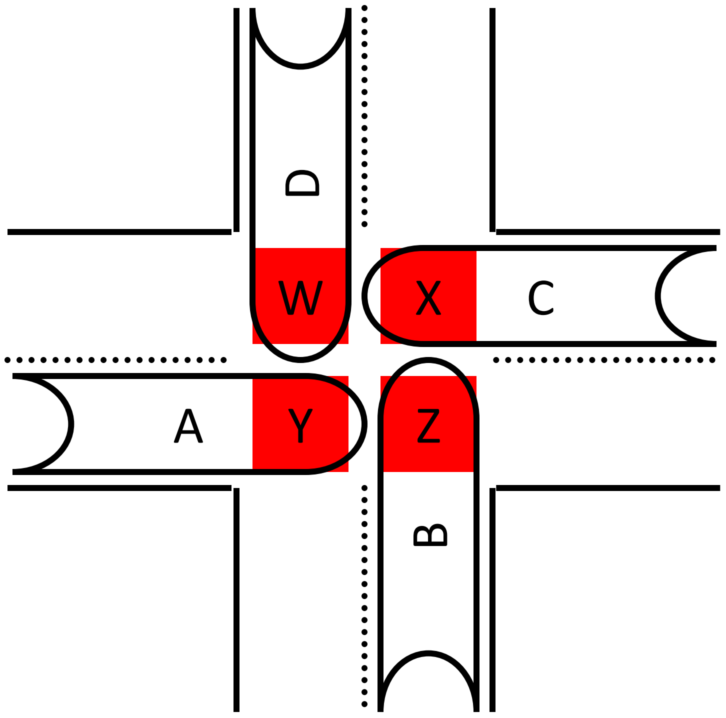
Traffic Jam as Example of Deadlock



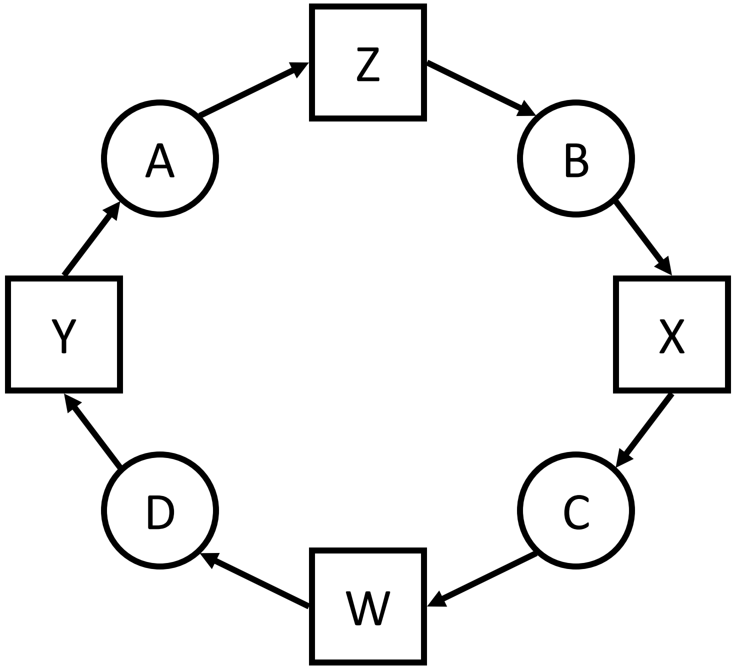
Cars deadlocked
in an intersection

- 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.

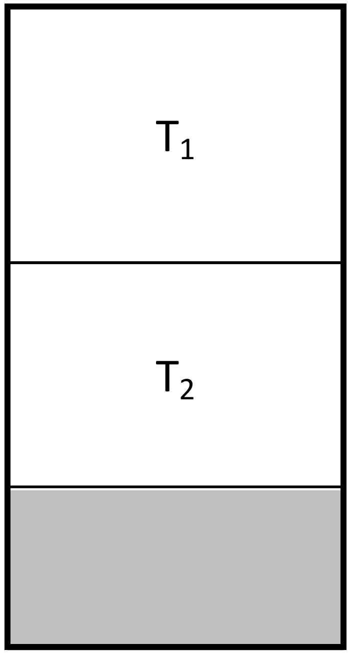
Why are all four necessary?

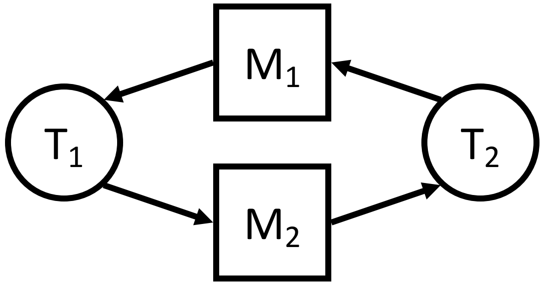
For each condition, assume it doesn’t occur, but the other 3 do, and explain why deadlock can’t happen.

1. Mutual Exclusion
2. Hold-and-Wait
3. No Preemption
4. Circular Wait

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);
```

T_0 holds A’s lock, will make no progress until it can get B’s.
T_1 holds B’s lock, will make no progress until it can get A’s.

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.

Deadlock Prevention

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
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!

