CS 31: Intro to Systems
Deadlock

Kevin Webb
Swarthmore College

December 4, 2018

“Deadly Embrace”

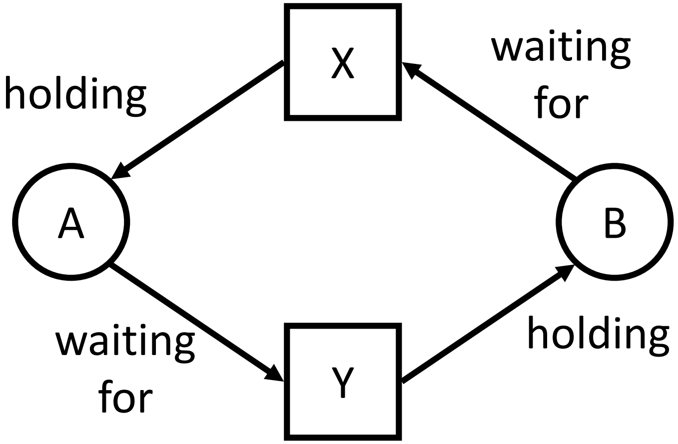
- *The Structure of the THE-Multiprogramming System* (Edsger Dijkstra, 1968)
- Also introduced semaphores
- Deadlock is as old as synchronization

What is Deadlock?

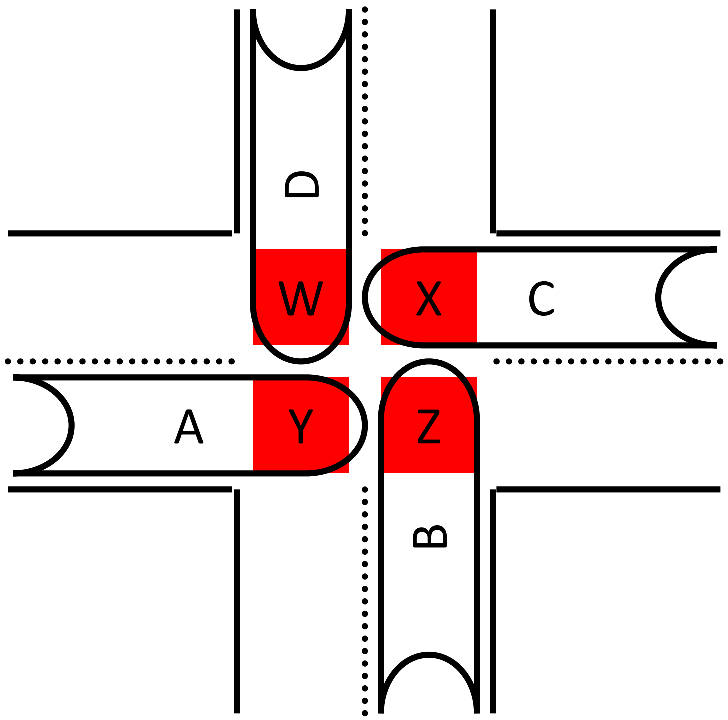
- Deadlock is a problem that can arise:
  - When processes compete for access to limited resources
  - When threads are incorrectly synchronized
- Definition:
  - Deadlock exists among a set of threads if every thread is waiting for an event that can be caused only by another thread in the set.

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



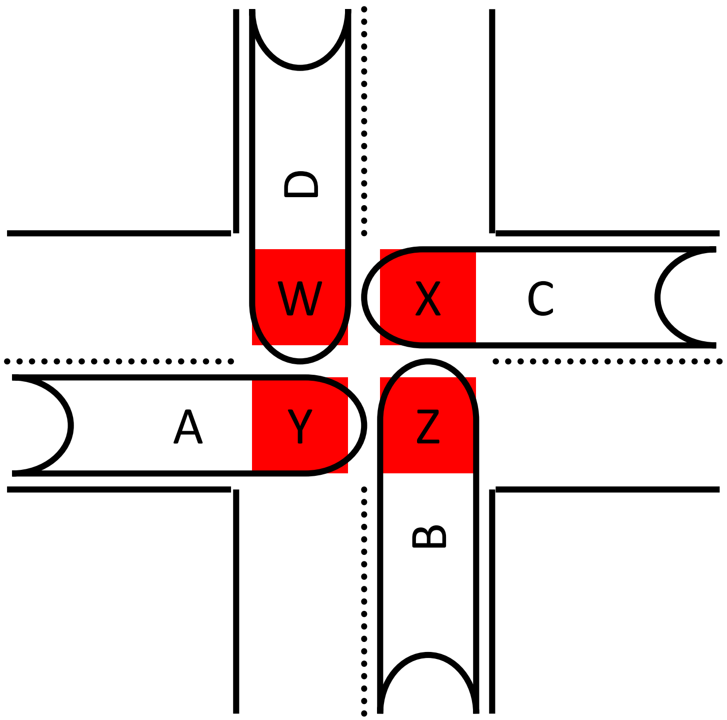
Traffic Jam as Example of Deadlock



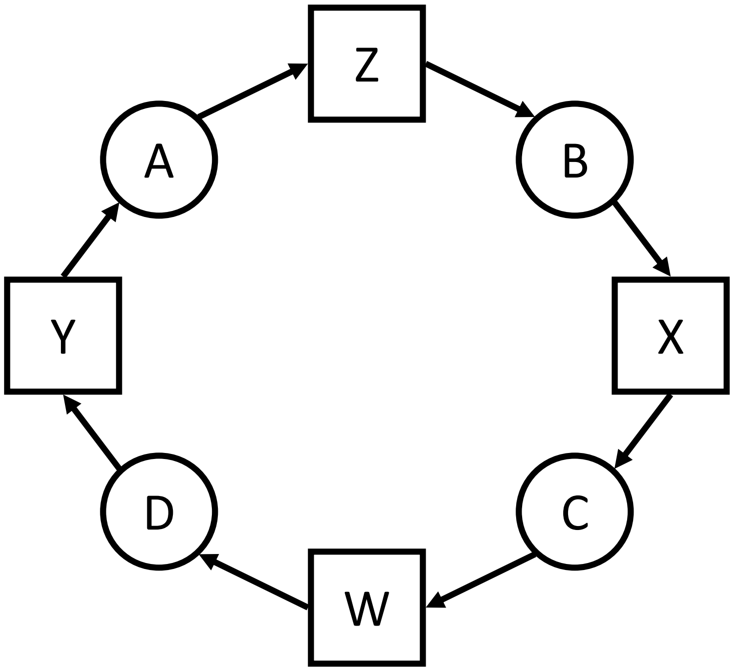
Cars deadlocked
in an intersection

- 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

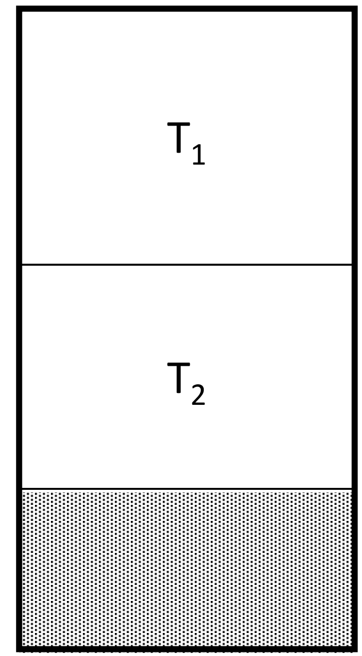
1. Mutual Exclusion
   – Only one thread may use a resource at a time.
