Synchronization

Heechul Yun

Disclaimer: some slides are adopted from the book authors and Dr. Kulkani

Recap

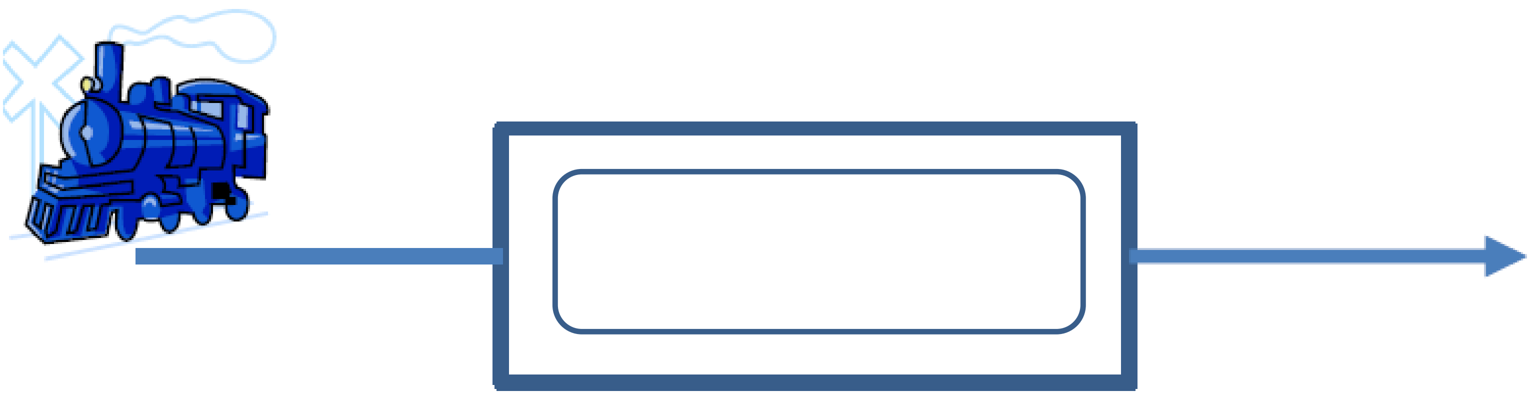
- Semaphore
  - Blocking
  - Binary semaphore = mutex
  - Integer semaphore
  - Solved synchronization problems
    - Bounded-buffer
      - 2 Integer semaphores, 1 binary semaphore
    - Train control
      - 1 integer semaphore
    - Reader/write
      - 2 binary semaphores

KU THE UNIVERSITY OF KANSAS

Recap: Train Control Problem

- No more than two trains can enter the rail

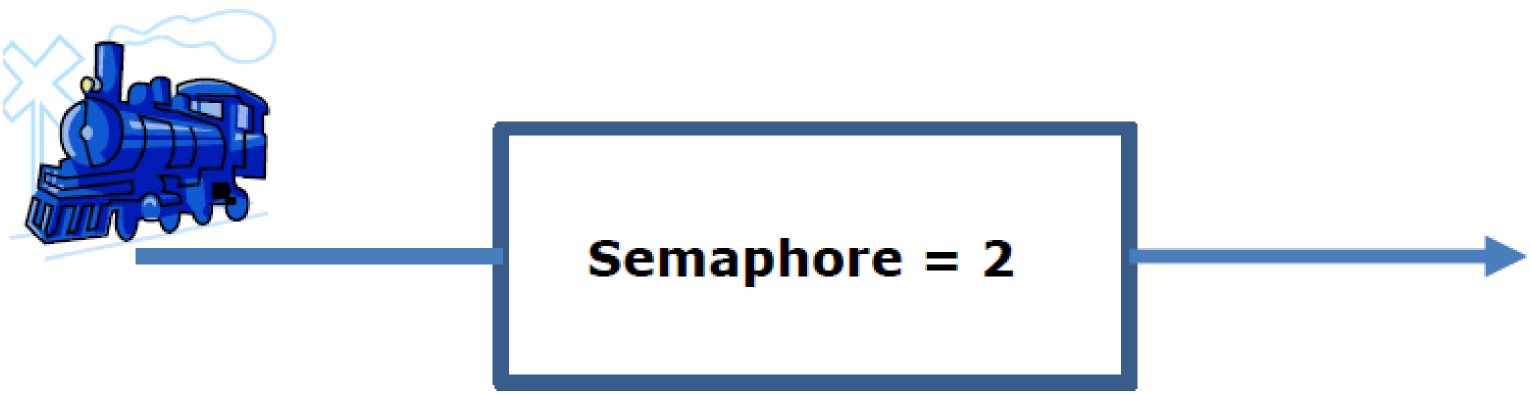
```
Enter() {      }
Leave() {      }
```