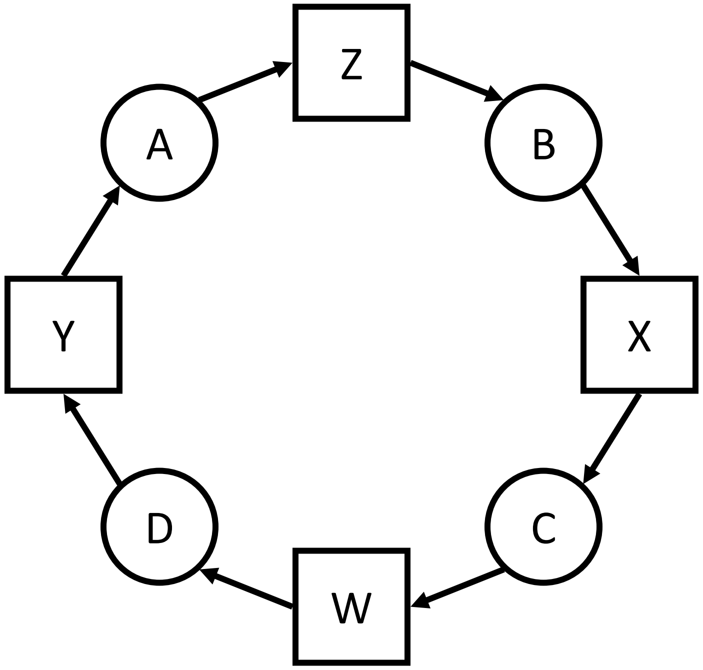
Deadlock Avoidance

- Only allow resource acquisition if there is no way it could lead to deadlock.
- This is necessarily conservative, so there will be more waiting.
- We must know max resource usage in advance.
  - How could we know this and track it?
  - Depends on the resources involved.

Detecting a Deadlock

- Construct resource graph

- Requires
  - Identifying all resources
  - Tracking their use
  - Periodically running detection algorithm



Recovery from Deadlock

1. Abort all deadlocked threads / processes
    - Will remove deadlock, but drastic and costly

2. Abort deadlocked threads one-at-at-time
    - Do until deadlock goes away (need to detect)
    - What order should threads be aborted?

Recovery from Deadlock

3. Preempt resources (force their release)
   - Need to select thread and resource to preempt
   - Need to rollback thread to previous state
   - Need to prevent starvation

4. What about resources in inconsistent states
   - Such as files that are partially written?
   - Or interrupted message (e.g., file) transfers?

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

A. Deadlock prevention

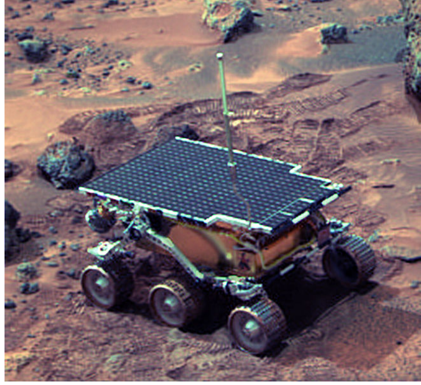
B. Deadlock avoidance

C. Deadlock detection/recovery

D. Something else

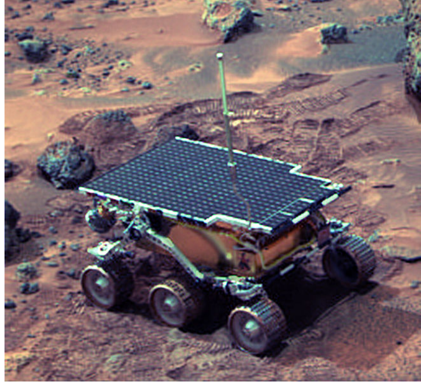
"Ostrich Algorithm"

A mars rover deadlock



- 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.