2. Hold-and-Wait
   – Thread holds resource while waiting for another.
3. No Preemption
   – Can’t take a resource away from a thread.
4. Circular Wait
   – The waiting threads form a cycle.

Four Conditions for Deadlock

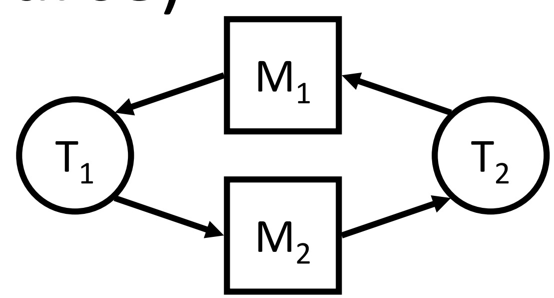
1. Mutual Exclusion
   – Only one thread may use a resource at a time.
2. Hold-and-Wait
   – Thread holds resource while waiting for another.
3. No Preemption
   – Can’t take a resource away from a thread.
4. Circular Wait
   – The waiting threads form a cycle.

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);
```

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?

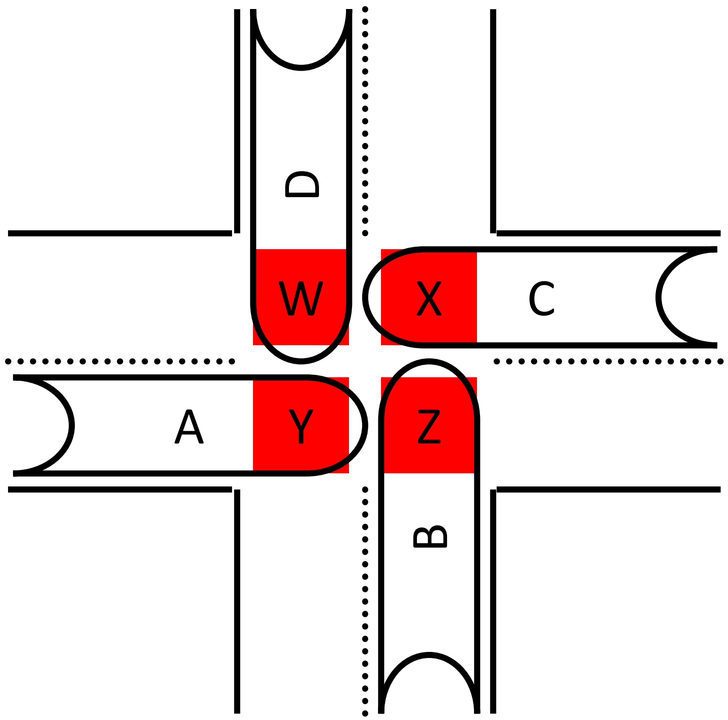
- Deadlock Prevention
  - Make deadlock impossible by removing a condition
- Deadlock Avoidance
  - Avoid getting into situations that lead to deadlock
- Deadlock Detection
  - Don’t try to stop deadlocks
  - Rather, if they happen, detect and resolve

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

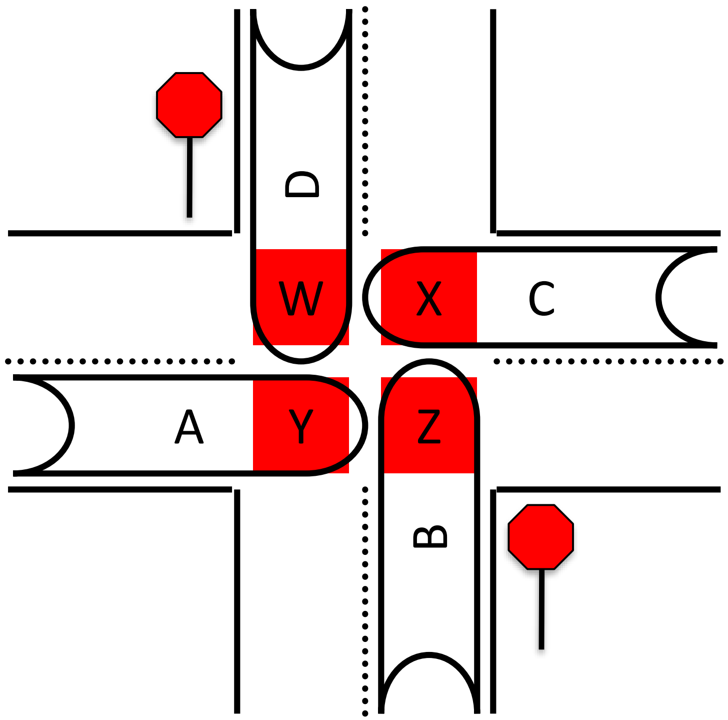
How Can We Prevent a Traffic Jam?



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

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!