Recap: Train Control Problem

- No more than two trains can enter the rail

```
Enter() { P(); }
Leave() { V(); }
```



Quiz

- Semaphore mutex = 1;
- Semaphore **full = 0;**
- Semaphore **empty = N;**

Producer

```
do {
    Produce new resource
    _________;
    mutex.P();
    Add resource to next buffer
    mutex.V();
    _________;
} while (TRUE);
```

Consumer

```
do {
    _________;
    mutex.P();
    Remove resource from buffer
    mutex.V();
    _________;
    Consume resource
} while (TRUE);
```

Quiz

- Semaphore mutex = 1;
- Semaphore **full = 0;**
- Semaphore **empty = N;**

Producer

```
do {
    Produce new resource
    empty.P();
    mutex.P();
    Add resource to next buffer
    mutex.V();
    full.V();
} while (TRUE);
```

Consumer

```
do {
    full.P()
    mutex.P();
    Remove resource from buffer
    mutex.V();
    empty.V();
    Consume resource
} while (TRUE);
```

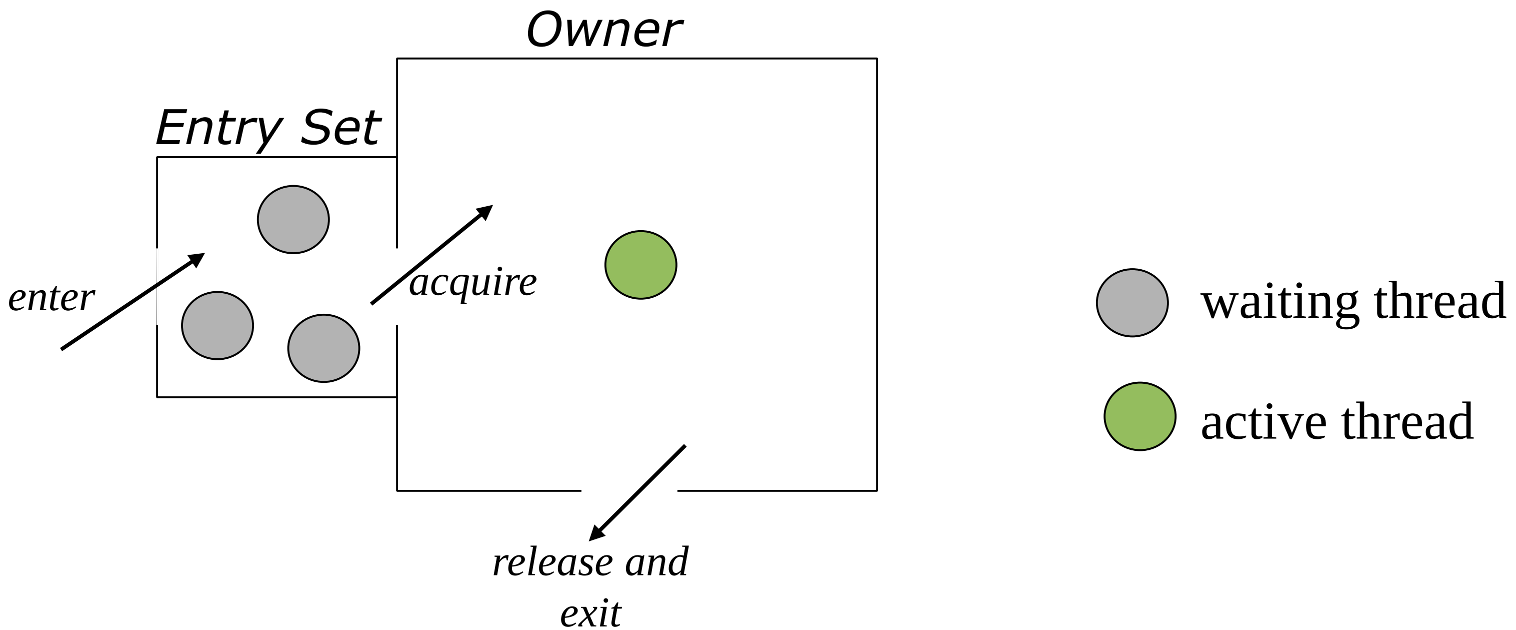
Monitor

- Monitor
  - A lock (mutual exclusion) + condition variables (scheduling)
  - Some languages like Java natively support this, but you can use monitors in other languages like C/C++
- Lock: mutual exclusion
  - Protects the shared data structures inside the monitor
  - Always acquire it to enter the monitor
  - Always release it to leave the monitor
- Condition Variable: scheduling
  - Allow thread to wait on certain events inside the monitor
  - Key idea: to wait (sleep) inside the monitor, it first releases the lock and go to sleep atomically

KU THE UNIVERSITY OF KANSAS

Monitor

- Lock: mutual exclusion
  - Only one thread can execute any monitor procedure at a time.
  - Other threads invoking a monitor procedure when one is already executing some monitor procedure must wait.
  - When the active thread exits the monitor procedure, one other waiting thread can enter.