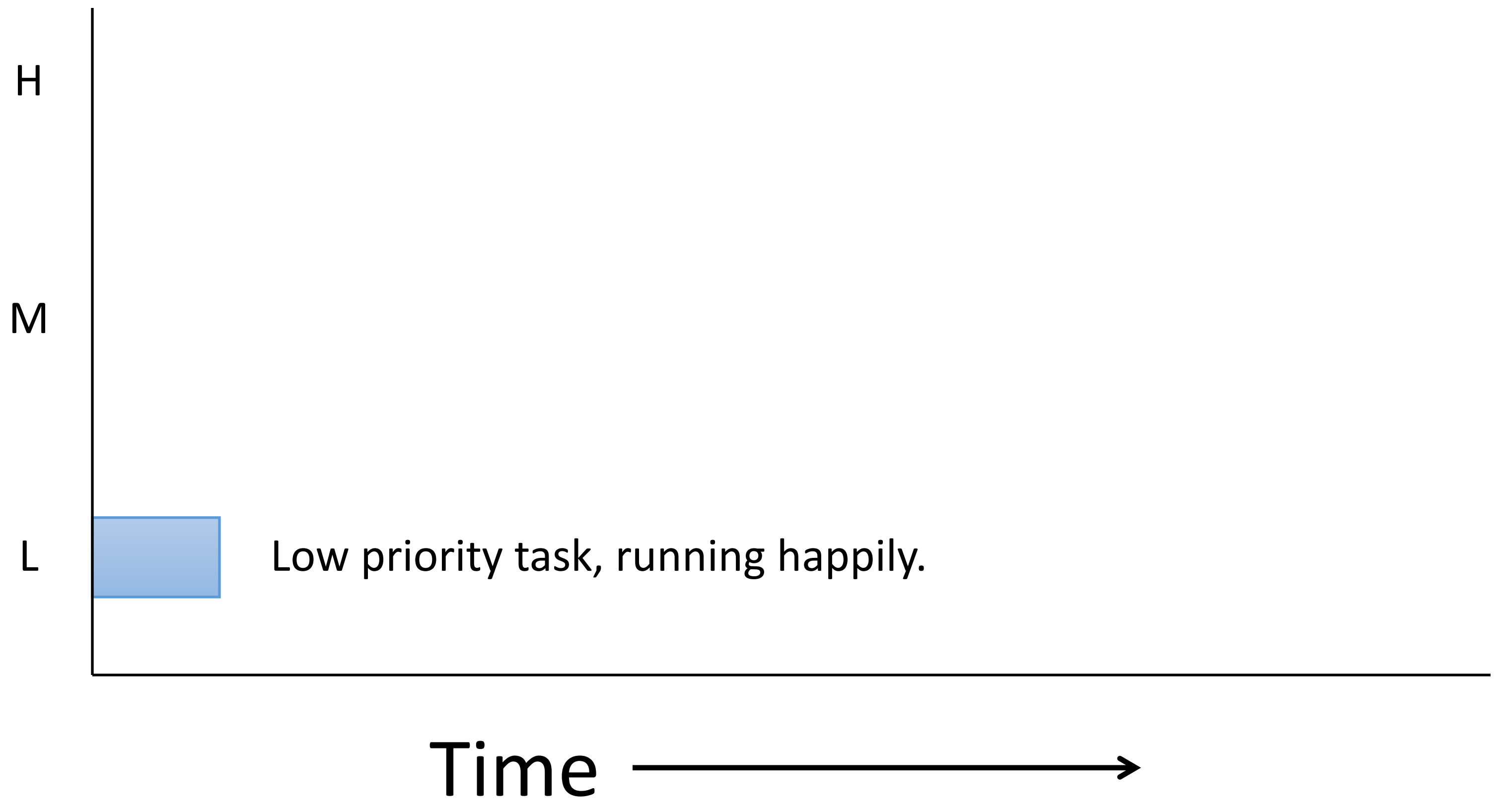


Mars Rover

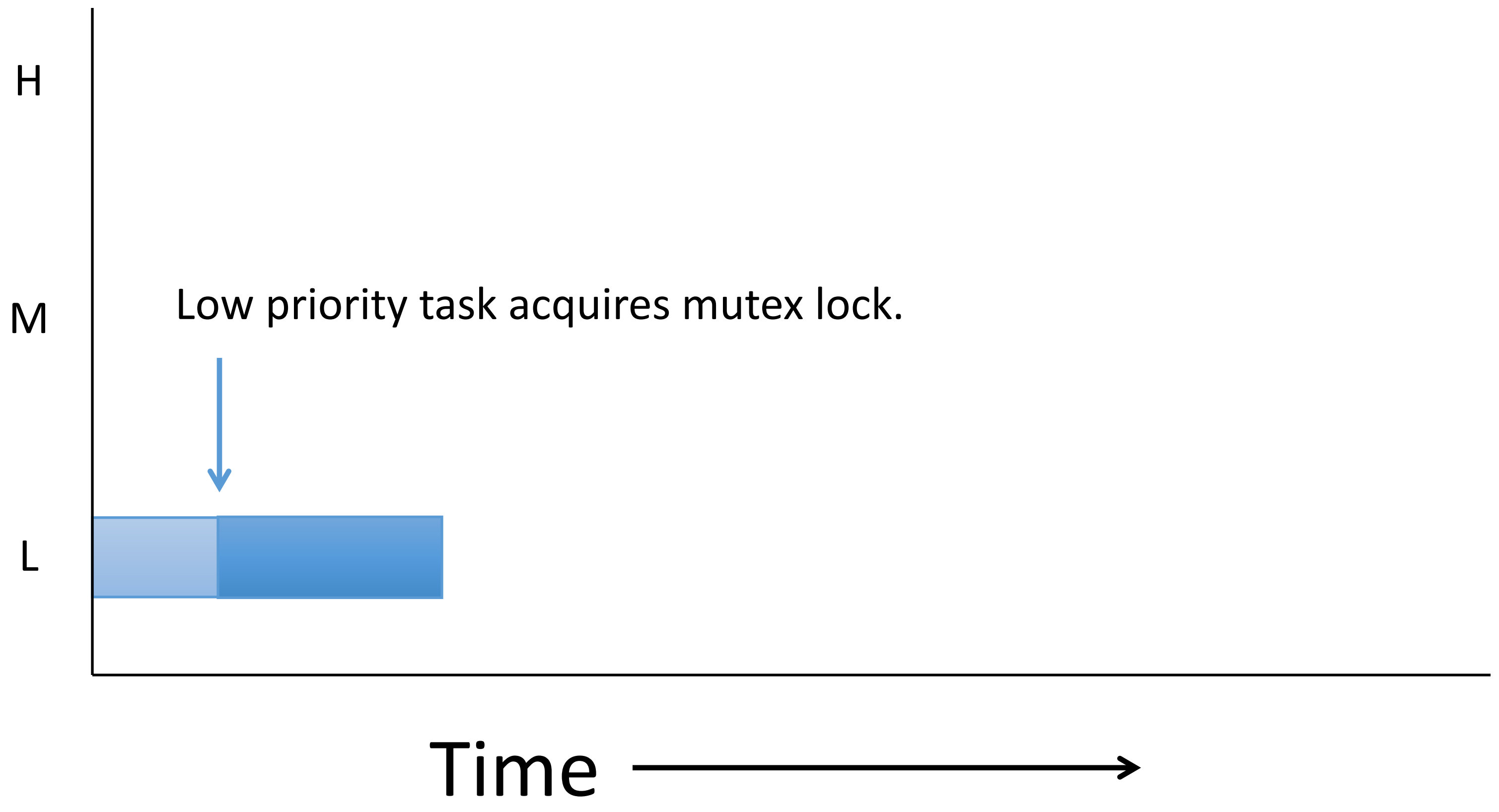
- Failsafe timer (watchdog): if high priority task doesn’t complete in time, reboot system
- Observation: uh-oh, this thing seems to be rebooting a lot, we’re losing data…

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".

