Synchronization

Disclaimer: some slides are adopted from the book authors' slides

Recap: *TestAndSet* Instruction

- Pseudo code

```
boolean TestAndSet (boolean *target)
{
      boolean rv = *target;
      *target = TRUE;
      return rv:
}
```

Recap: **Spin**lock using *TestAndSet*

```
int mutex;
init_lock (&mutex);

do {

    lock (&mutex);
        critical section
    unlock (&mutex);

        remainder section
} while(TRUE);
```

```
void init_lock (int *mutex)
{
    *mutex = 0;
}

void lock (int *mutex)
{
    while(TestAndSet(mutex))
        ;
}

void unlock (int *mutex)
{
    *mutex = 0;
}
```

Recap

- Synchronization instructions
  - *test&set, compare&swap*
  - All or nothing
- Spinlock
  - Spin on wait
  - Good for short critical section but can be wasteful

KU THE UNIVERSITY OF KANSAS

More reading: mutex.c in Linux

```
void mutex_init (mutex_t *lock)
{
   lock->value = 0;
...
...
}

void  mutex_lock (mutex_t *lock)
{
...
   while(TestAndSet(&lock->value)) {
...
...
...
...
...
   }
...
}

void mutex_unlock (mutex_t *lock)
{
...
   lock->value = 0;
...
...
...
}
```

More reading: [mutex.c in Linux]

```
void mutex_init (mutex_t *lock)
{
   lock->value = 0;
   list_init(&lock->wait_list);
   spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
   spin_lock(&lock->wait_lock);
   while(TestAndSet(&lock->value)) {
      current->state = WAITING;
      list_add(&lock->wait_list, current);
      spin_unlock(&lock->wait_lock);
      schedule();
      spin_lock(&lock->wait_lock);
   }
  spin_unlock(&lock->wait_lock);
}

void mutex_unlock (mutex_t *lock)
{
   spin_lock(&lock->wait_lock);
   lock->value = 0;
   if (!list_empty(&lock->wait_list))
      wake_up_process(&lock->wait_list)
   spin_unlock(&lock->wait_lock);
}
```

Thread waiting list

To protect waiting list

Thread state change

Add the current thread to the waiting list

Sleep or schedule another thread

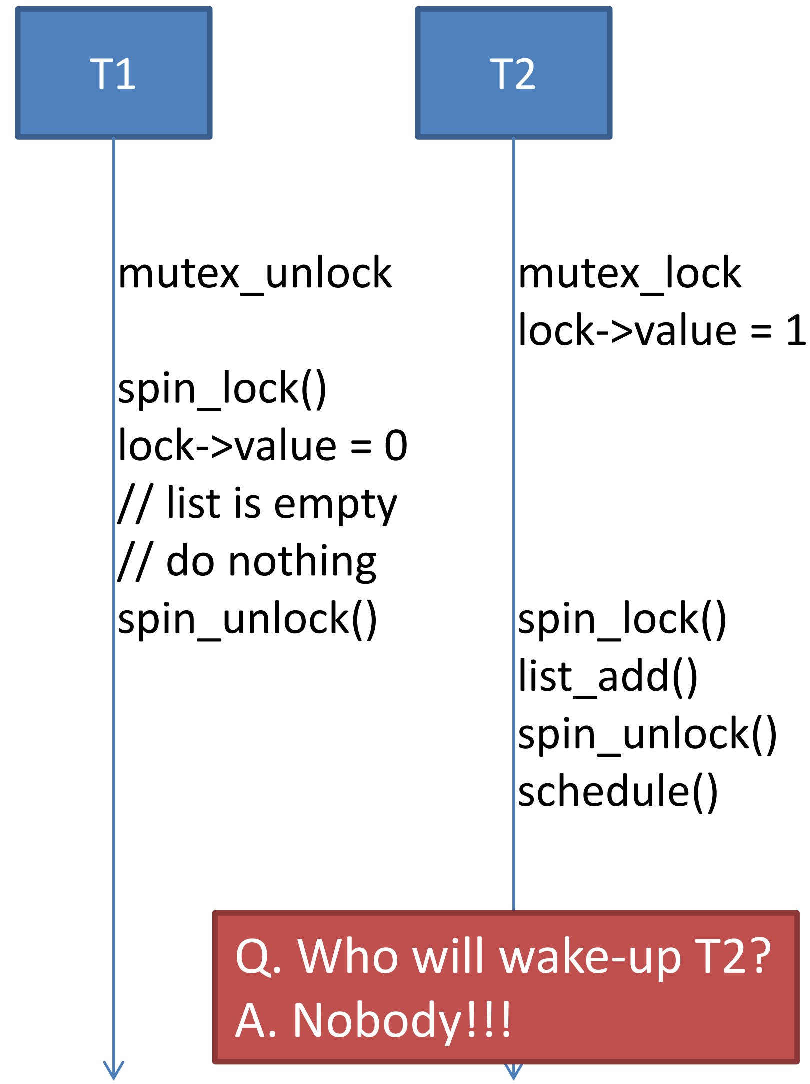
Someone is waiting for the lock

Wake-up a waiting thread

More reading: [mutex.c in Linux]

```
void mutex_init (mutex_t *lock)
{
    lock->value = 0;
    list_init(&lock->wait_list);
    spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
    while(TestAndSet(&lock->value)) {
        current->state = WAITING;
        spin_lock(&lock->wait_lock);
        list_add(&lock->wait_list, current);
        spin_unlock(&lock->wait_lock);
        schedule();
    }
}

void mutex_unlock (mutex_t *lock)
{
    spin_lock(&lock->wait_lock);
    lock->value = 0;
    if (!list_empty(&lock->wait_list))
        wake_up_process(&lock->wait_list)
    spin_unlock(&lock->wait_lock);
}
```

Correct?