A Simple Monitor Example

C++

```
Mutex lock;
Queue queue;

produce (item)
{
   lock.acquire();
   queue.enqueue(item);
   lock.release();
}

consume()
{
   lock.acquire();
   item = queue.dequeue(item);
   lock.release();
   return item;
}
```

A Simple Monitor Example

C++

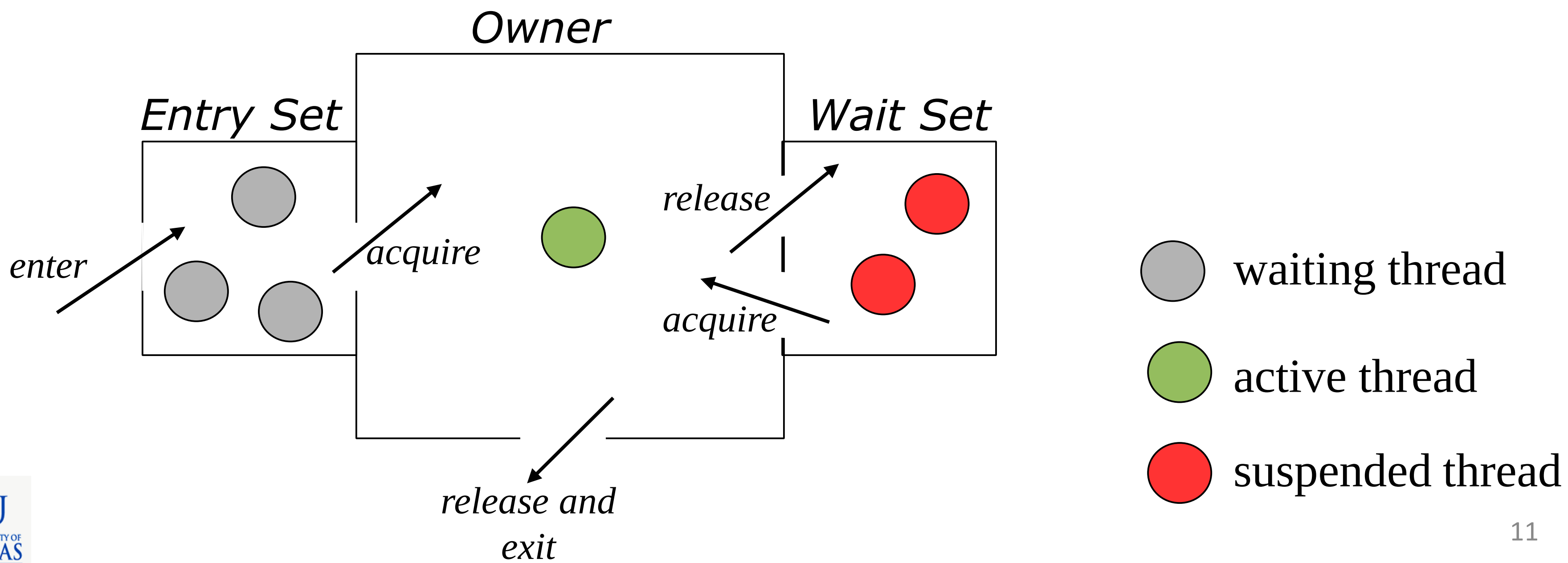
```
Mutex lock;
Queue queue;

produce (item)
{
   lock.acquire();
   queue.enqueue(item);
   lock.release();
}

consume()
{
   lock.acquire();
   item = queue.dequeue(item);
   lock.release();
   return item;
}
```

Java

```
Queue queue;

Synchronized produce (item)
{

   queue.enqueue(item);

}

Synchronized consume()
{

   item = queue.dequeue(item);

   return item;
}
```

Monitor

- What if a thread needs to wait inside a monitor?
- Condition variable: Scheduling
  - Wait(&lock): atomically release the lock and sleep; Re-acquire the lock on returning.
  - Signal(): wake-up one waiter, if exists
  - Broadcast(): wake-up all waiters



Monitor with Condition Variable

```
Mutex lock;
Condition full;
Queue queue;

produce (item)
{
   lock.acquire();
   queue.enqueue(item);
   full.signal();
   lock.release();
}

consume()
{
   lock.acquire();
   while (queue.isEmpty())
      full.wait(&lock);
   item = queue.dequeue(item);
   lock.release();
   return item;
}
```

Why not 'if'?