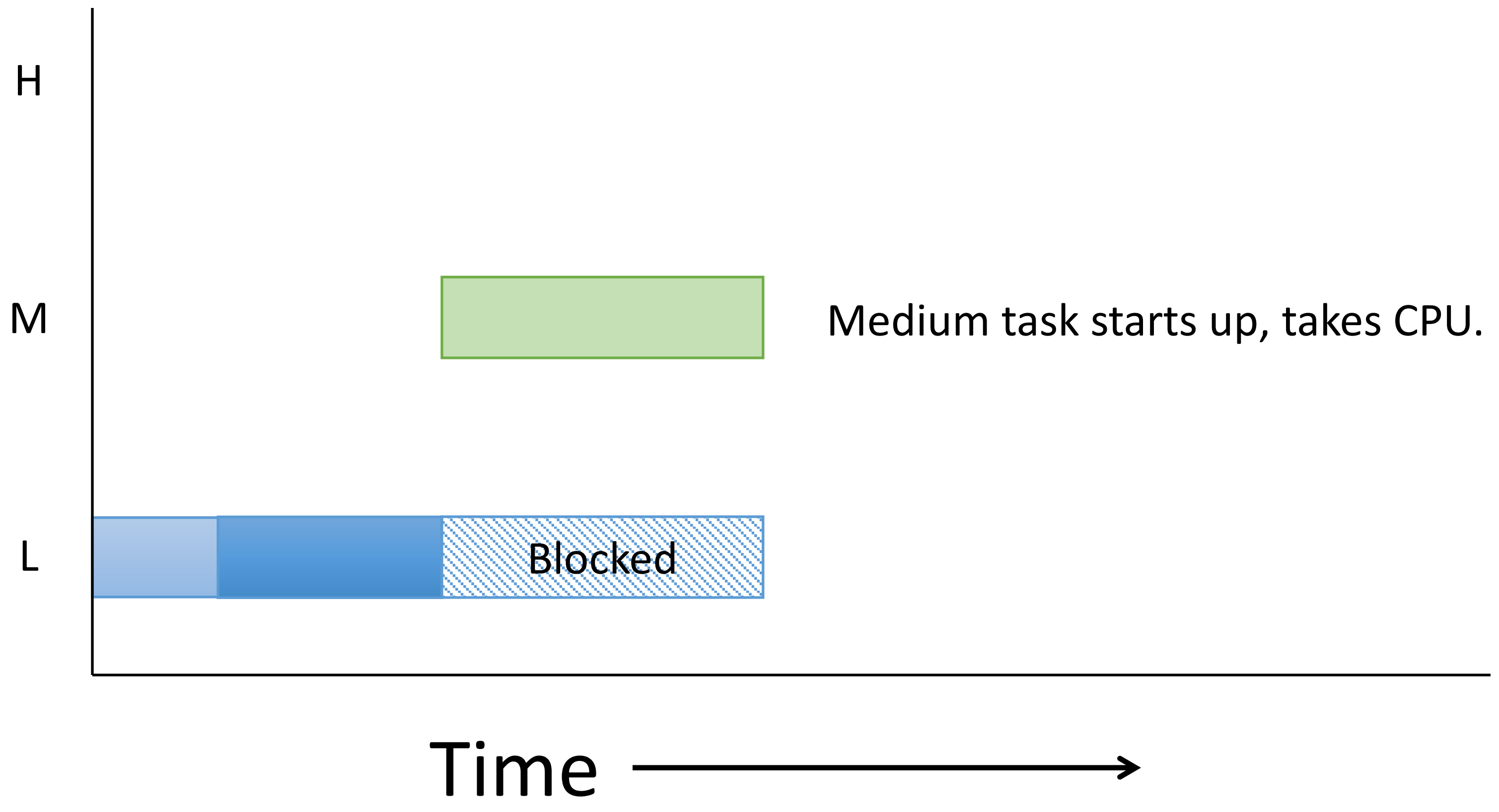
What Happened: Priority Inversion



What Happened: Priority Inversion



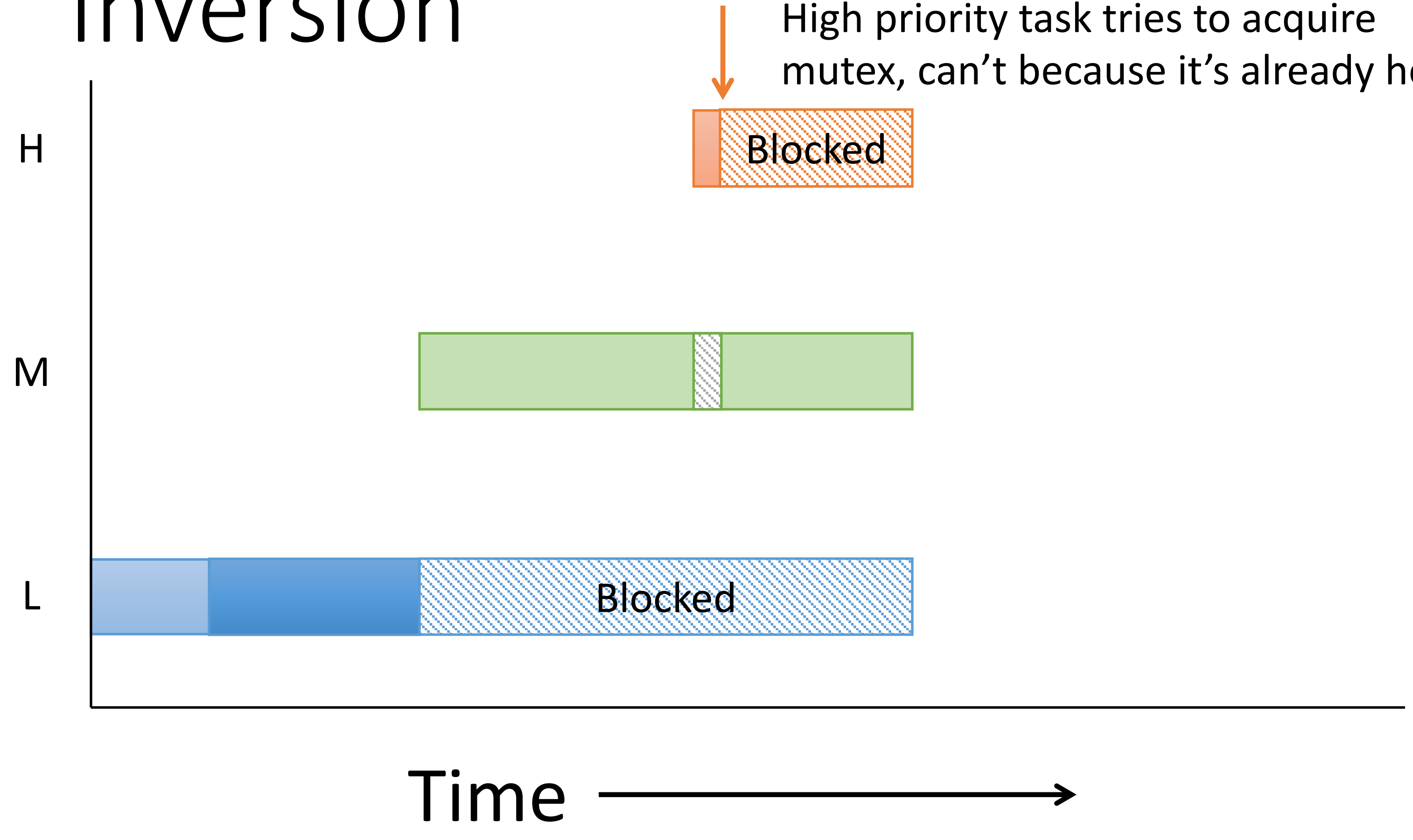
