CS 475
Operating Systems

Department of Mathematics
and Computer Science

Lecture 6
Synchronization (Part I)

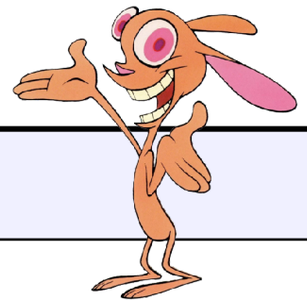
Problem: Race Conditions

- *Race Condition*: When 2 or more threads **read and modify** a **shared resource** without *synchronization*.

- The **correctness of the end result** depends on:
  - Which context switches occurred, if any
  - Which thread ran after the context switch
    - Which thread got switched out
  - What the threads were doing
    - Reading the shared variable? Writing to the shared variable? Neither?
  - *Difficulty: Nondeterministic runtime sequencing; may not be repeatable (race conditions are super hard to debug)*

Too Much Milk Problem

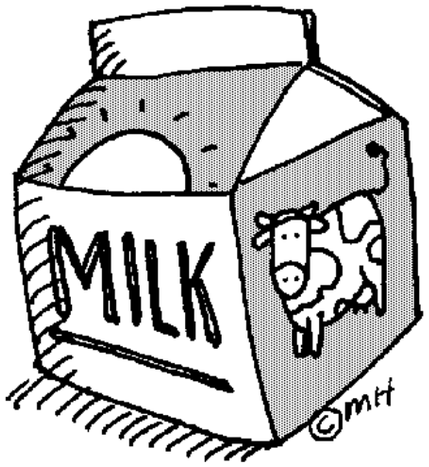
- To understand *race conditions*... let's consider this scenario
  - Ren and Stimpy live together, but don't like to communicate 😒

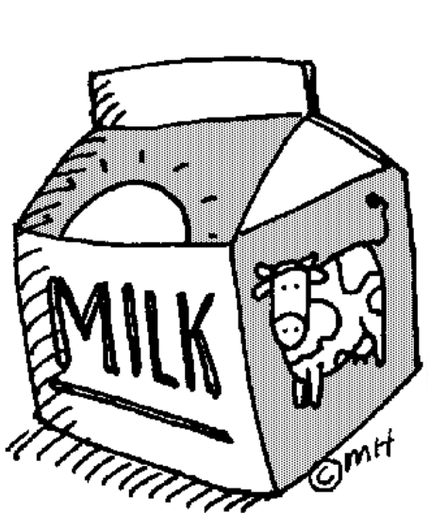


Time	Ren	Stimpy
3:00	Look in fridge	
3:05	*"Out of milk!"*	
3:10	Leave for store	Look in fridge
3:15	Arrive at store	*"Out of milk!"*
3:20	Buy Milk	Leave for store
3:25	Arrive at home, put milk away	Arrive at store
3:30		Buy Milk
3:35		Arrive at home, put milk away

Too Much Milk Problem (2)

Actual picture of fridge from a 2016 OS student





Code Example of a Race Condition

- Assume that `y` is shared between two threads:

  Initially, `y = 12;`

Thread 1

```
x = 1;
x = y + 1;
```

Thread 2

```
y = 5;
y = y * 2;
```

- Let the threads run concurrently. *What is the value of **x** after both threads are done executing?*

Code Example of a Race Condition

- Assume that `y` is shared between two threads:

  Initially, `y = 12;`

Thread 1

```
x = 1;
x = y + 1;
```

Thread 2

```
y = 5;
y = y * 2;
```

- Let the threads run concurrently. *What is the value of **x** after both threads are done executing?*

- *How many different schedules could there be? (In this case, 6 schedules!)*

$$\frac{\left(\sum_i O_i\right)!}{\prod_i O_i!}$$ where O_i is the number of operations in the ith thread

More Fundamentally...

- Assume variable x is shared between two threads.

Thread 1

```
x = 1;
```

Thread 2

```
x = 2;
```

- Can ***x*** be 3 after the two threads run?

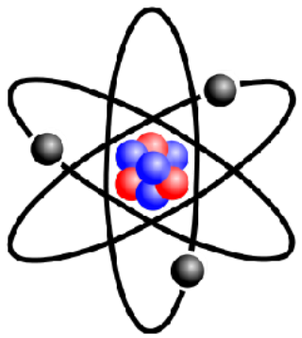
How could it be 3? What if register access is interleave-able?

Can x Be 3? (No - phew!)