Semantics

- Hoare monitors (original)
  - Signaler immediately switches to the waiting thread
  - Waiter's condition is guaranteed to hold when it resumes
- Mesa monitors
  - Waiter is simply placed on ready queue, signaler continues
  - Waiter's condition may no longer be true when it resumes
- Almost always Mesa style in practice

Bounded Buffer Problem Revisit

```
Mutex lock;
Condition full, empty;

produce (item)
{
   lock.acquire();
…
…
   queue.enqueue(item);
   full.signal();
   lock.release();
}

consume()
{
   lock.acquire();
   while (queue.isEmpty())
      full.wait(&lock);
   item = queue.dequeue(item);
…
   lock.release();
   return item;
}
```

Bounded Buffer Problem Revisit

```
Mutex lock;
Condition full, empty;

produce (item)
{
    lock.acquire();
    while (queue.isFull())
        empty.wait(&lock);
    queue.enqueue(item);
    full.signal();
    lock.release();
}

consume()
{
    lock.acquire();
    while (queue.isEmpty())
        full.wait(&lock);
    item = queue.dequeue(item);
    empty.signal();
    lock.release();
    return item;
}
```

Bounded Buffer Problem Revisit

Monitor version

```
Mutex lock;
Condition full, empty;

produce (item)
{
    lock.acquire();
    while (queue.isFull())
        empty.wait(&lock);
    queue.enqueue(item);
    full.signal();
    lock.release();
}

consume()
{
    lock.acquire();
    while (queue.isEmpty())
        full.wait(&lock);
    item = queue.dequeue(item);
    empty.signal();
    lock.release();
    return item;
}
```

Semaphore version

```
Semaphore mutex = 1, full = 0,
empty = N;

produce (item)
{
    P(&empty);
    P(&mutex);
    queue.enqueue(item);
    V(&mutex);
    V(&full);
}

consume()
{
    P(&full);
    P(&mutex);
    item = queue.dequeue();
    V(&mutex);
    V(&empty);
    return item;
}
```

More Synchronization Primitives

- RCU (Read-Copy-Update)
  - Optimized for frequent read, infrequent write
  - Read is zero cost
  - Write can be costly
  - Heavily used in Linux kernel
- Transactional memory (Intel TSX instruction)
  - Opportunistic synchronization
  - Declare a set of instructions as a transaction
  - If no other CPUs update the shared data while executing the transaction, it is committed
  - If not (i.e., someone tries to modify in the middle of the transaction), the transaction will be aborted
  - If successful, there's no synchronization overhead

Summary

- Synchronization
  - Spinlock
    - Implement using h/w instructions (e.g., test-and-set)
  - Mutex
    - Sleep instead of spinning.
  - Semaphore
    - powerful tool, but often difficult to use
  - **Monitor**
    - Powerful and (relatively) easy to use

Team Project

- At company, you will likely to work in a team. So, better to practice now
- Ideal team size = ??
  - Two is the minimum and a very good one
  - Communication, communication, communication
- A couple of tips
  - Work division
  - Coding standard
  - Version control software

KU THE UNIVERSITY OF KANSAS

Work Division

- Function based
  - Person 1: built-in commands (cd,…)
  - Person 2: pipe and I/O redirection,..
- Job based
  - Person 1: coder
  - Person 2: tester

Coding Standard

- Function naming

```
PrintData()
print_data()
```

- indentation, commenting, …

```
/**
  * commenting style 1
  */
If (condition) {
    // 4 spaces  or
        // 8 spaces
}
```

Version Control Software

- Git
  - Most popular these days.
  - Relatively high learning curve
  - https://git.eecs.ku.edu/ or github

- SVN
  - Still very popular
  - Relatively simple to learn
  - http://wiki.eecs.ku.edu/subversion