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 Avoidance

- Avoid situations that lead to deadlock
  - Selective prevention
  - Remove condition <u>only when deadlock a possibility</u>
- Works with incremental resource requests
  - Resources are asked for in increments
  - Do not grant request that can lead to a deadlock
- Requires knowledge of maximum resource requirements

Banker’s Algorithm

- Fixed number of threads and resources
  - Each thread has zero or more resources allocated
- System state: either safe or unsafe
  - Depends on allocation of resources to threads

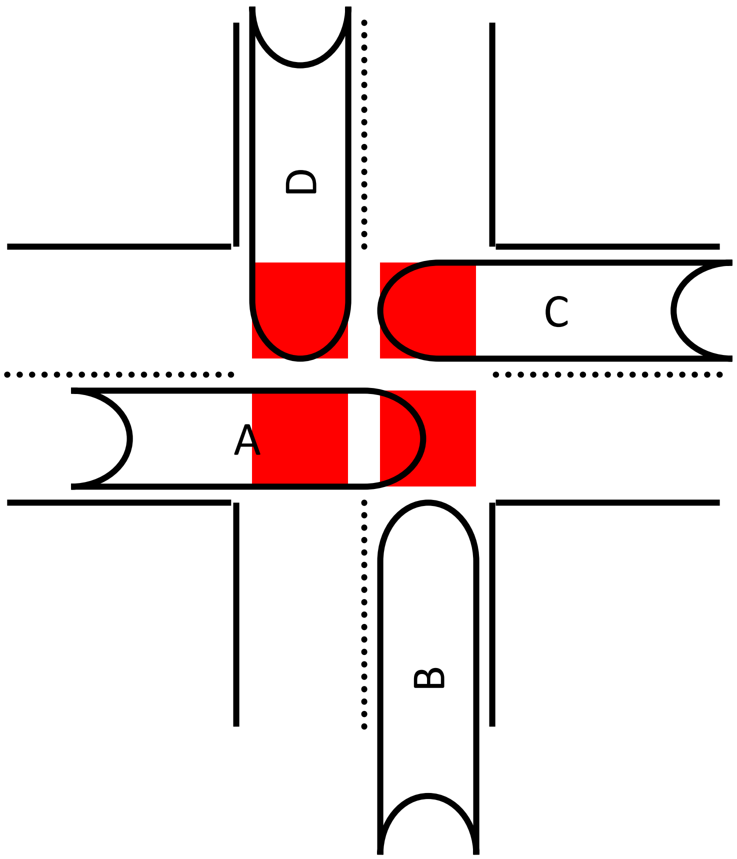
- Safe: deadlock is absolutely avoidable
  - Can avoid deadlock by certain order of execution
- Unsafe: deadlock is possible (but not certain)
  - May not be able to avoid deadlock

Banker’s Algorithm for Avoidance

- The Banker’s Algorithm is the classic approach to deadlock avoidance for resources with multiple units
  - 1. Assign a credit limit to each customer (thread)
    - Maximum credit claim must be stated in advance
  - 2. Reject any request that leads to a dangerous state
    - A dangerous state is one where a sudden request by any customer for the full credit limit could lead to deadlock
    - A recursive reduction procedure recognizes dangerous states
  - 3. In practice, the system must keep resource usage well below capacity to maintain a resource surplus