- **Definition:** *Atomic Operations*
  - An operating is *atomic* if it cannot be broken up into multiple smaller operations

- In modern CPU architectures, register references (*e.g.*, `loads` and `stores`) are guaranteed to be atomic *(Phew!)*

(But this still doesn't fix nondeterminism caused by race conditions between atomic operations!)



Another Example: What Gets Printed?

add2m.c

```c
#include <pthread.h>
#include <stdio.h>

int x = 0;   // shared global

void* doStuff(void* args) {
    for (int i = 0; i < 1000000; i++) {
        x++;
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, doStuff, NULL);
    pthread_create(&t2, NULL, doStuff, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Value of x is %d\n", x);
    return 0;
}
```

I want to increment **x** to 2,000,000 but split the work across 2 threads to do it in half the time!

Output:

```
$ gcc add2m.c -lpthread -o add2m
$ ./add2m
Value of x is 1029184

$ ./add2m
Value of x is 1224682

$ ./add2m
Value of x is 1004207

$ ./add2m
Value of x is 1003645
```

Reason: The **++** operator is not atomic!

Goals for This Lecture...

- Basic Problem Definition
  - Race Conditions and the Critical Section Problem
- Mechanisms to Control Access to Shared Resources
  - Low Level Mechanisms:
    - Busy-Waiting (Spin) Locks
    - Self-Blocking Locks
  - High Level Mechanisms:
    - Semaphores
    - Condition Variables and Monitors

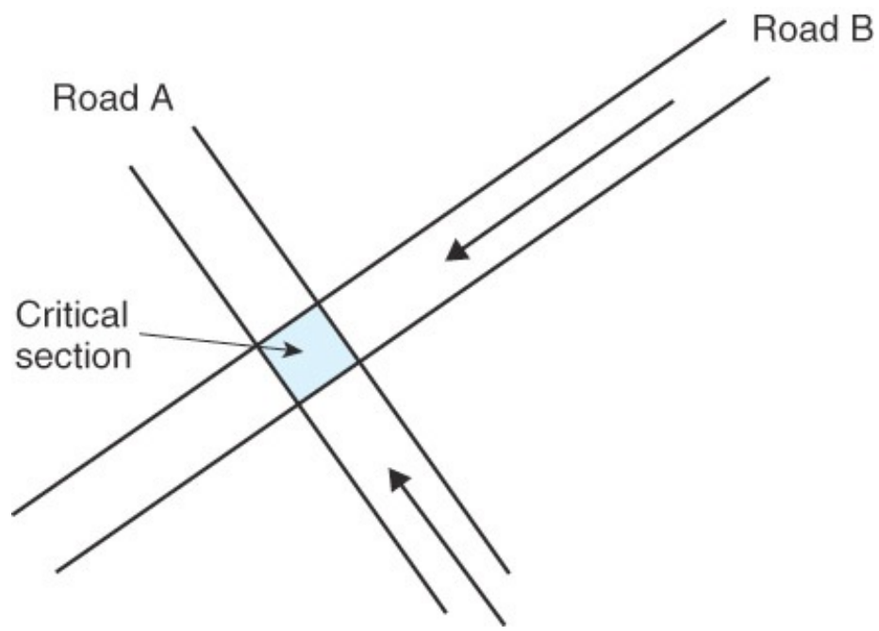
Critical Section Problem

- **Definitions:**
  - *"Critical Section"*
    - A fragment of code where multiple threads read+write shared data.
    - Its execution must be controlled to prevent race conditions.
  - *"Critical Section Problem"*
    - To ensure that multiple threads access shared resources in a way that prevents race conditions while maintaining
      - Mutual exclusion
      - Progress
      - Bounded wait

Critical Section Problem (2)