What Happened: Priority Inversion



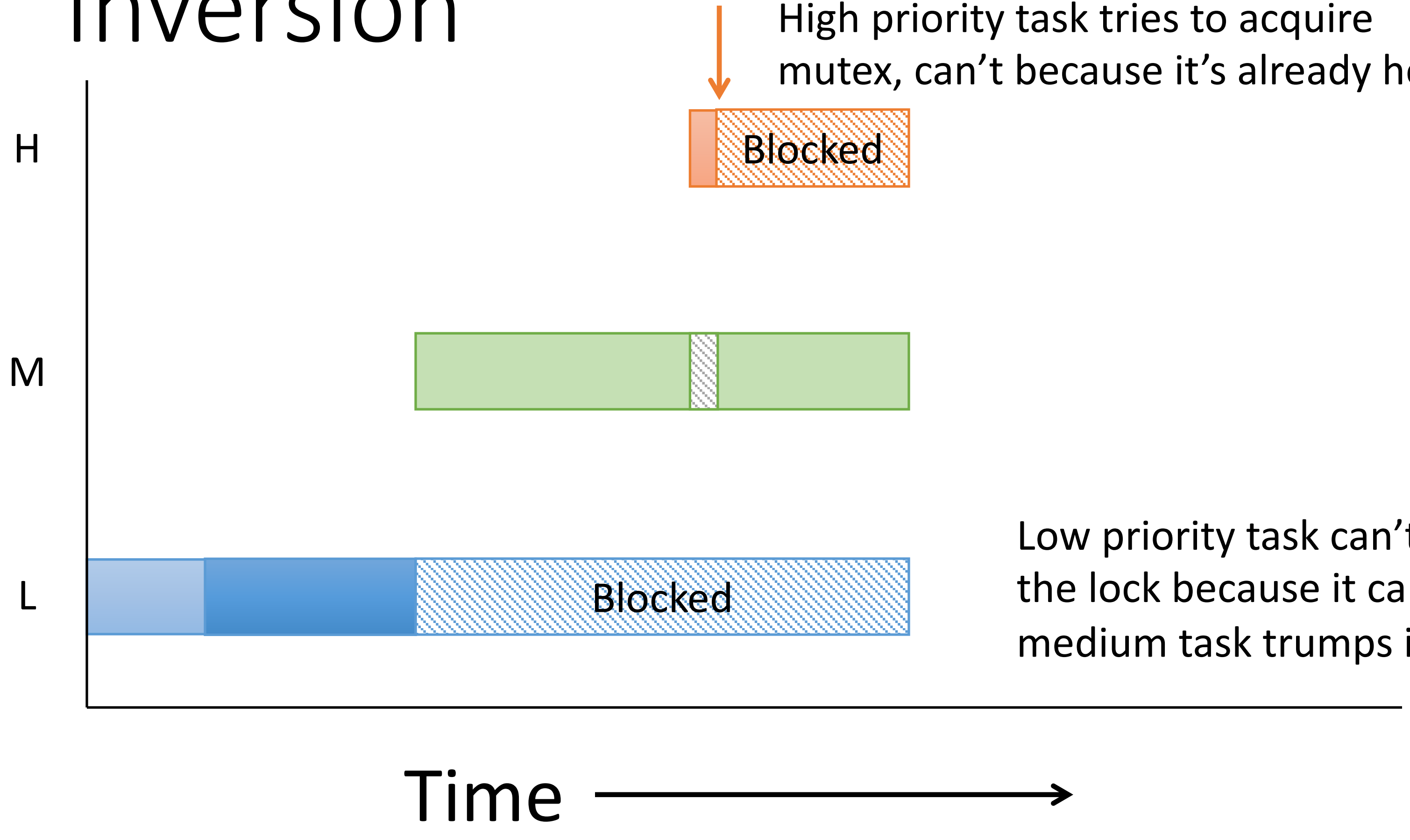
Medium task starts up, takes CPU.

What Happened: Priority Inversion