Agenda

- High-level synchronization mechanisms
  - Mutex 互斥锁
  - **Semaphore** 信号量
  - Monitor

Semaphore

- High-level synchronization primitive
  - Designed by Dijkstra in 1960’
- Definition
  - Semaphore is an integer variable
  - Only two operations are possible:
    - P() or wait() or down()
    - V() or signal() or up()

P(sv)：如果sv的值大于零，就给它减1；如果它的值为零，就挂起该进程的执行；运行 P（wait()），信号量s的值将被减少。企图进入临界区块的进程，需要先运行 P（wait()）。当信号量s减为负值时，进程会被挡住，不能继续；当信号量s不为负值时，进程可以获准进入临界区块。
V(sv)：如果有其他进程因等待sv而被挂起，就让它恢复运行，如果没有进程因等待sv而挂起，就给它加1.

Simple Semaphore Implementation

- P() operation

```
P(semaphore *S) {
        S–>value— ;
        If (S–>value < 0) {
                S->list->addQ(P);
                schedule( );
        }
}
```

执行，当<0 被挂起，等待前面一个完成

schedule ( ) – schedule another thread

- V() operation

```
V(semaphore *S) {
        S–>value++ ;
        if(S-> value <= 0) {
                P = delQ(&S->list);
                wakeup(P);
        }
}
```

wakeup( ) – wake up a thread

举个例子，就是 两个进程共享信号量sv，一旦其中一个进程执行了P(sv)操作，它将得到信号量，并可以进入临界区，使sv减1。而第二个进程将被阻止进入临界区，因为 当它试图执行P(sv)时，sv为0，它会被挂起以等待第一个进程离开临界区域并执行V(sv)释放信号量，这时第二个进程就可以恢复执行

https://www.cnblogs.com/fangshenghui/p/4039946.html

What’s wrong with the code?

Simple Semaphore Implementation

- P() operation

```
P(semaphore *S) {
    S->lock->Acquire();
    S–>value— ;
    if(S–>value < 0) {
        addQ(&S->list, P);
        S->lock->Release();
        schedule( );
        S->lock->Acquire();
    }
    S->lock->Release();
}
```

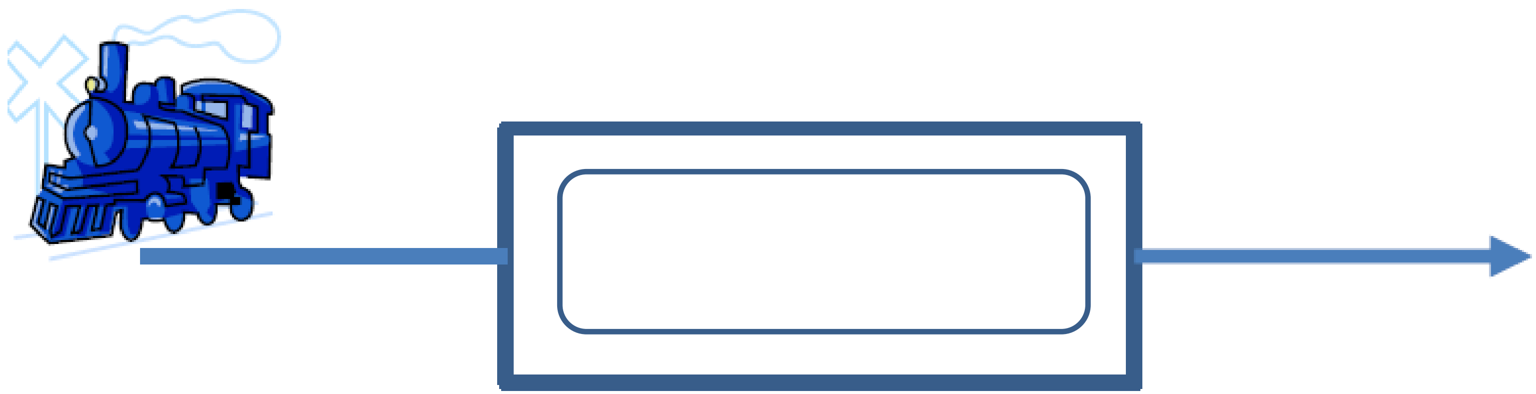
- V() operation

```
V(semaphore *S) {
    S->lock->Acquire();
    S–>value++ ;
    if(S->value <= 0) {
        P = delQ(&S->list);
        wakeup(P);
    }
    S->lock->Release();
}
```