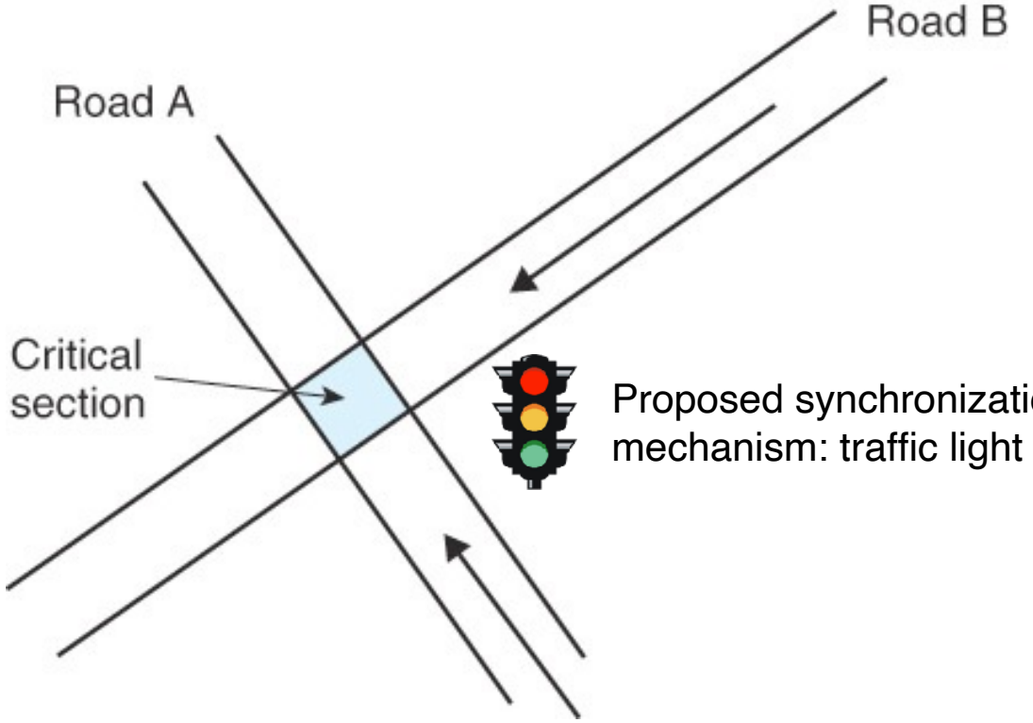
- An analogy: Roads and intersections
  - Roads = Threads
  - Intersections = Shared Resource
    - Its *access* represents the critical section

- How do we go about solving the **intersection** problem in real life?
  - Signals, stop signs, yield signs, etc.
  - All are used to *synchronize* access to the intersection
- 3 desirable features of a CS solution:
  - (Next)



Critical Section Feature: Mutual Exclusion

- Feature 1: Mutual Exclusion (this property is non-negotiable)
  - Only **one** thread is allowed to access the CS at any point in time
  - Implies all other threads wanting access to the critical section must wait



Proposed synchronization mechanism: traffic light

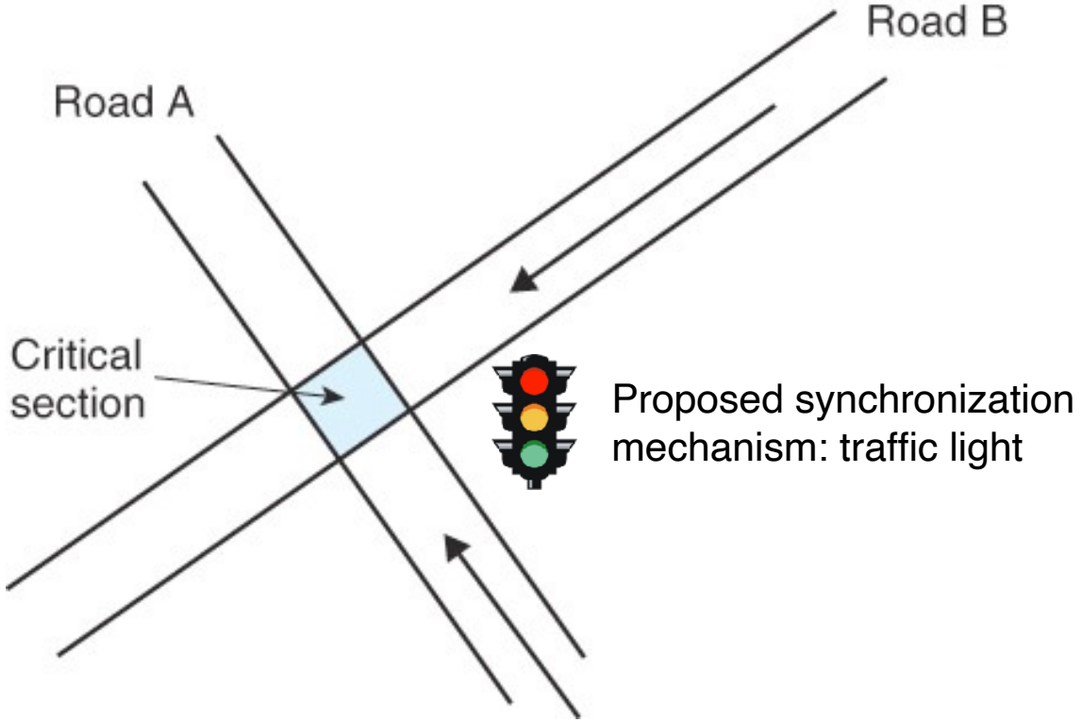
Does the **traffic light** provide mutual exclusion of the intersection?