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

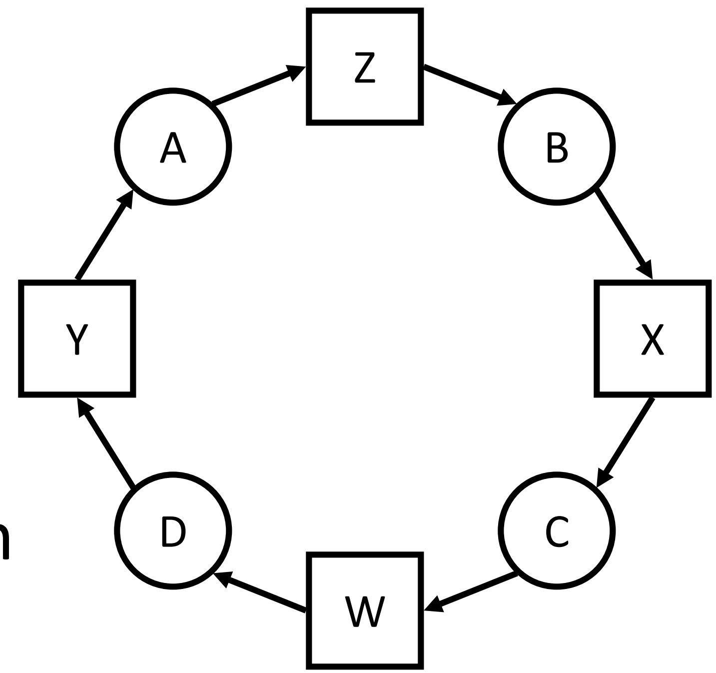
- Deadlock Prevention
  - Make deadlock impossible by removing a condition
- Deadlock Avoidance
  - Avoid getting into situations that lead to deadlock
- Deadlock Detection
  - Don’t try to stop deadlocks
  - Rather, if they happen, detect and resolve

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)

Detecting a Deadlock

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



Recovery from Deadlock

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

Recovery from Deadlock

- Abort all deadlocked threads / processes
  - Will remove deadlock, but drastic and costly
- Abort deadlocked threads one-at-at-time
  - Do until deadlock goes away (need to detect)
  - What order should threads be aborted?

Recovery from Deadlock

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

- 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

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

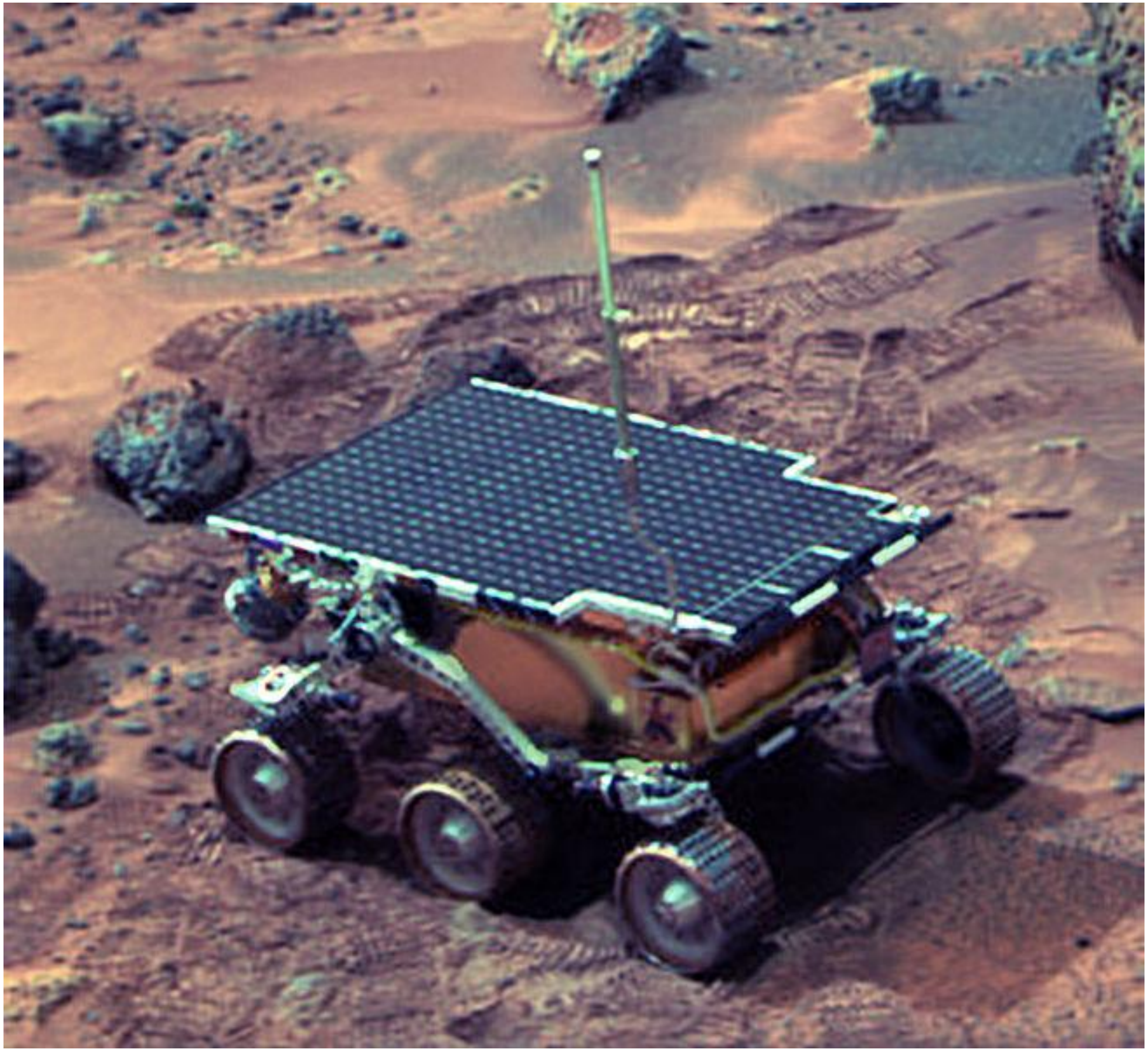
- <u>These all have major drawbacks…</u>

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)



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

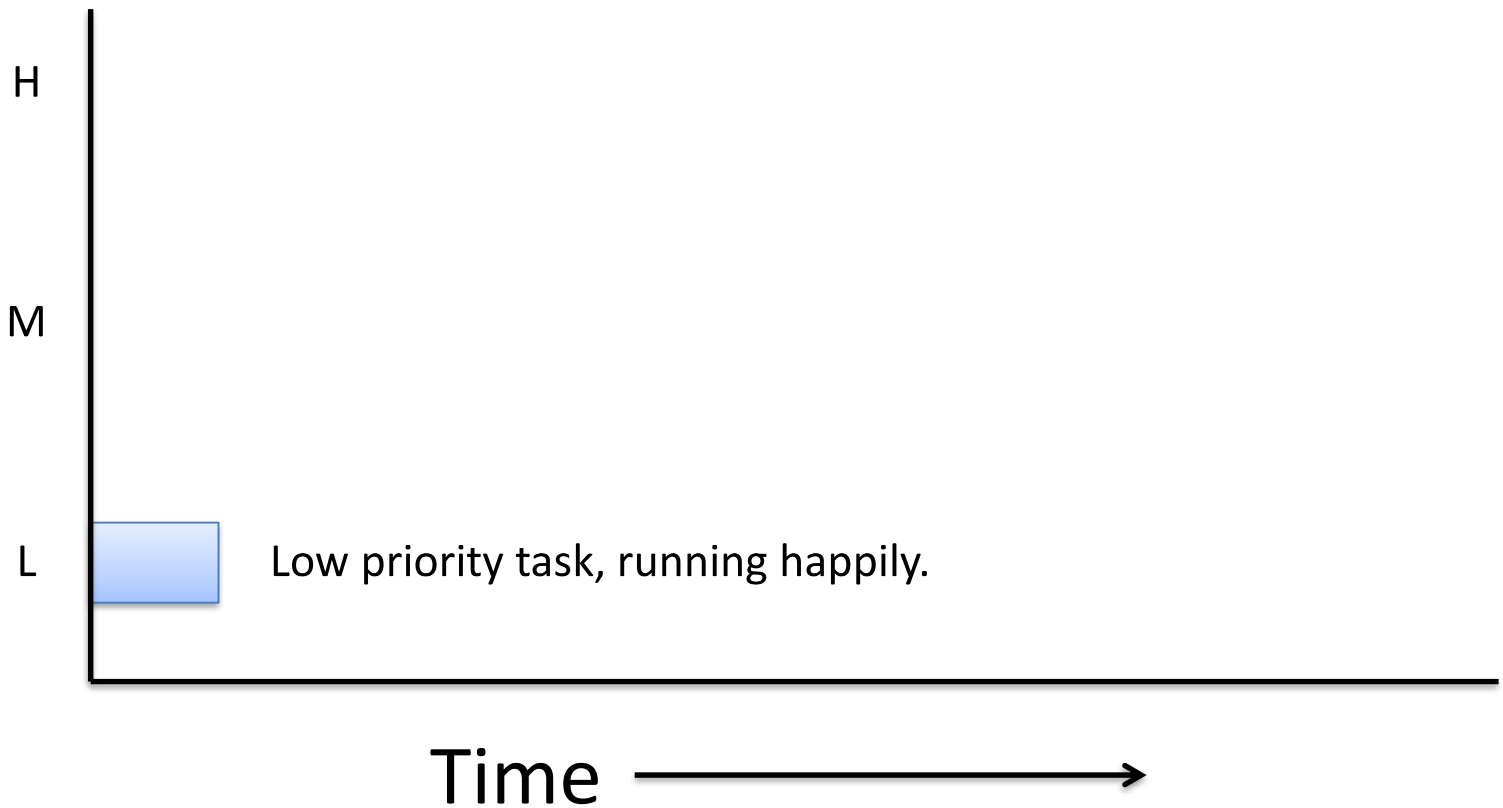
- Tasks 1 and 3 require exclusive access to a hardware bus to move data.
  – Bus protected by a mutex.