Train Control Problem

- No more than two trains can enter the rail

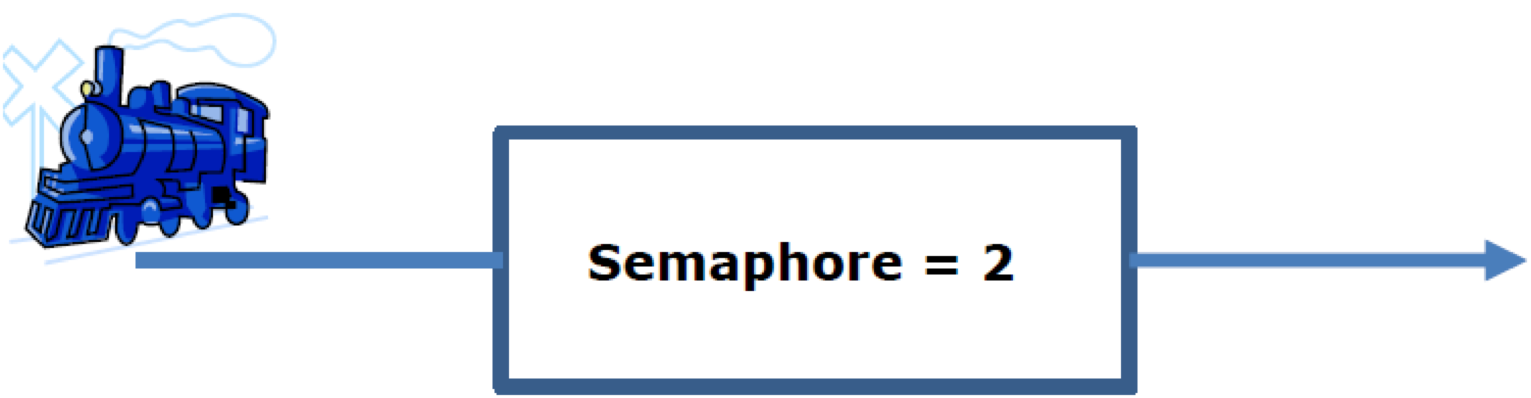
```
Enter() {      }
Leave() {      }
```



Train Control Problem

- No more than two trains can enter the rail

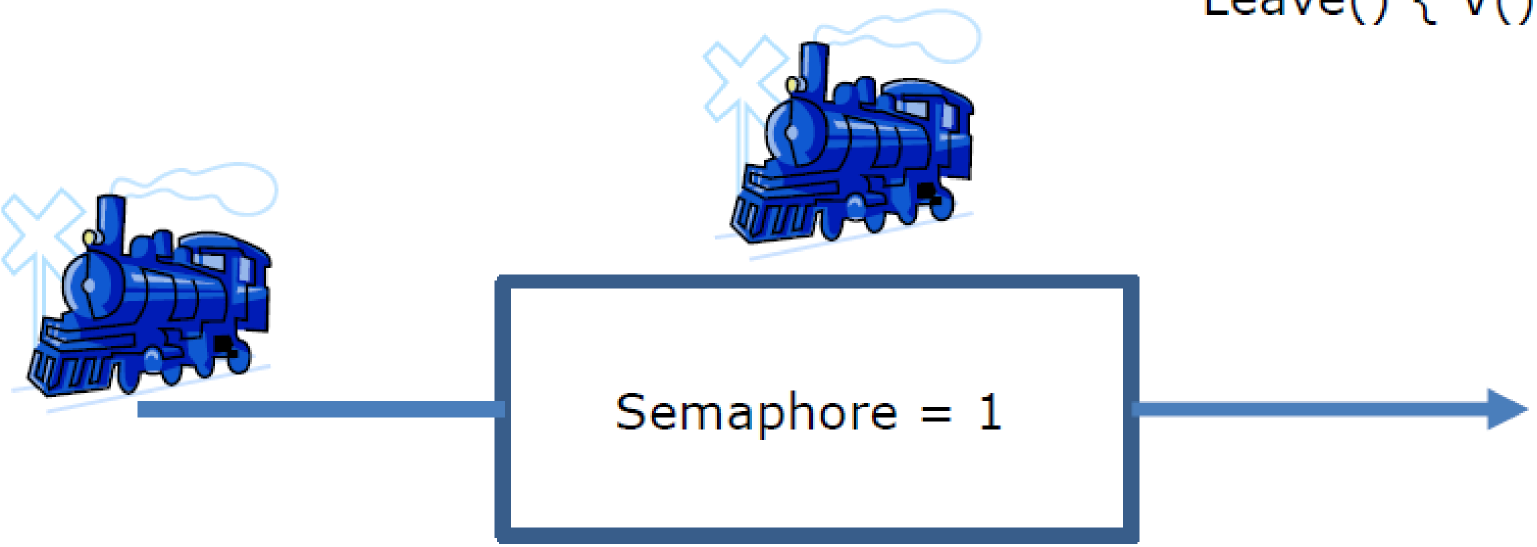
```
Enter() { P(); }
Leave() { V(); }
```