Yes, it allows traffic on one road to be active in the critical section. All other traffic must wait.

Critical Section Feature: Progress

Feature 2: Progress

- If the CS is available (no other thread is executing it), a requesting thread must be permitted entry without delay.
  - Otherwise, the thread is waiting for no reason, which hurts performace!



Proposed synchronization mechanism: traffic light

Does the **traffic light** provide progress?

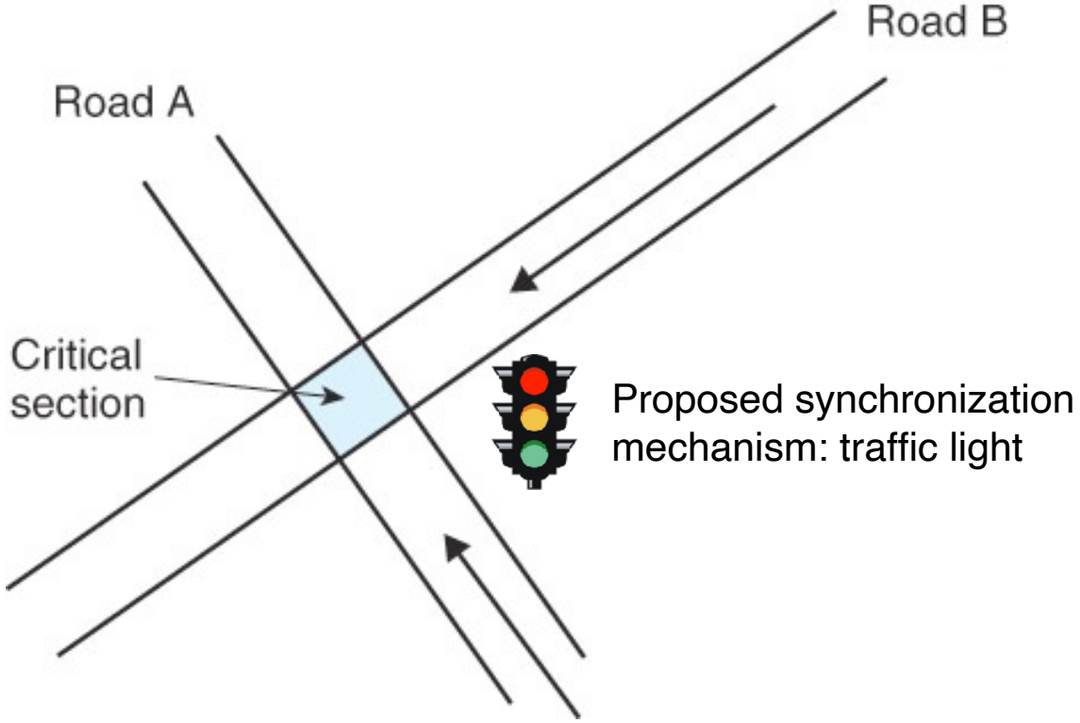
Not really. Sometimes you have to wait at a red light even when there's no traffic on the other road.

Real-world solution? A signal tripper?

Critical Section Feature: Bounded Wait

Feature 3: Bounded Wait

- A requesting thread cannot be delayed indefinitely.



Proposed synchronization mechanism: traffic light

Does the **traffic light** provide bounded wait?

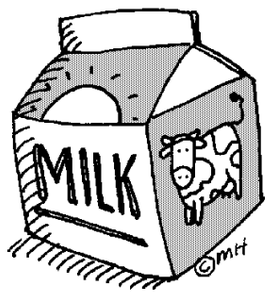
Yes. Cars converging at the intersection will eventually get its turn

Back to the Too Much Milk Problem

- Identify the critical section of code:
  - Actions that we don't want two roommates to be doing at the same time

Time	Ren	Stimpy
3:00	Open fridge	
3:05	*Look for milk -- Out of milk!*	
3:10	Leave for store	Open fridge
3:15	Arrive at store	*Look for milk -- Out of milk!*
3:20	Purchase Milk	Leave for store
3:25	Arrive home, put milk away	Arrive at store
3:30		Purchase Milk
3:35		Arrive home, put milk away