Mars Rover

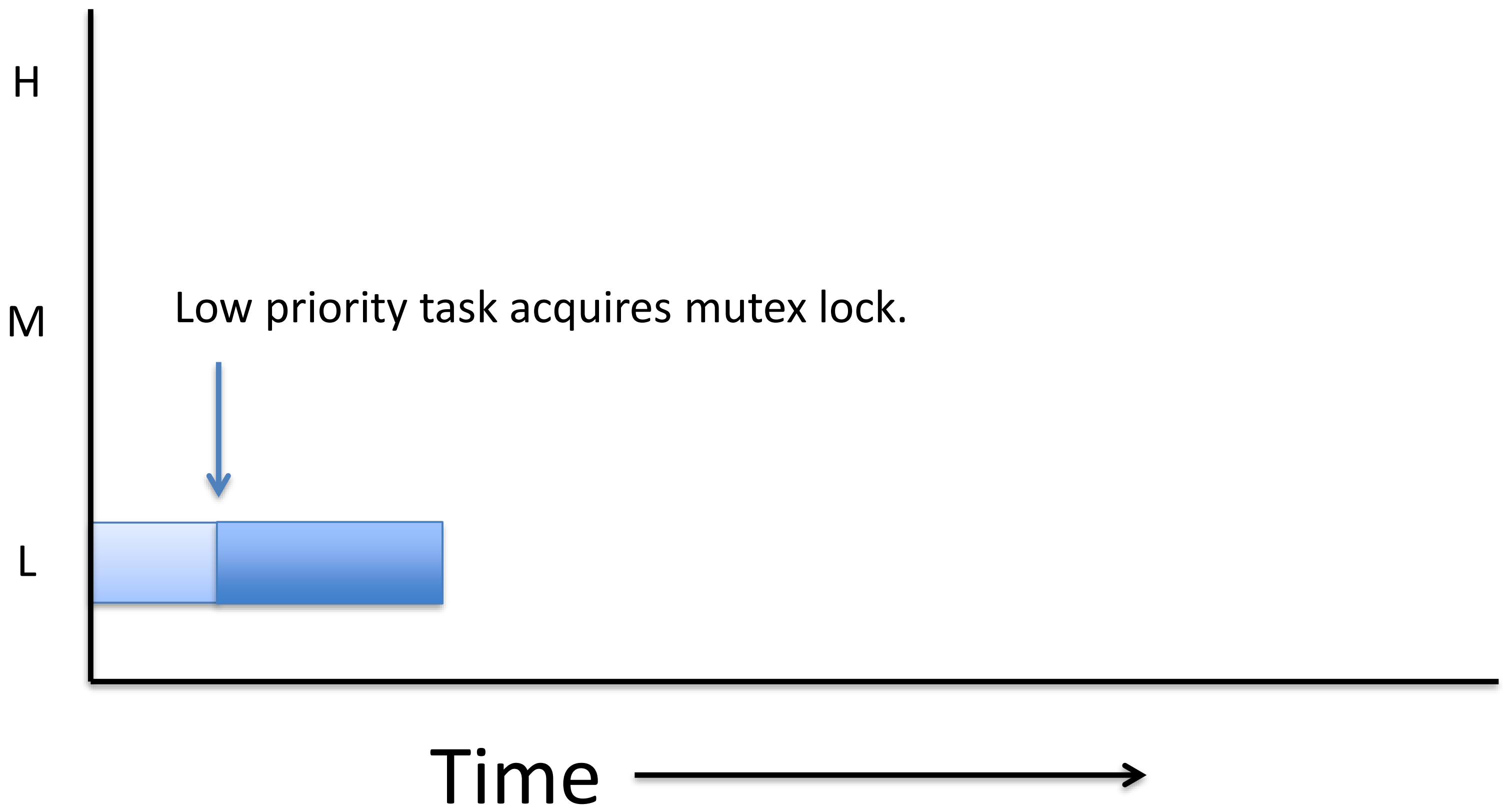
- Failsafe timer (watchdog): if high priority task doesn’t complete in time, reboot system
- Observation: uh-oh, this thing seems to be rebooting a lot, we’re losing data…