Train Control Problem

- No more than two trains can enter the rail

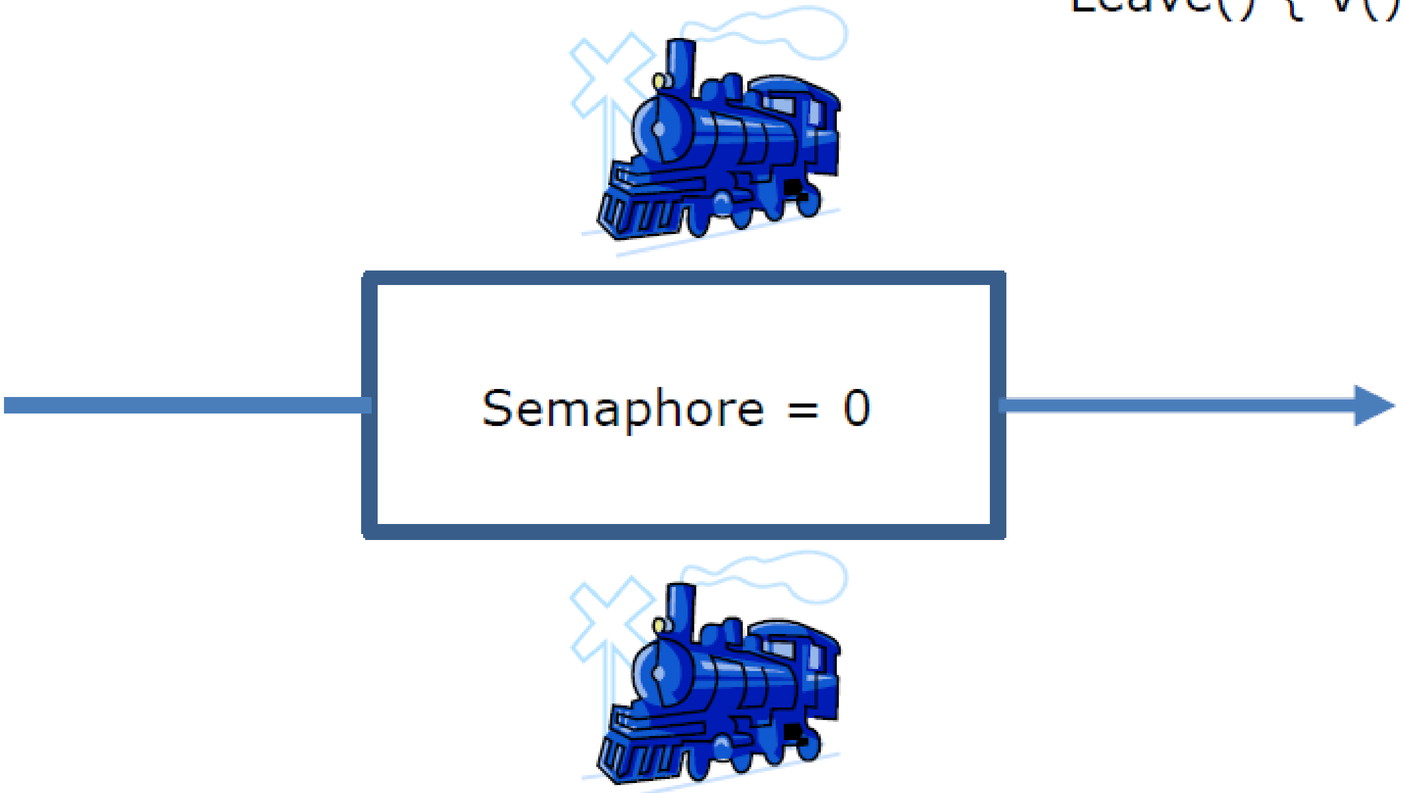
```
Enter() { P(); }
Leave() { V(); }
```



Train Control Problem

- No more than two trains can enter the rail

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



Bounded Buffer Problem

- Problem synopsis
  - A buffer is shared between a producer and a consumer
    - The size of the buffer is N.
  - Producer inserts resources into the buffer
    - If the buffer is **not full**; (owise wait)
  - Consumer removes resources from the buffer
    - If the buffer is **not empty**; (owise wait)
  - producer and consumer execute **asynchronously**
    - CPU scheduler determines what run when

Bounded Buffer Problem

- We use *two integer semaphores* and *one binary semaphore*
- *mutex* 互斥操作
  - Mutual exclusion for accessing the buffer
- *empty* ==0 代表空的
  - Available space, initialized as N
  - If zero, the producer has to wait
- *full* 是否有东西，==0 代表里面没有数据
  - #of data in the buffer; initialized as 0
  - If zero, the consumer has to wait

对于生产者：

判断缓冲区是否为满，如果为满，则等待；否则，允许一个生产者写入。

对于消费者：

判断缓冲区是否为空，如果为空，则等待；否则，允许一个消费者读取。

Bounded Buffer Problem

- Semaphore bool mutex;
- Semaphore int **full = 0;**
- Semaphore int **empty = N;**

Producer

信号量的加锁，每次只能放入指定数量的东西，否则等待

加锁和解锁，每一次要么进行producer要么进行consumer

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

Consumer

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

Reader-Writer Problem

- Problem synopsis
  - An object is shared among several threads
  - Some threads only read the object (Readers)
  - Some threads only write the object (Writers)

- Correctness constraints
  - **Multiple readers** can access the shared resource simultaneously
  - But only **one writer** can update the object, when there is no other reader or writer
  - Readers can't access the object when a writer updates it

Reader-Writer Problem

- Problem synopsis
  - An object is shared among several threads
  - Some threads only read the object (Readers)
  - Some threads only write the object (Writers)

- Correctness constraints
  - **Multiple readers** can access the shared resource simultaneously
  - But only **one writer** can update the object, when there is no other reader or writer
  - Readers can't access the object when a writer updates it

Recap: Semaphore

- High-level synchronization primitive
  - Designed by Dijkstra in 1960'
- Definition
  - Semaphore is an integer variable
  - Only two operations are possible:
    - P() or wait() or down()
    - V() or signal() or up()

Reader-Writer Problem

- A solution using two binary semaphores
  - mutex
    - ensure mutual exclusion for the **readcount** variable
    - mutex semaphore, initialized to 1
  - wrt
    - ensure **mutual exclusion for writers**
    - ensure **mutual exclusion between readers and writer**
    - mutex semaphore, initialized to 1

Reader-Writer Problem

```
semaphore mutex = 1, wrt = 1;
int readcount = 0;
```

Writer

```
do {
    P(wrt);
    write object resource
    V(wrt);
} while (TRUE);
```

Reader-Writer Problem

```
semaphore mutex = 1, wrt = 1;
int readcount = 0;
```

Writer

```
do {
    P(wrt);
    write object resource
    V(wrt);
} while (TRUE);
```

readcount是共享资源

Reader

```
do {

    readcount++;
    if (readcount == 1)
        P(wrt);

    read from object resource

    readcount--;
    if (readcount == 0)
        V(wrt);

} while (TRUE);
```

What's wrong with this code?