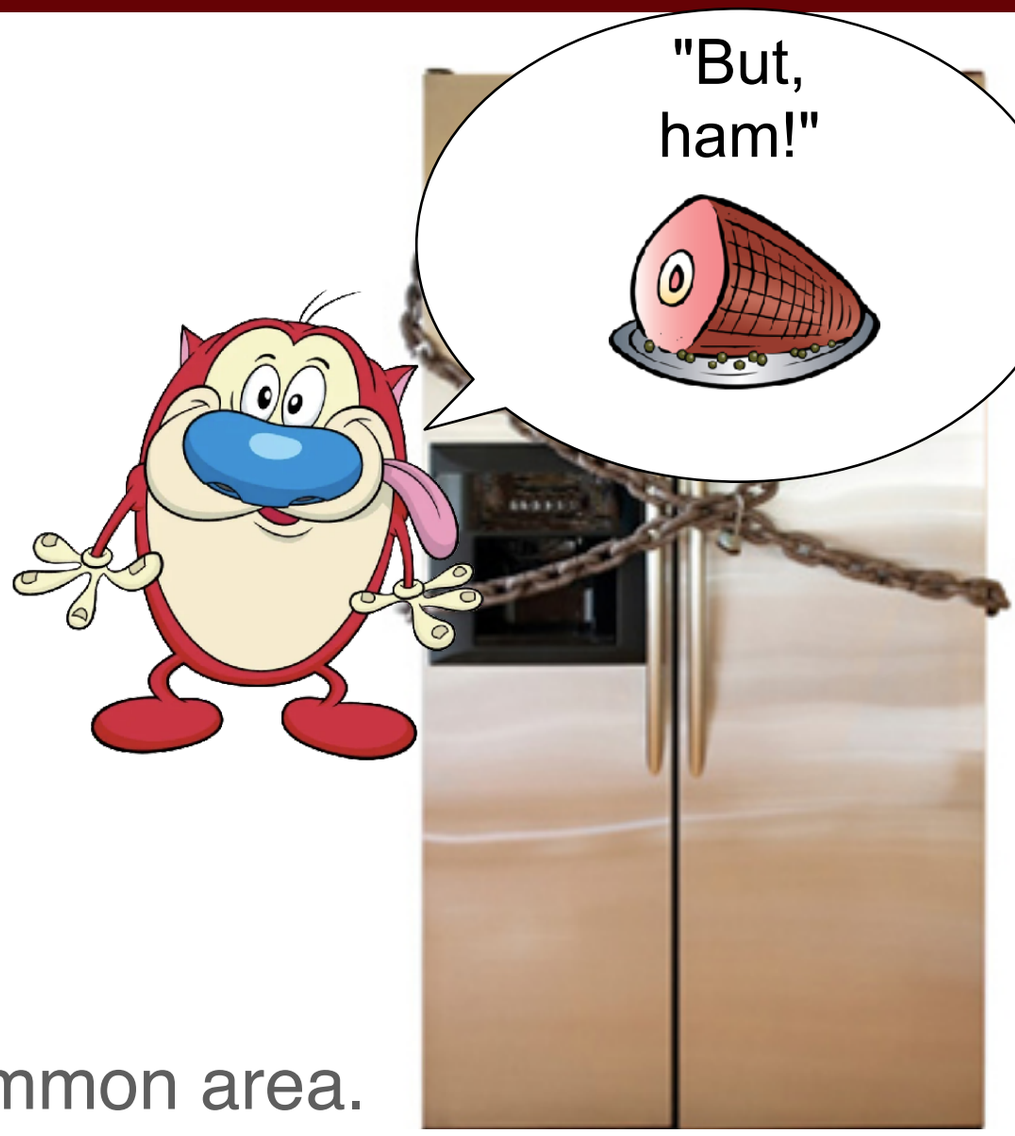
These "code blocks" are not atomic, and thus should not be allowed to interleave.



Solution #0: Lock the Fridge!

- What can be used as our "traffic light"?

```
while (1) {
   Lock the fridge & take the key
   if (noMilk) {
     buyMilk();
   }
   Unlock fridge
   Replace key in common area
}
```



- Is the pad-lock a correct CS solution?
  - Mutual exclusion? Yes.
  - Progress? Yes, grab the key from the common area.
  - Bounded Wait? Assuming Ren or Stimpy don't lose key and returns home, yes.
  - **It works!** But it's a poor solution. Poor performance to shared resource.