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".

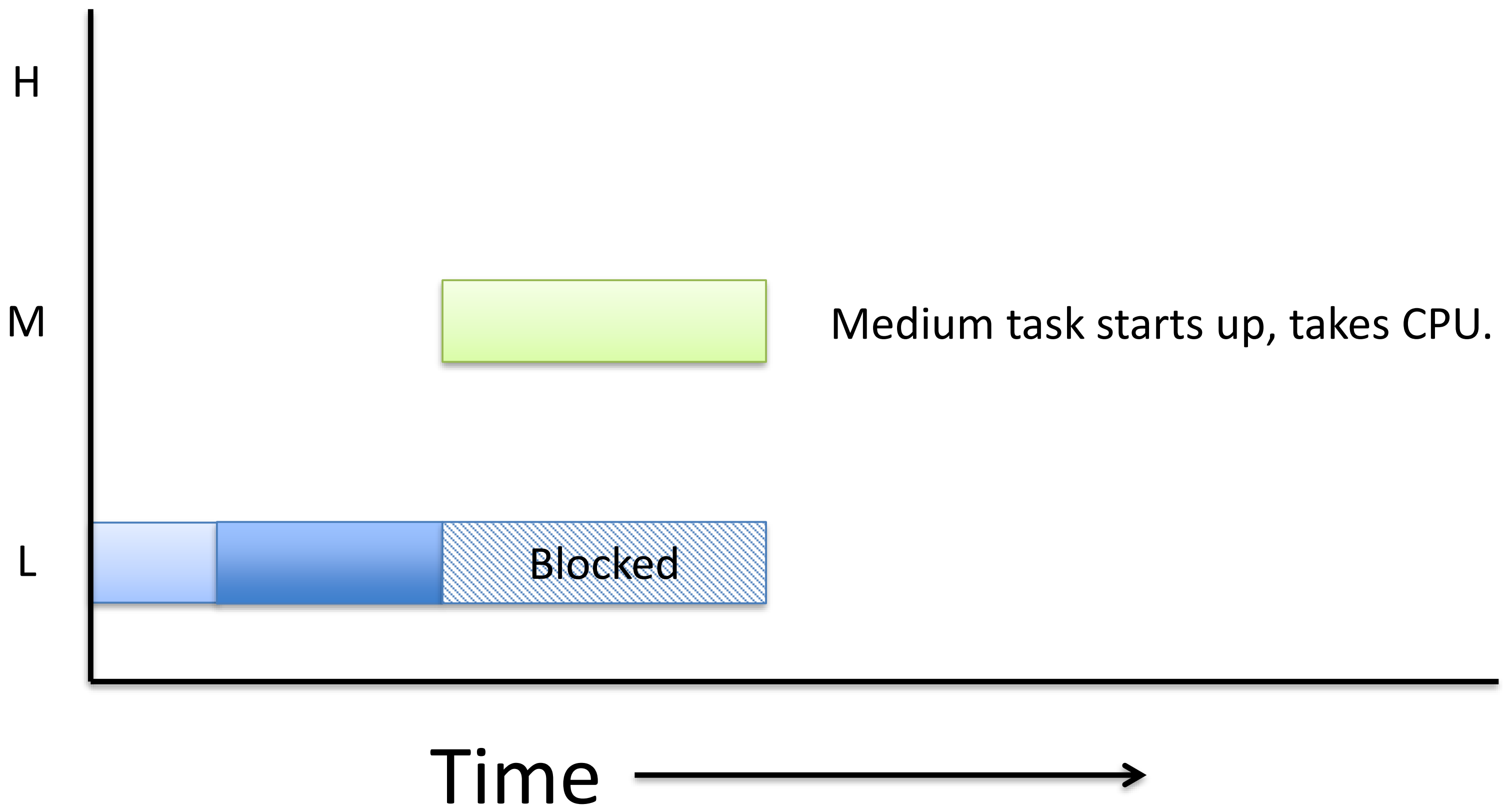
What Happened: Priority Inversion



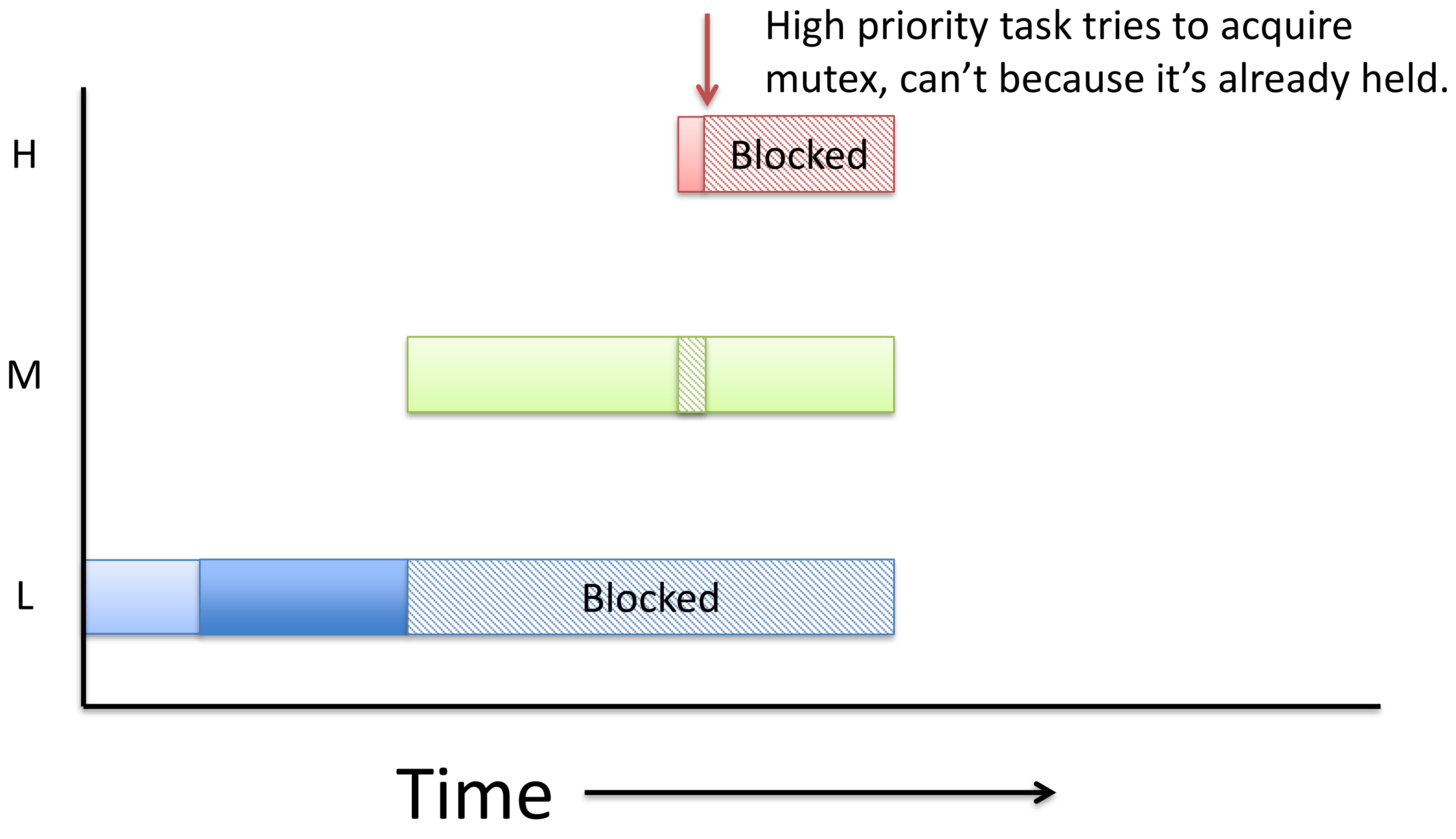
What Happened: Priority Inversion