Reader-Writer Problem

```
semaphore mutex = 1, wrt = 1;
int readcount = 0;
```

Writer

```
do {
    P(wrt);
    write object resource
    V(wrt);
} while (TRUE);
```

Reader

```
do {
    P(mutex);
    readcount++;
    if (readcount == 1)
        P(wrt);
    V(mutex)
    read from object resource
    P(mutex);
    readcount--;
    if (readcount == 0)
        V(wrt);
    V(mutex)
} while (TRUE);
```

Semaphore Review

- Semaphores can solve many synchronization problems
  - A “huge” step up from locks
- Drawbacks of semaphores
  - Semaphores are still low-level primitives
  - Used for both mutual exclusion and scheduling
  - Very easy to screw up

Agenda

- High-level synchronization mechanisms
  - Mutex
  - Semaphore
  - **Monitor**

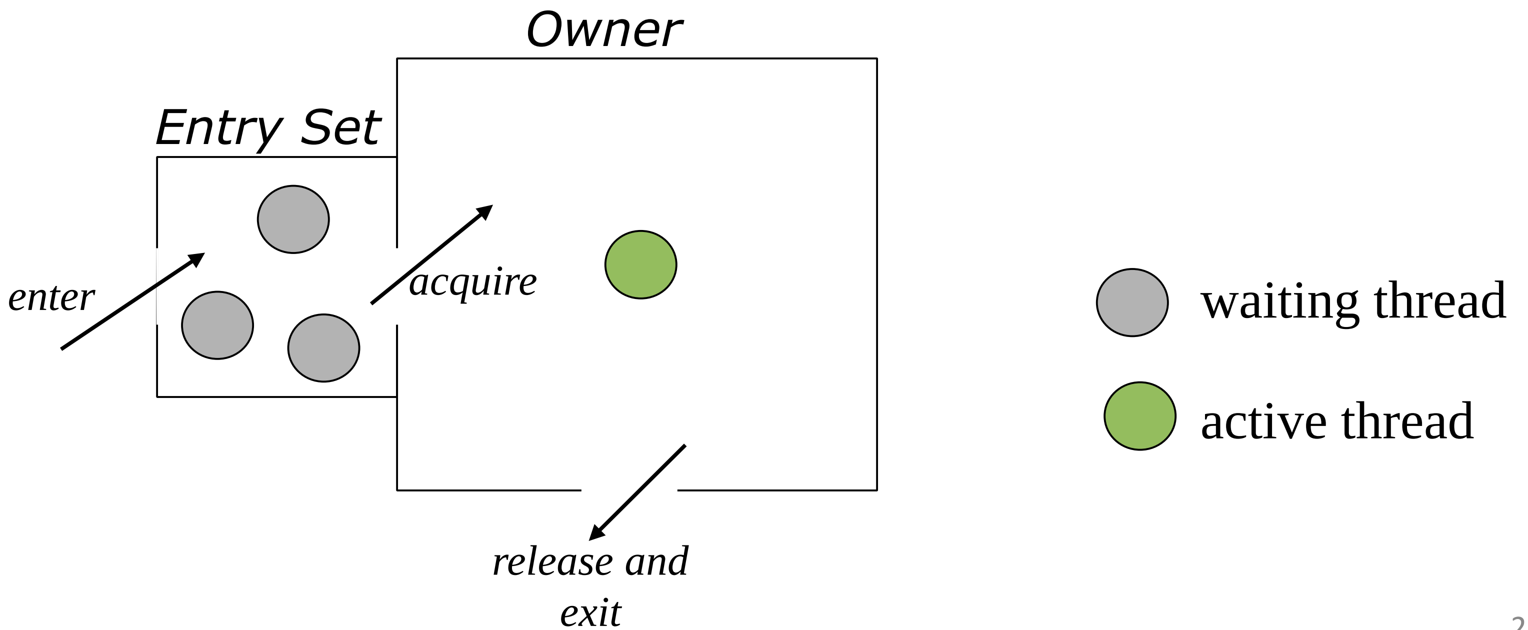
Monitor

管程 或者 监视器cheng