KU THE UNIVERSITY OF KANSAS

Agenda

- Famous Synchronization Bugs
  - THERAC-25
  - Mars Pathfinder

Therac 25

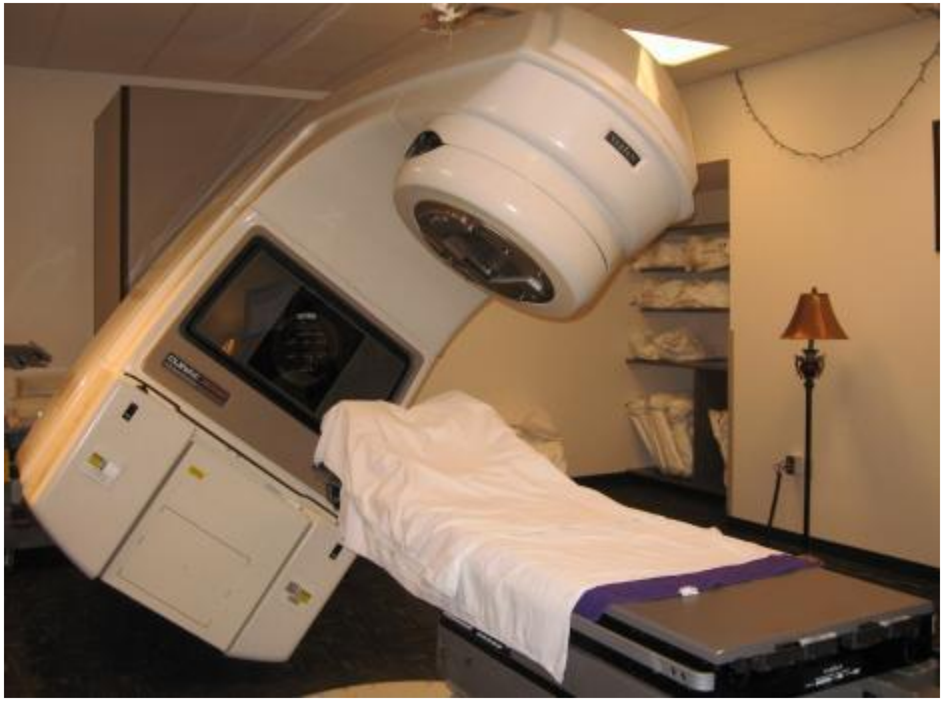


Image source: http://idg.bg/test/cwd/2008/7/14/21367-radiation_therapy.JPG

- Computer controlled medical X-ray treatments
- Six people died/injured due to massive overdoses (1985-1987)

Accident History

Date	What happened
June 1985	First overdose
July-Dec 1985	2nd and 3rd overdoses. Lawsuit against the manufacturer and hospital
Jan-Feb 1986	Manufacturer denied the possibility of overdoses
Mar-Apr 1986	Two more overdoses
May-Dec 1986	FDA orders corrective action plans to the manufacturer
Jan 1987	Sixth overdose
Nov 1988	Final safety analysis report

KU THE UNIVERSITY OF KANSAS

The Problem

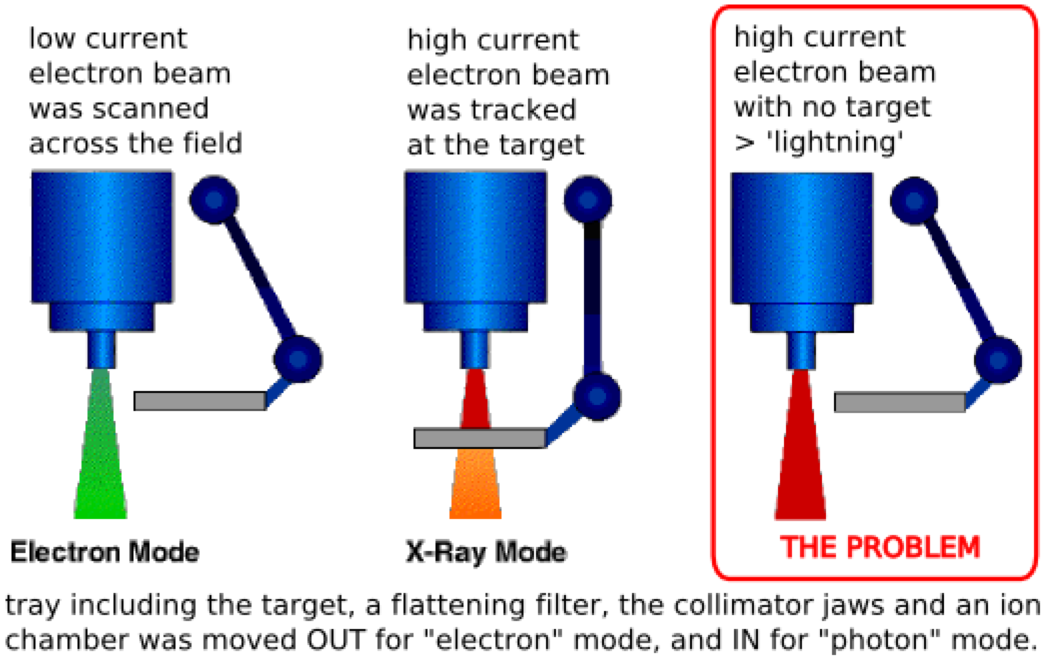
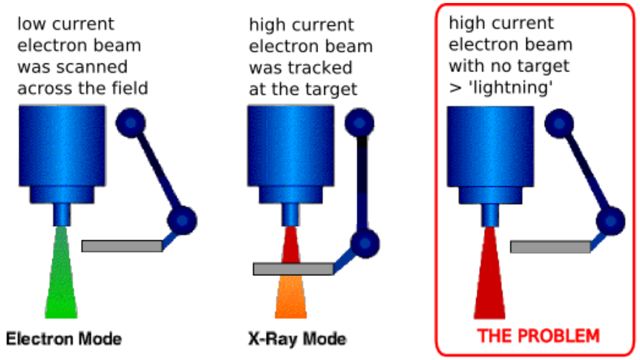