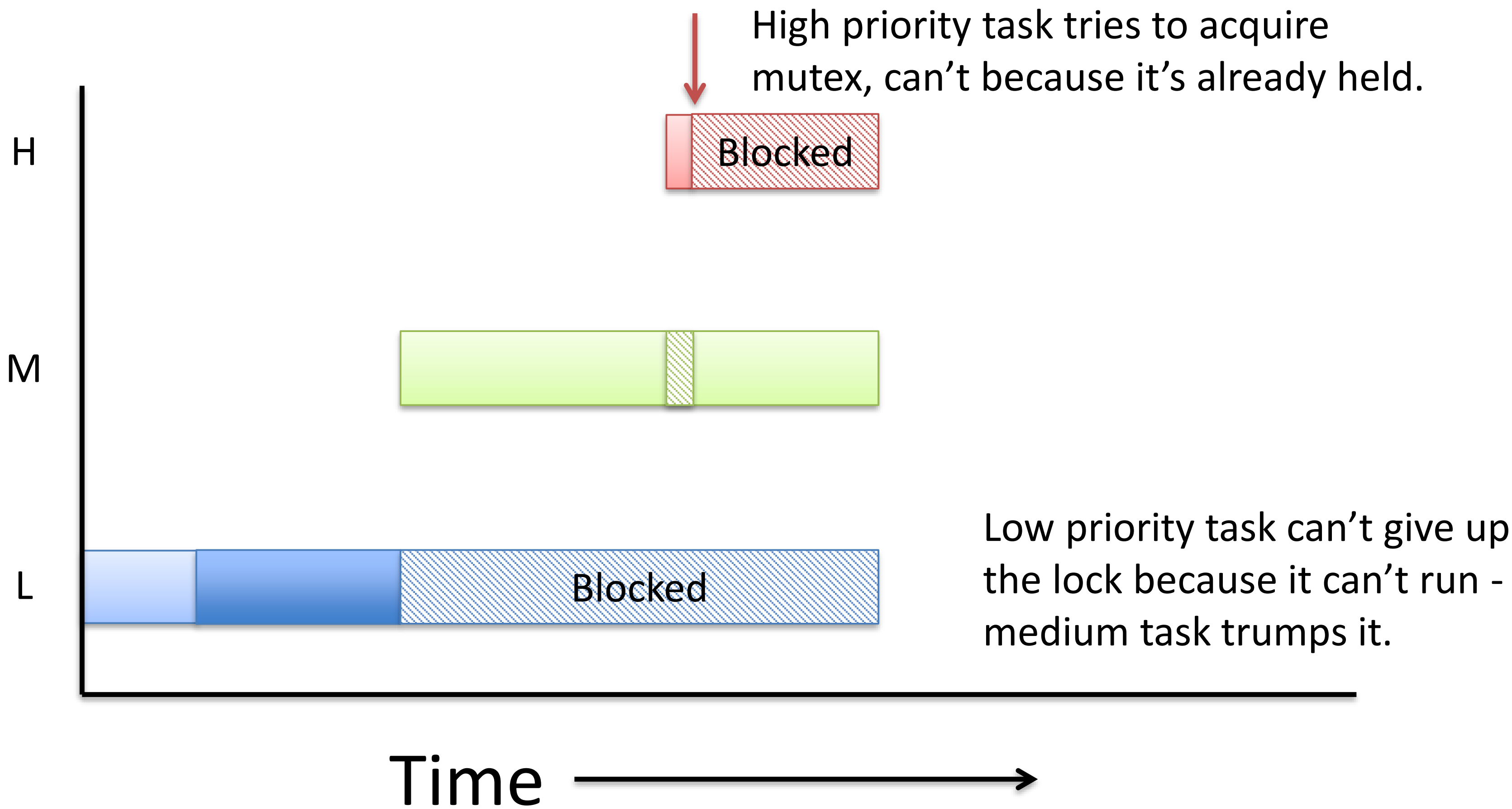
What Happened: Priority Inversion



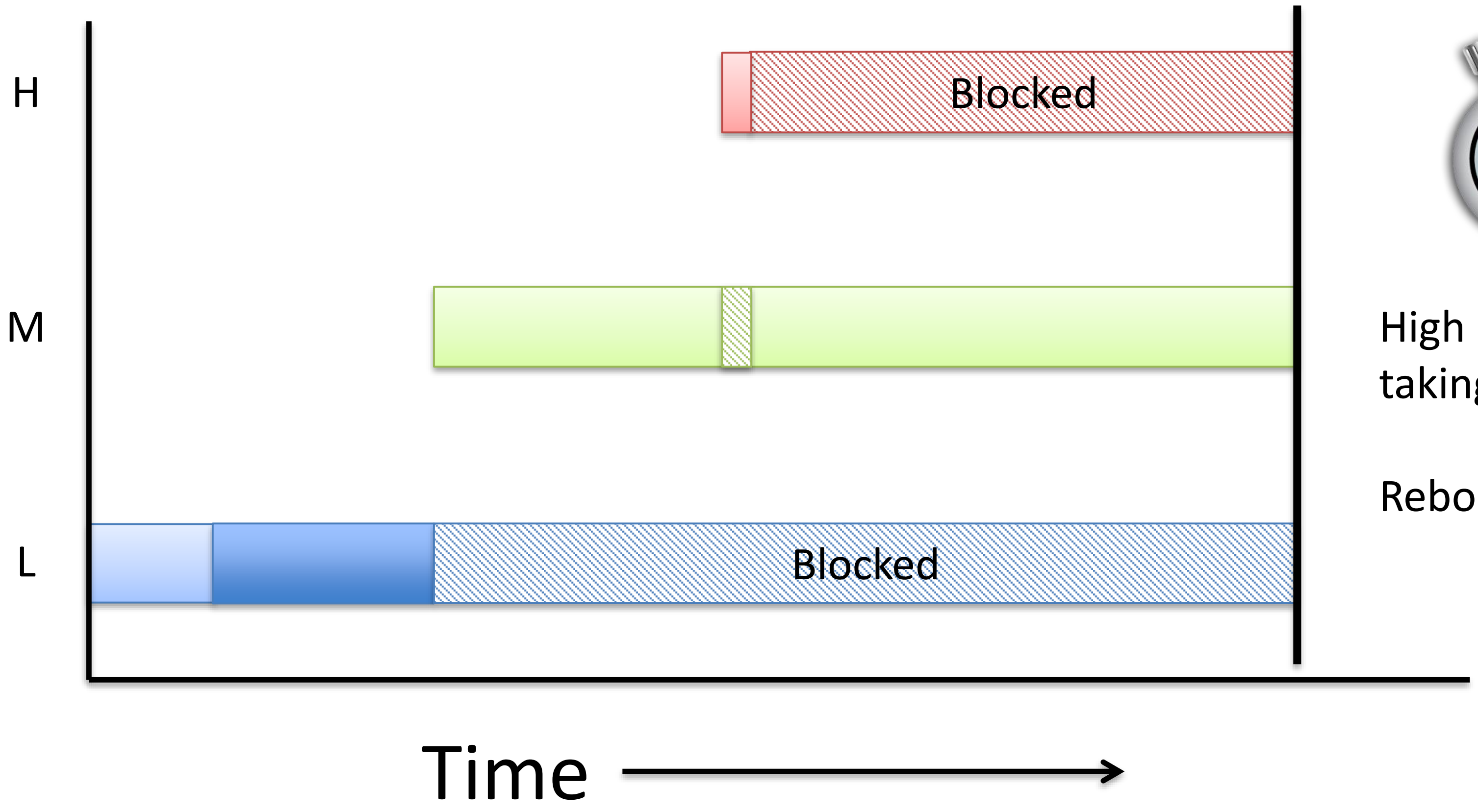
What Happened: Priority Inversion



What Happened: Priority Inversion