- 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

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++

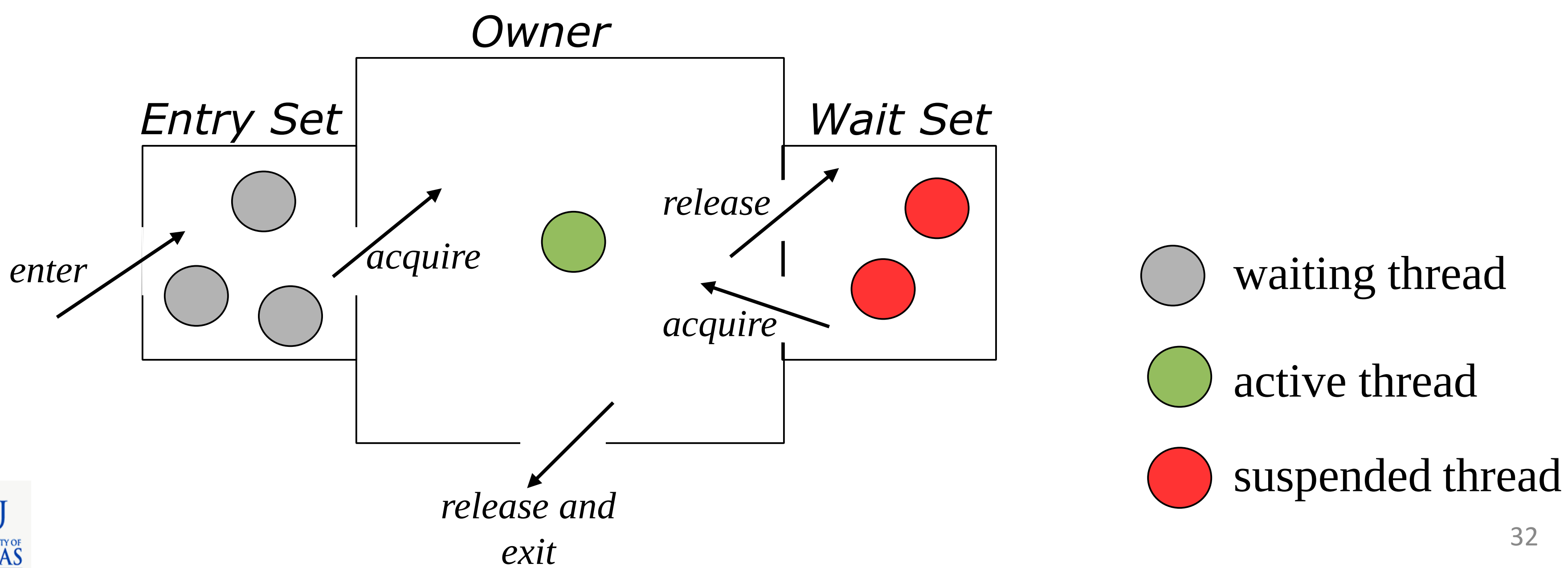
```
Mutex lock;
Queue queue;

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’?

idk，可能就是想一直check lock的状态

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

Agenda

- Famous Synchronization Bugs
  - THERAC-25
  - Mars Pathfinder

Therac 25

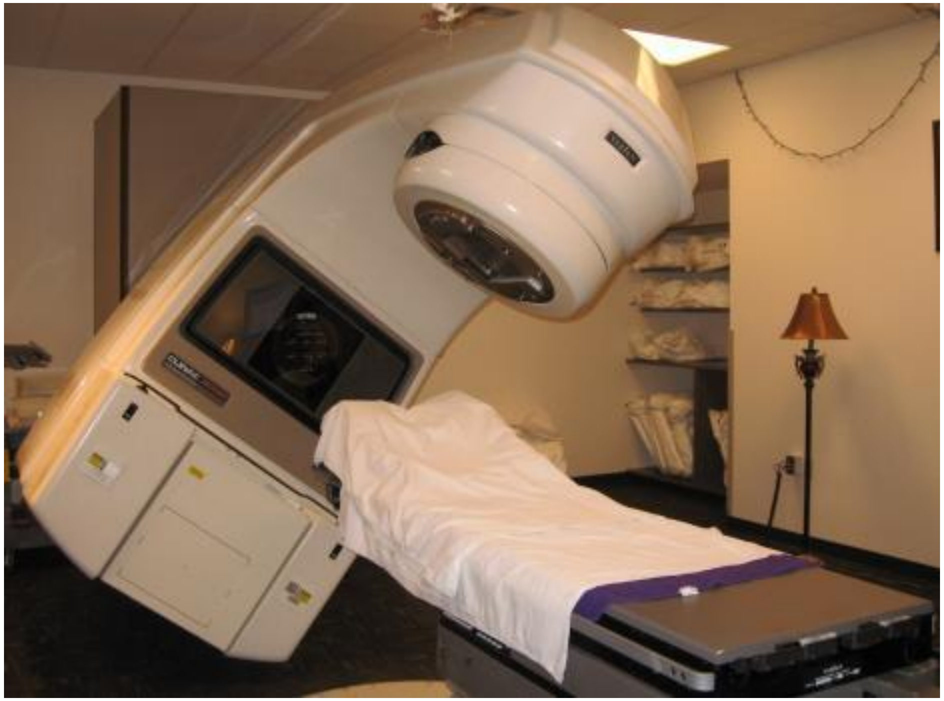


Image source: http://idg.bg/test/cwd/2008/7/14/21367-radiation_therapy.JPG

- 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

The Problem

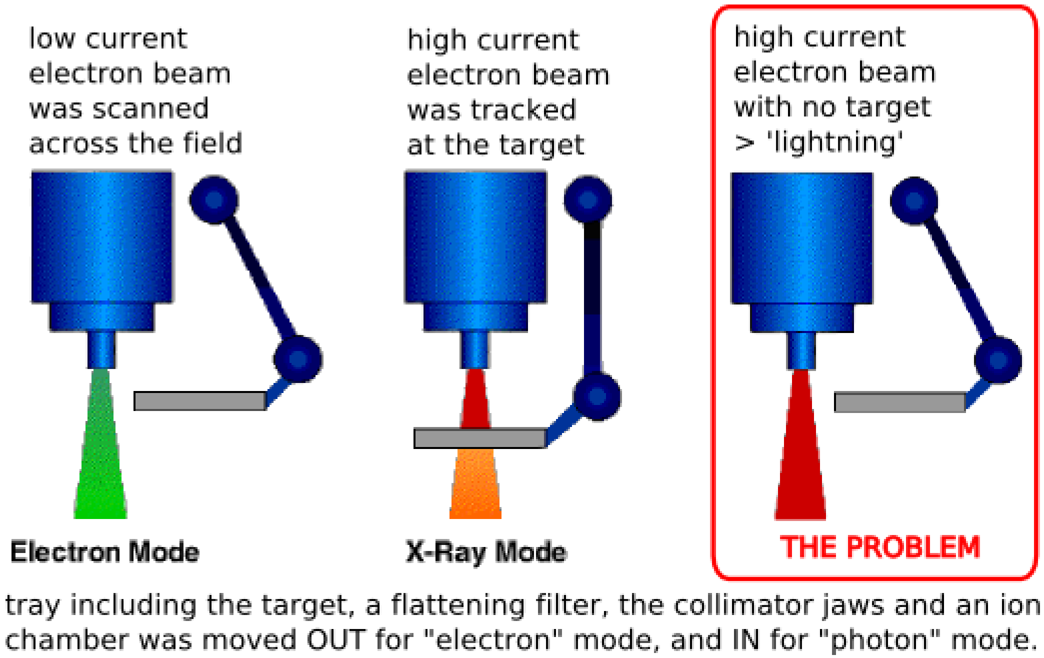
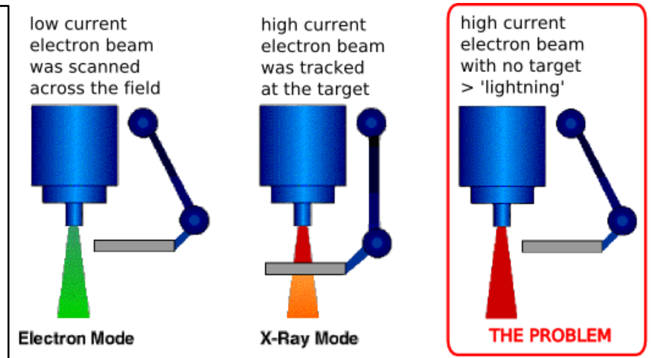


Image source: http://radonc.wdfiles.com/local--files/radiation-accident-therac25/Therac25.png

- 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();
```