Image source: http://radonc.wdfiles.com/local--files/radiation-accident-therac25/Therac25.png

- X-ray must be dosed with the filter in place
- But sometimes, X-ray was dosed w/o the filter

The Bug

```
unsigned char in_progress = 1;

Thread 1 : // tray movement thread (periodic)
   if (system_is_ready())
       in_progress = 0;
   else
       in_progress++;


Thread 2 : // X-ray control thread.
   if (in_progress == 0)
       start_radiation();
```

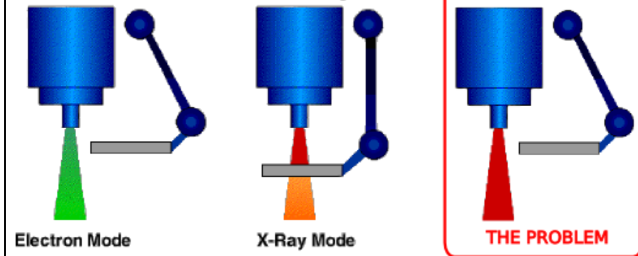


tray including the target, a flattening filter, the collimator jaws and an ion chamber was moved OUT for "electron" mode, and IN for "photon" mode.

- Can you spot the bug?

Fixed Code

```
unsigned char in_progress;

Thread 1 : // tray movement thread (periodic)
   if (system_is_ready())
       in_progress = 0;
   else
       in_progress = 1;


Thread 2 : // X-ray control thread.
   if (in_progress == 0)
       start_radiation();
```



tray including the target, a flattening filter, the collimator jaws and an ion chamber was moved OUT for "electron" mode, and IN for "photon" mode.

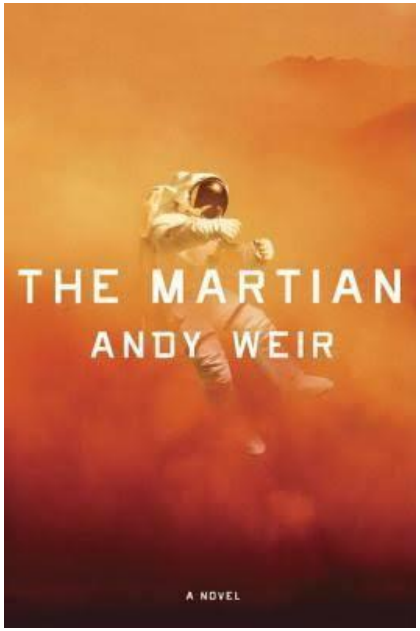
- Can you do better using a monitor?

Monitor Version

```
Mutex lock;
Condition ready;
unsigned char in_progress;

Thread 1 : // on finishing tray movement
   lock.acquire();
   in_progress = 0;
   ready.signal();
   lock.release();

Thread 2 : // X-ray control thread.
   lock.acquire();
   while (in_progress)
       ready.wait(&lock);
   start_radiation();
   lock.release();
```

- No periodic check is needed.