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



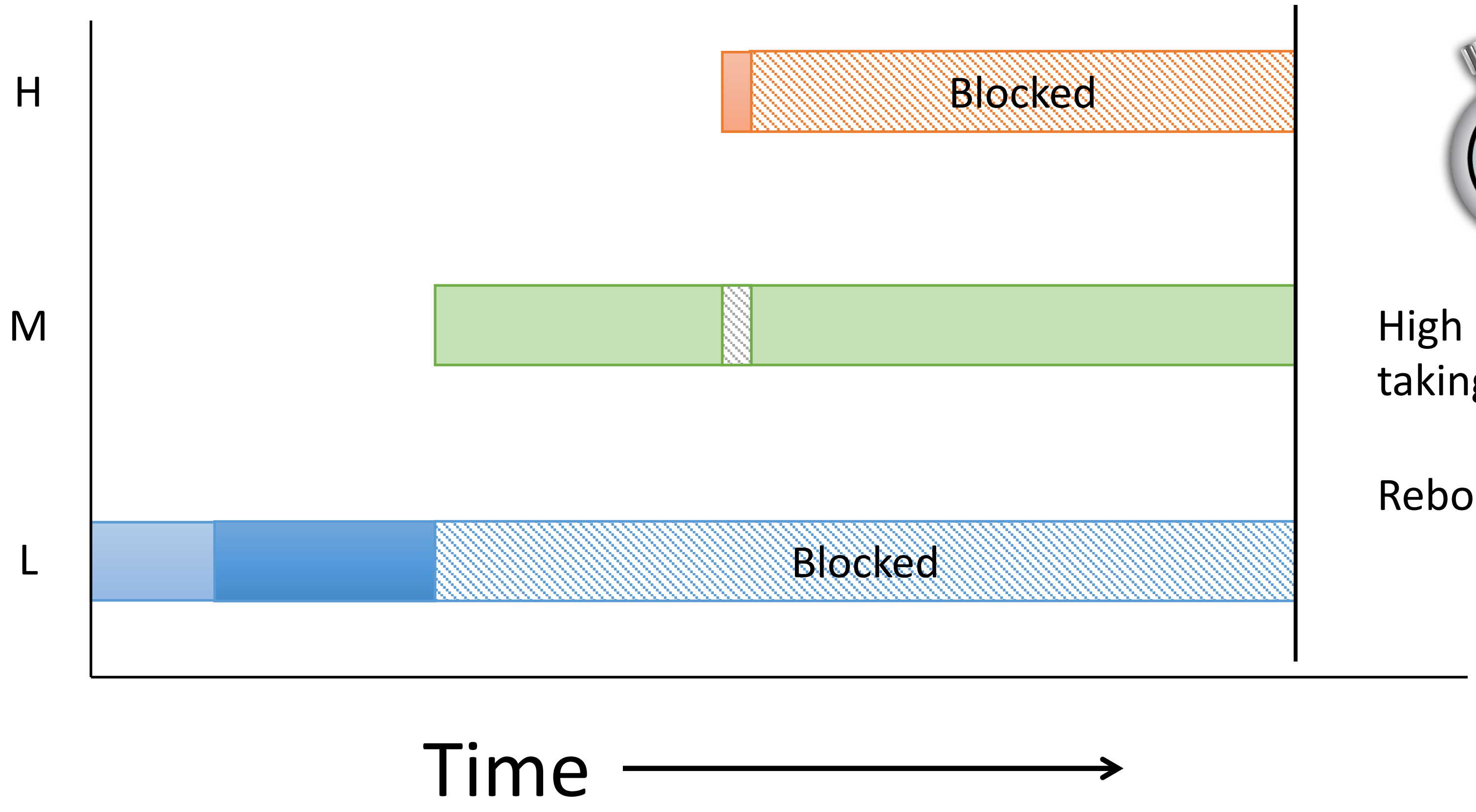
What Happened: Priority Inversion



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.