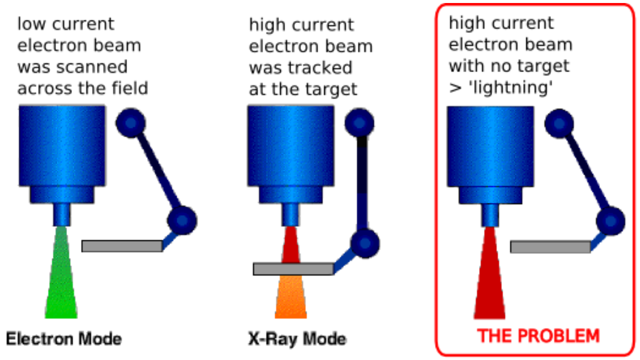


tray including the target, a flattening filter, the collimator jaws and an ion chamber was moved OUT for "electron" mode, and IN for "photon" mode.

- 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?

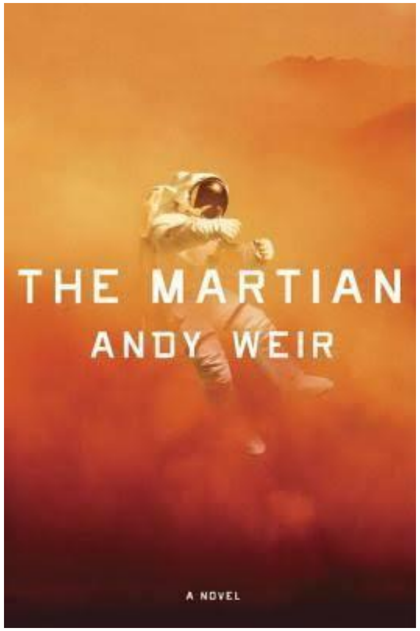
Monitor Version

```
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.

KU THE UNIVERSITY OF KANSAS

Mars Pathfinder

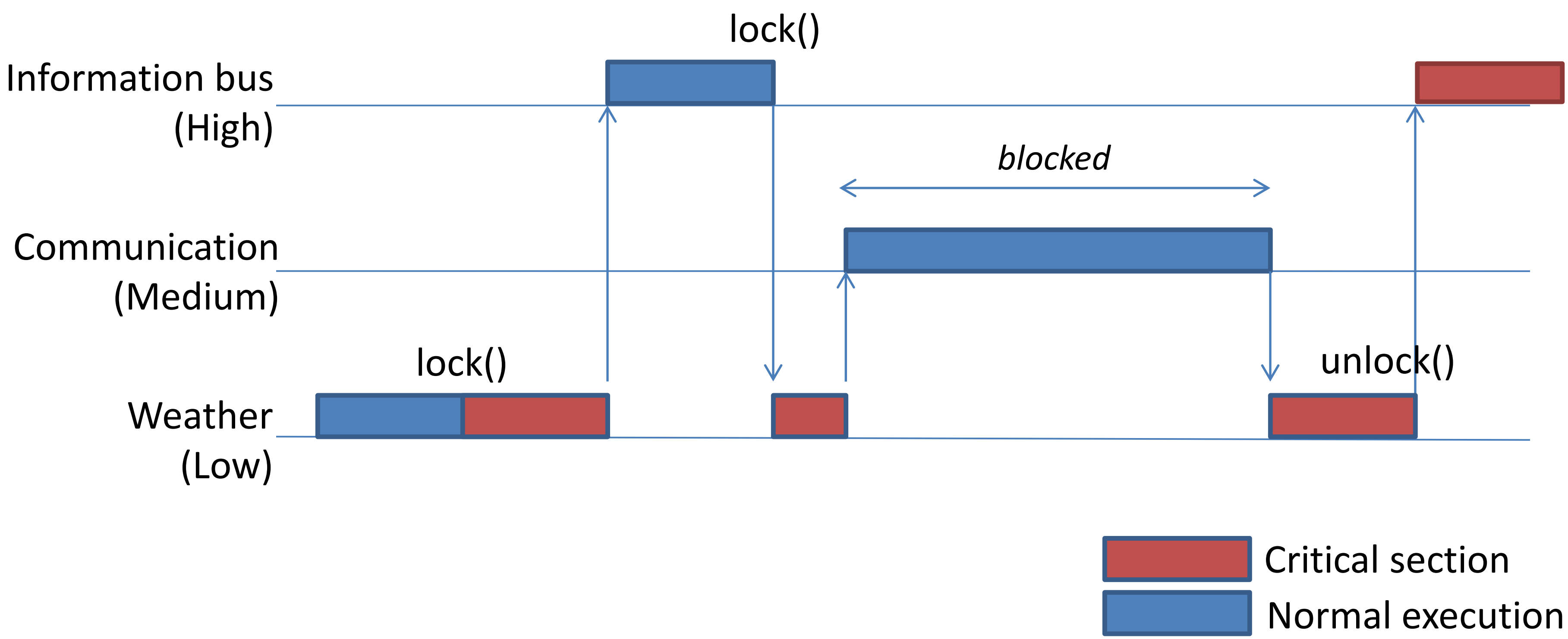