Mars Pathfinder

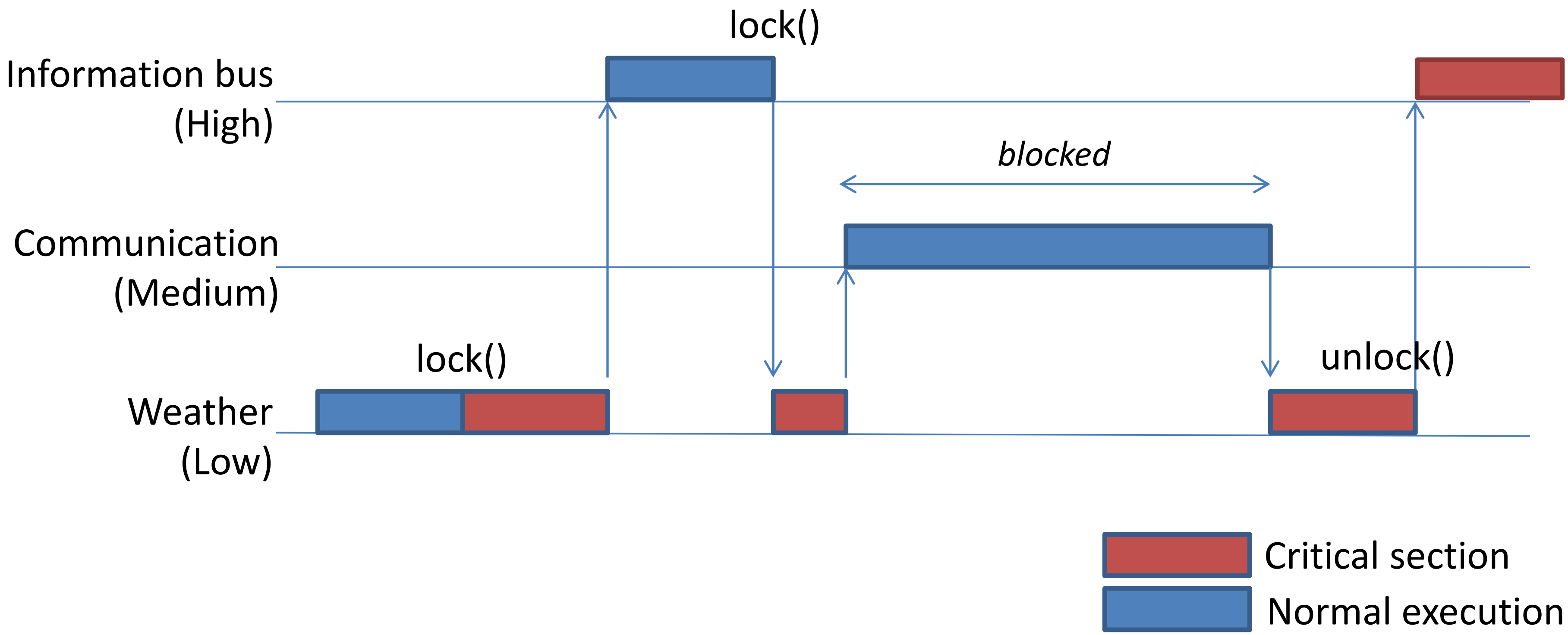


- Landed on Mars, July 4, 1997
- After operating for a while, it rebooted itself

The Bug

- Three threads with priorities
  - Weather data thread (low priority)
  - Communication thread (medium priority)
  - Information bus thread (high priority)
- Each thread obtains a lock to write data on the shared memory
- High priority thread can’t acquire the lock for a very long time → something must be wrong. Let’s reboot!

Priority Inversion



- High priority thread is delayed by the medium priority thread (potentially) indefinitely!!!