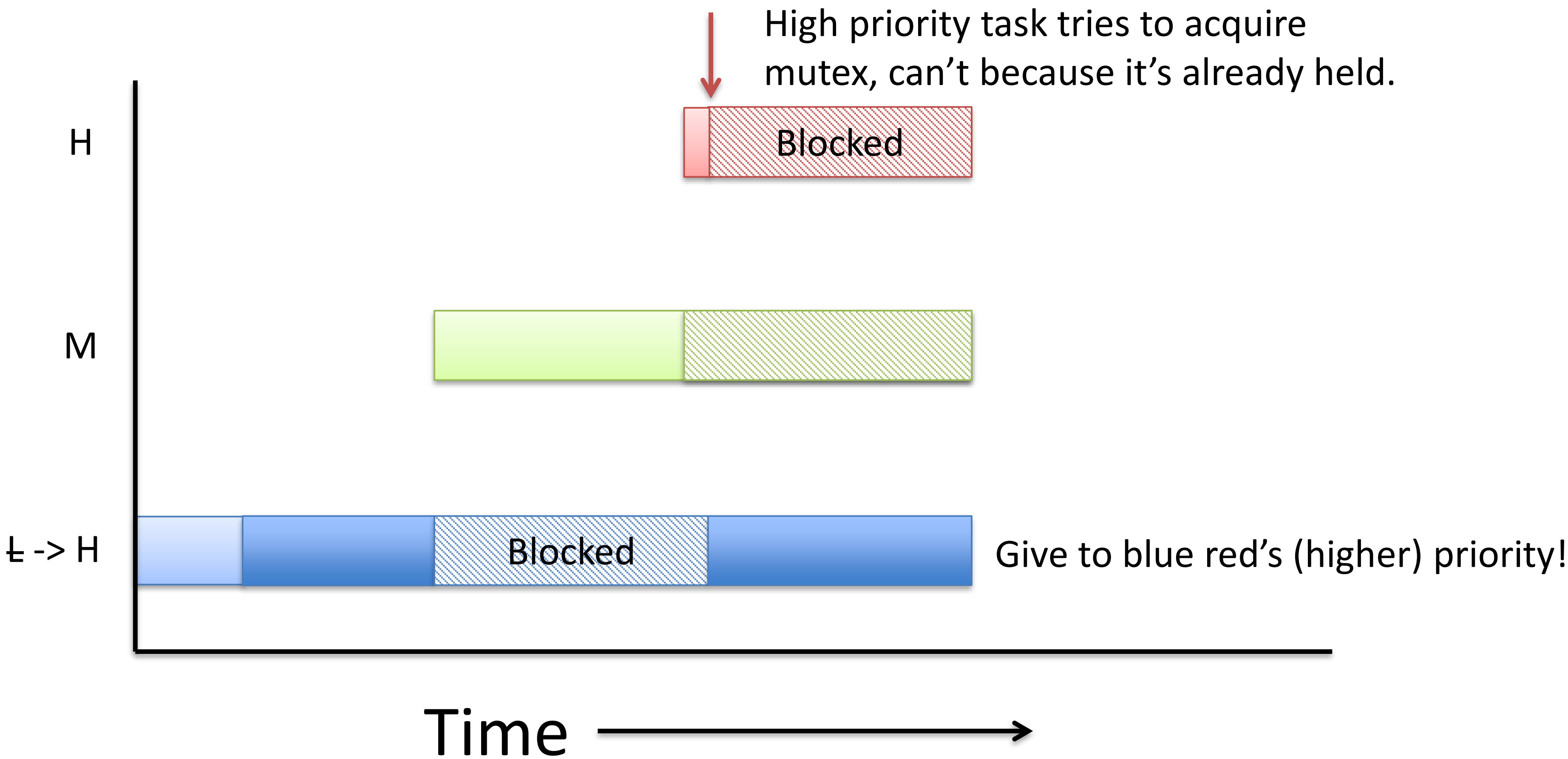
What Happened: Priority Inversion



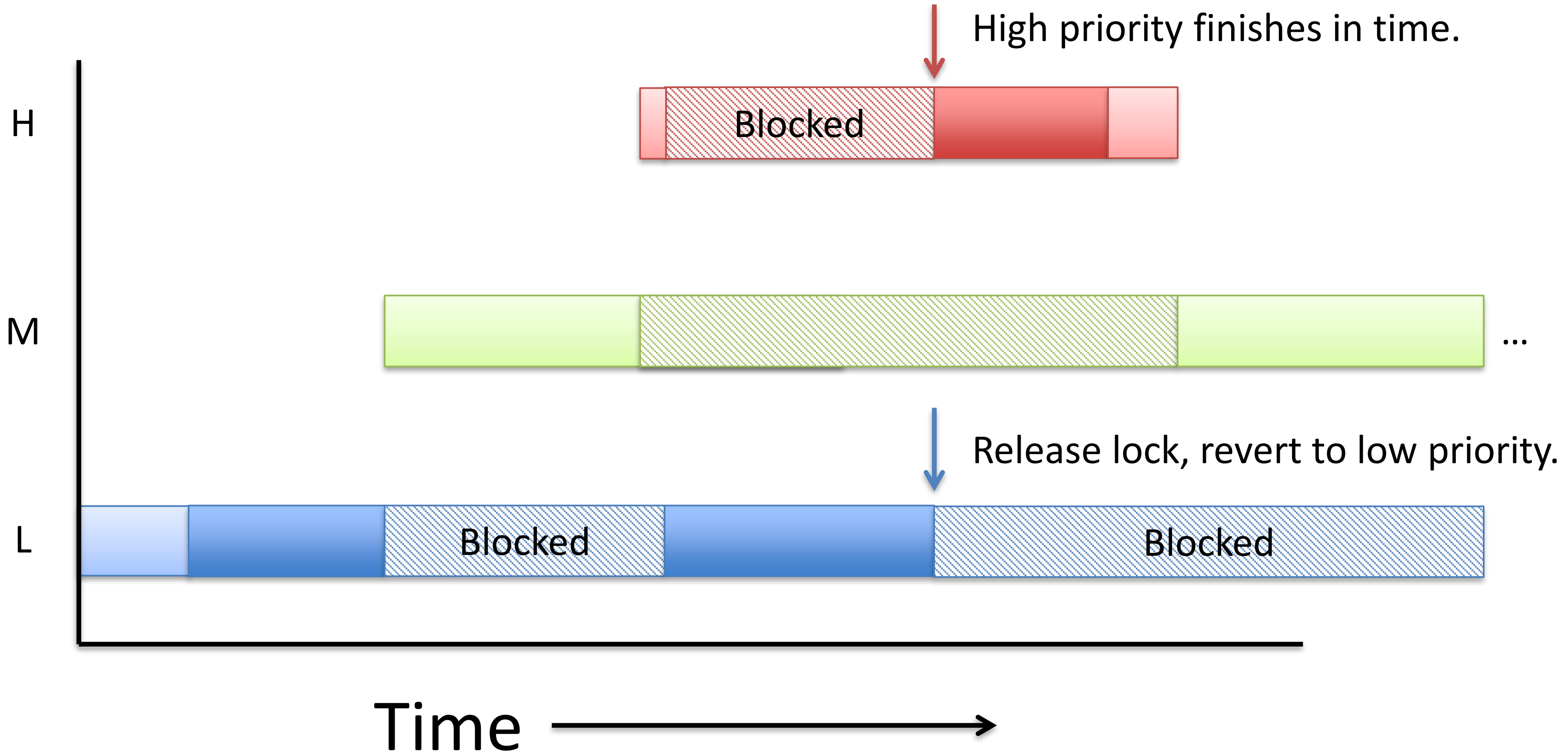
High priority is taking too long.

Reboot!

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