- Landed on Mars, July 4, 1997
- After operating for a while, it rebooted itself

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!!!

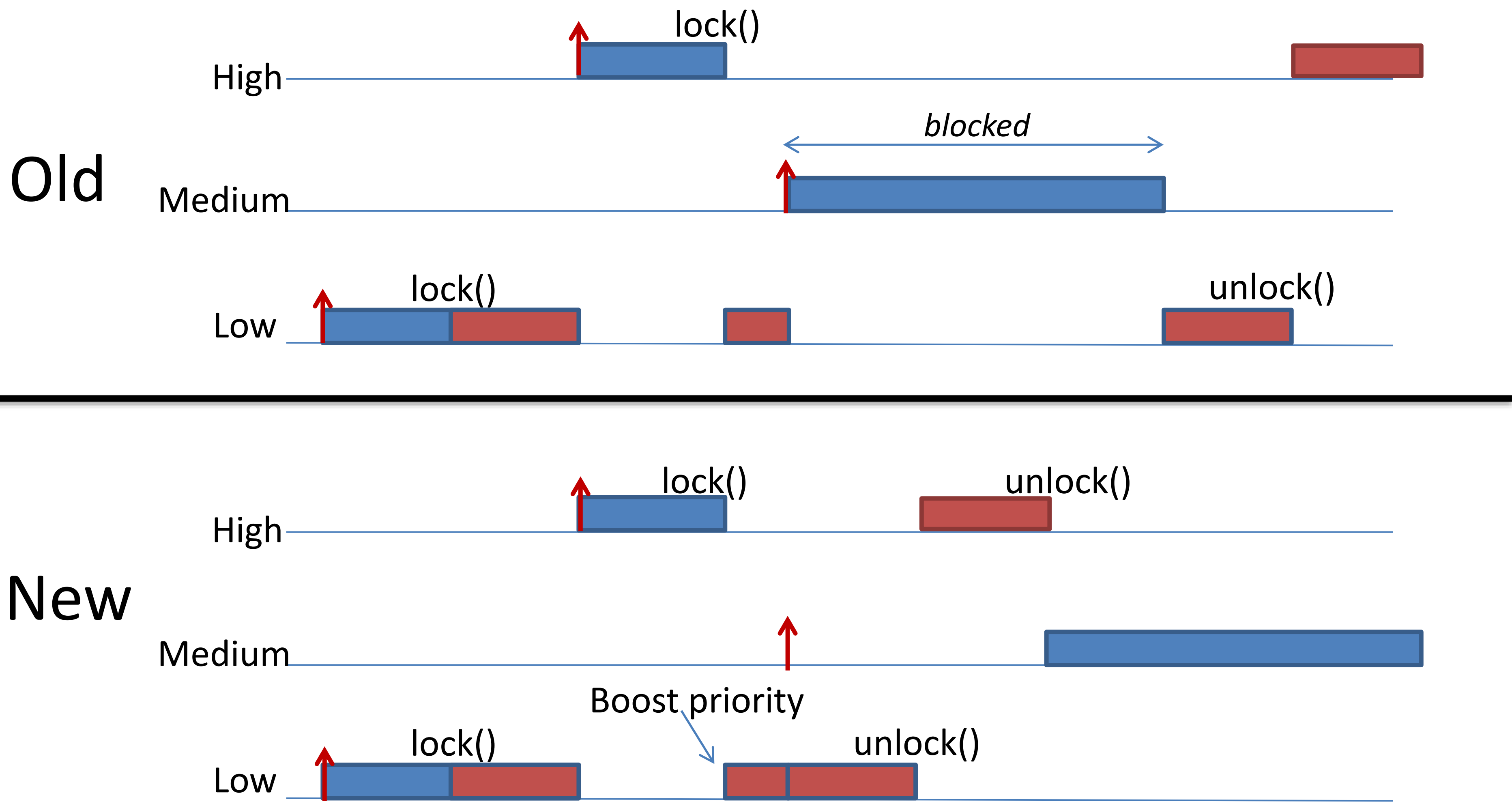
More reading: [What really happened on Mars?]()

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

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

- 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);
```

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;
}
```