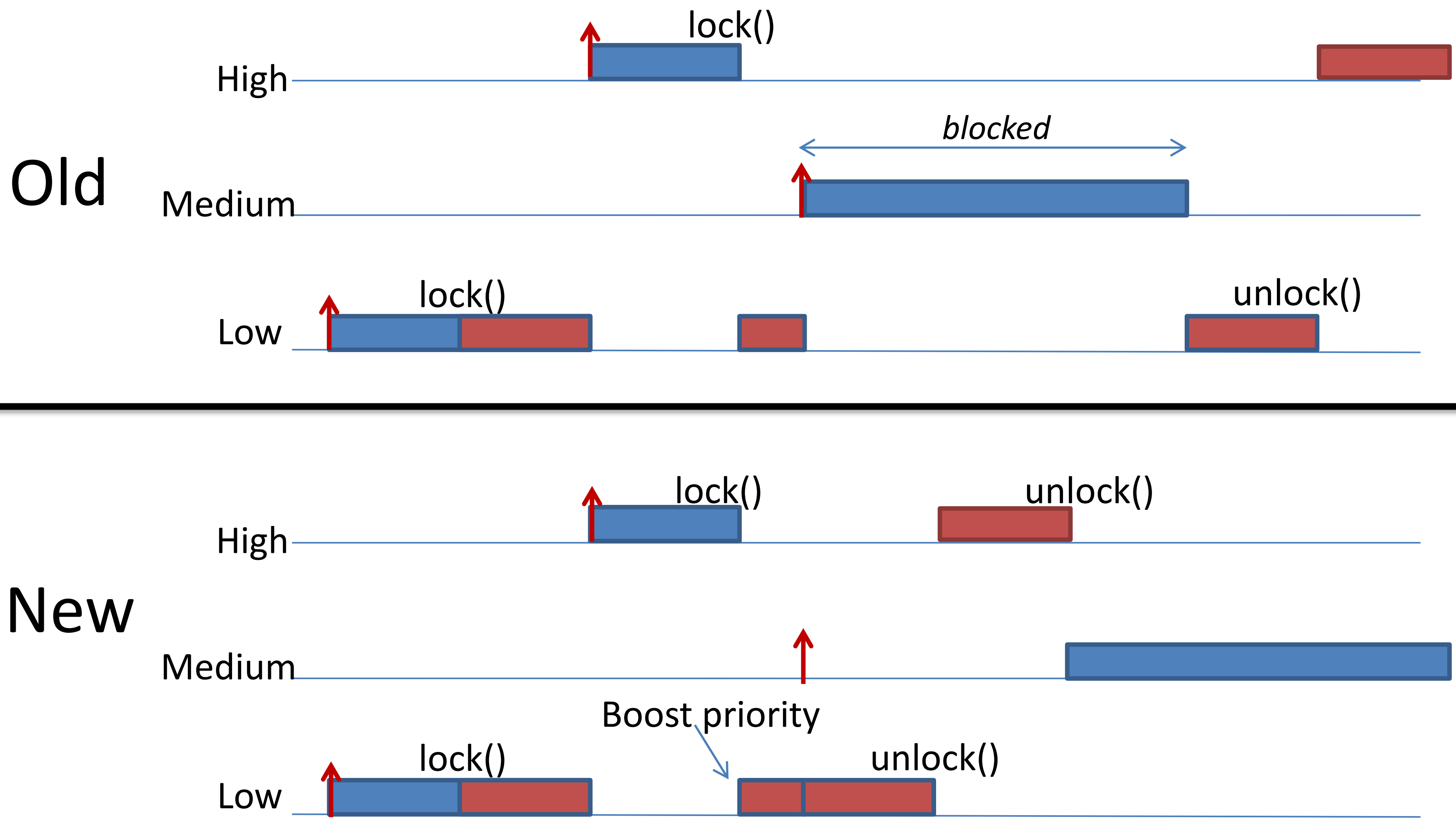
More reading: [What really happened on Mars?]()

Solution

- Priority inheritance protocol [Sha'90]
  - If a high priority thread is waiting on a lock, boost the priority of the lock owner thread (low priority) to that of the high priority thread.
- Remotely patched the code
  - To use the priority inheritance protocol in the lock
  - First-ever(?) interplanetary remote debugging

L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In IEEE Transactions on Computers, vol. 39, pp. 1175-1185, Sep. 1990.

Priority Inheritance



Summary

- Race condition
  - A situation when two or more threads **read and write** shared data at the same time
- Critical section
  - Code sections of potential race conditions
- Mutual exclusion
  - If a thread executes its critical section, *no other threads* can enter their critical sections
- Peterson's solution
  - Software only solution providing mutual exclusion

KU THE UNIVERSITY OF KANSAS

Summary

- Spinlock
  - Spin on waiting
  - Use synchronization instructions (test&set)
- Mutex
  - Sleep on waiting
- Semaphore
  - Powerful tool, but often difficult to use
- Monitor
  - Powerful and (relatively) easy to use

Quiz

```
Mutex lock;
Condition full, empty;

produce (item)
{
    ______________
    while (queue.isFull())
        empty.wait(&lock);
    queue.enqueue(item);
    full.signal();
    ______________
}

consume()
{
    ______________
    while (queue.isEmpty())
        ______________
    item = queue.dequeue(item);
    ______________
    ______________
    return item;
}
```

Quiz

```
Mutex lock;
Condition full, empty;

produce (item)
{
   lock.acquire();
   while (queue.isFull())
      empty.wait(&lock);
   queue.enqueue(item);
   full.signal();
   lock.release();
}

consume()
{
   lock.acquire();
   while (queue.isEmpty())
      full.wait(&lock);
   item = queue.dequeue(item);
   empty.signal();
   lock.release();
   return item;
}
```