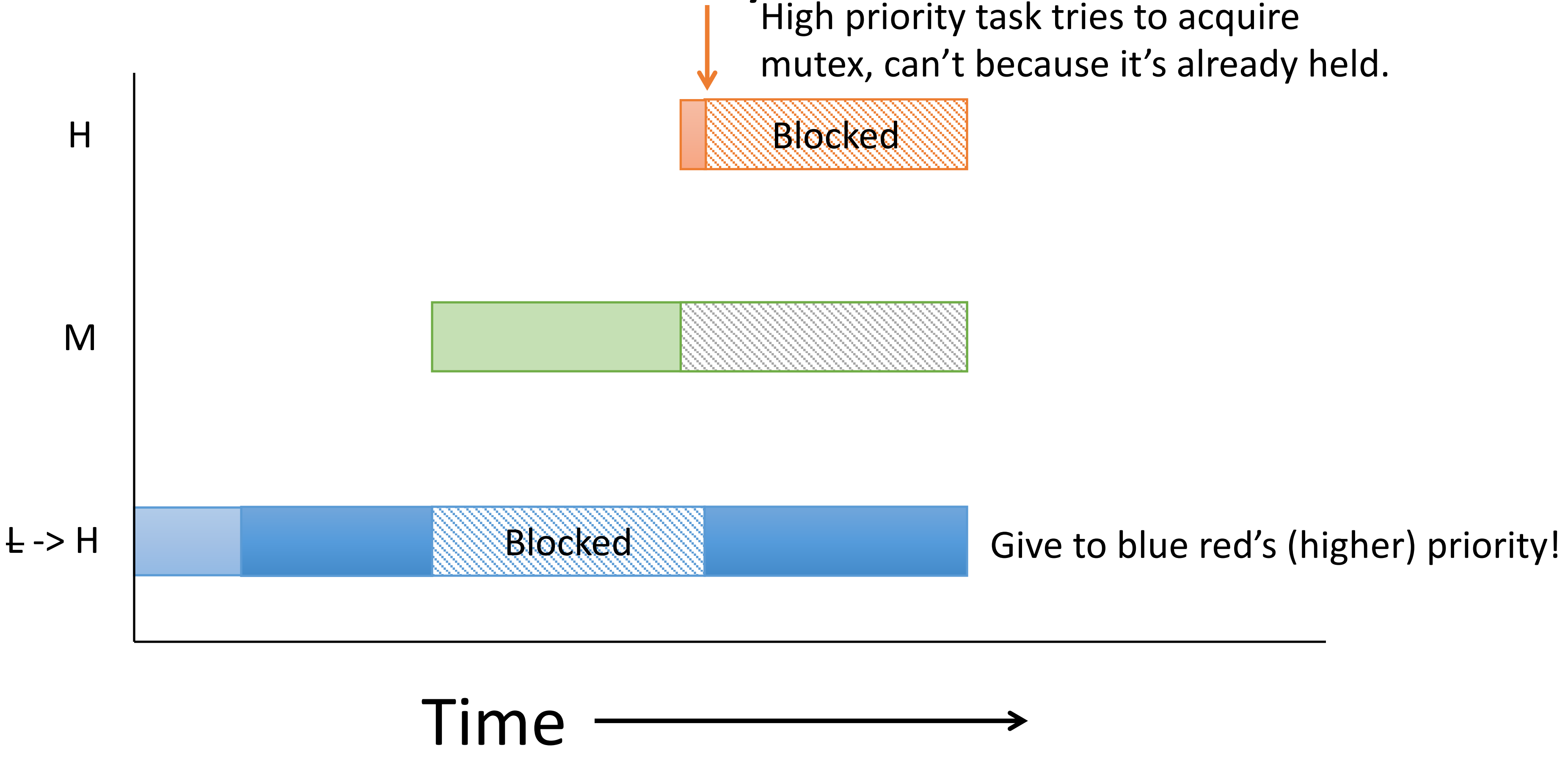
What Happened: Priority Inversion



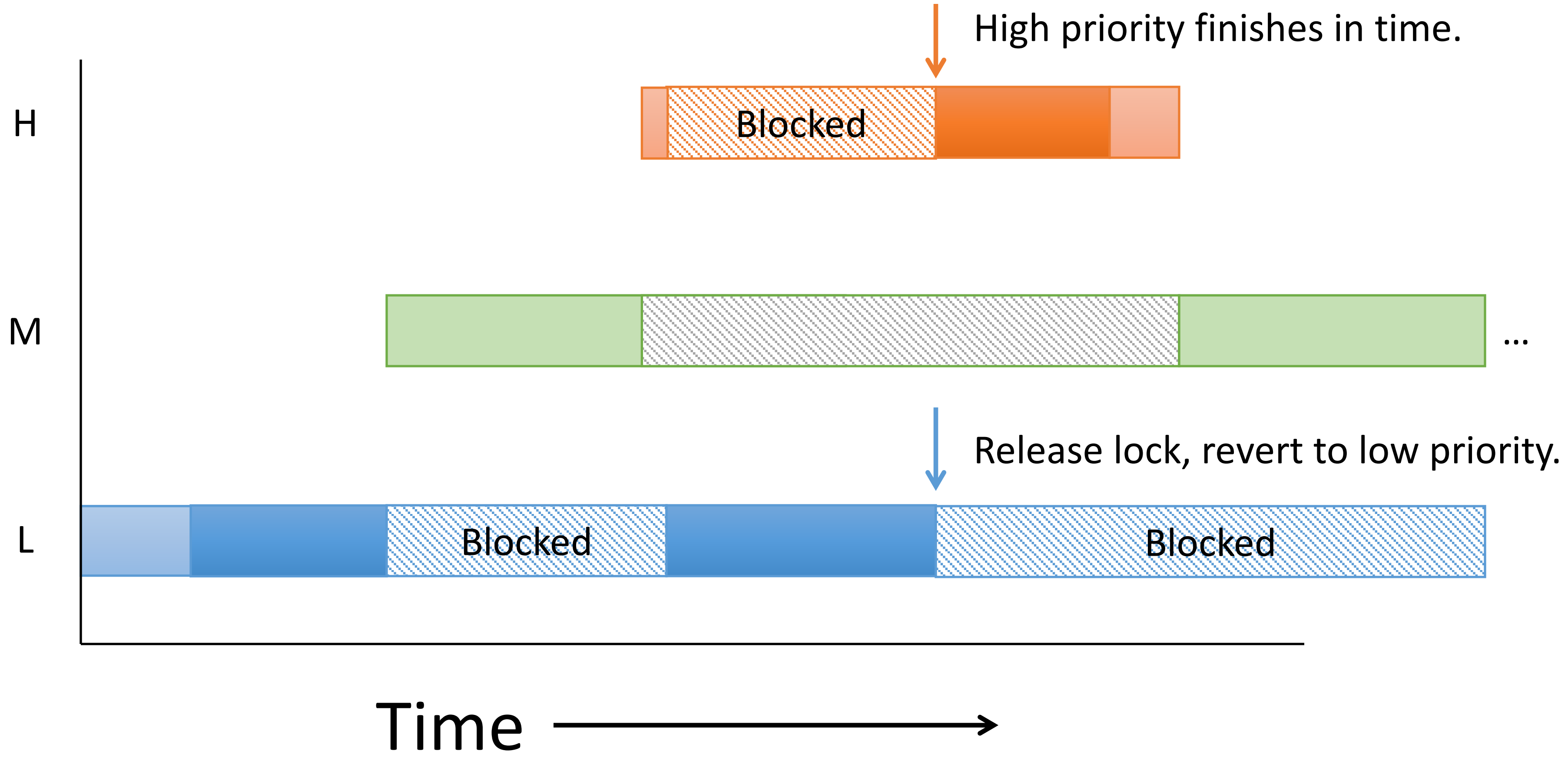
High priority is taking too long.

Reboot!

Solution: Priority Inheritance


High priority task tries to acquire mutex, can't because it's already held.
H
Blocked
M
~~L~~ -> H
Blocked
Give to blue red's (higher) priority!
Time

Solution: Priority Inheritance


High priority finishes in time.
H
Blocked
M
…
Release lock, revert to low priority.
L
Blocked
Blocked
Time

What’s wrong with this mutex?

```
lock:
  cmp $0 %ebx #check mutex
  jne lock    #wait
  mov $1 %ebx #lock mutex

...

  mov $0 %ebx #unlock mutex
```

We need an atomic read-modify-write operation.

Two that are commonly implemented in hardware:

- Compare-and-swap
    - Swap two memory locations only if the first has a specific value. Return success or failure.
- Test-and-set
    - Set a memory bit to 1 and return its old value.

Mutex with test-and-set

```
void Lock(int *lock) {
  while (test_and_set(lock) == 1);
}

void Unlock(int *lock) {
  *lock = 0);
}
```

What assembly would this translate to?

Exercise: write an assembly mutex using compare-and-swap.

`CMPXCHG`

Compares the value in the AL, AX, or EAX register (depending on the size of the operand) with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, or EAX register.

The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are set according to the results of the comparison operation.