Solution #1: Leave a Note

- Leave note instead! Still allows fridge to be accessed for other items.

```
while (1) {
  if (noMilk) {
    if (noNote) {
      leaveNote();
      buyMilk();
      removeNote();
    }
  }
}
```

- *Is the code correct?*
  - Mutual exclusion? *No! Could still have too much milk!*
  - Progress? *Doesn't matter*
  - Bounded Wait? *Doesn't matter*

Solution #2: Leave Note First!

- We left the note too late! Instead, let's:
  - Leave a note first
    - (Don't let my roommate inspect the fridge while I'm inspecting)
  - Check if no milk, then go buy
  - Remove note after buying

- Don’t buy milk if there is a note.

```
while (1) {
  leaveNote();
  if (noMilk) {
    if (noNote) {
      buyMilk();
    }
  }
  removeNote();
}
```

Mutual Exclusion? Well, yes, because ***no thread*** *ever enters.*

Solution #3? Use Colored Notes!

- Distinguish the notes (lesson from Solution #2)
  - Ren uses blue notes
  - Stimpy uses red notes
- Leave a note first (lesson from Solution #1)
  - Check if no note from *other* roommate
  - Check if no milk, then buy
  - Remove note after buying
- Don't buy milk if there is a note

Solution #3 (Cont.)

- This solution guarantees **mutual exclusion**.
  - Is there still a problem with **progress** and **bounded wait**?

Stimpy's critical section

```
while (1) {
   leaveNote(stimpy);
   if (noNote(ren)) {
      if (noMilk)
         buyMilk();
   }
   removeNote(stimpy);
}
```

Ren's critical section

```
while (1) {
   leaveNote(ren);
   if (noNote(stimpy)) {
      if (noMilk)
         buyMilk();
   }
   removeNote(ren);
}
```

Solution #3 (Cont.)

- This solution guarantees **mutual exclusion**.
  - How about **progress** and **bounded wait**?
    - Progress is not guaranteed.
      - Both threads leave notes right before either thread checks for note
    - Bounded wait is not guaranteed.
      - Say Stimpy wants in, but always lags behind Ren, so he always sees Ren's note

Stimpy's critical section

```
while (1) {
  leaveNote(stimpy);
  if (noNote(ren)) {
     if (noMilk)
        buyMilk();
  }
  removeNote(stimpy);
}
```

Ren's critical section

```
while (1) {
  leaveNote(ren);
  if (noNote(stimpy)) {
     if (noMilk)
        buyMilk();
  }
  removeNote(ren);
}
```

Locks

- **Bottom Line:** We need a simple mechanism that can scale to any number of threads!
  - Desired usage:

```
acquire(lock);
//critical section starts
if (noMilk) {
  buyMilk();
}
//critical section ends
release(lock);
```

  - Preferably the two operations are in user space so we don't need to bother the OS each time we want to (un)lock.

Goals for This Lecture...

- Basic Problem Definition
- Mechanisms to Control Access to Shared Resources
  - Low Level Mechanisms:
    - Disabling (masking) interrupts
    - Busy-Waiting (Spin) Locks
    - Self-Blocking Locks
  - High Level Mechanisms:
    - Semaphores
    - Condition Variables and Monitors

Implement Locks by Disabling Interrupts?

- First try at implementing locks in user space.
  - Attack the source: interrupts!
    - We need to stop context switches from occurring, and they occur on interrupt!

- Idea: Let users disable and enable interrupts!
  - **`cli`** is the assembly instruction to disable interrupts on Intel x86.
  - **`sti`** is the assembly instruction to enable them.
  - Should we let **`cli`** and **`sti`** be unprivileged so anyone can call them??

Implement Locks by Disabling Interrupts (3)

- Problems Abound!
  - What if the thread needs to do I/O or make syscall in the CS? **Can't!**
  - What if programmer forgets to unlock the CS before leaving? **Stuck.**
  - Or, what if a user makes a silly mistake like the following... **Stuck.**

```
void davidsThread() {
  cli();
  while (1);  // Oops I put a ; here
  {

    /* important critical
       section code here */
  }
  sti();
}
```

Summary of Using `cli` and `sti` as Locks

- System responsiveness?
  - A programmer can freeze up the whole system
- Effects on multicore CPUs?
  - Disabling interrupts is done on granularity of a single core!
    - Other cores unaffected; Can still run threads that access critical section
    - Must disable interrupts on all cores at once (There's no such thing)

- In summary, this is too aggressive (like locking out the whole fridge)
  - Any user thread could bring the system to a crawl or break it!
  - cli and sti must be privileged, and we need to find a better way

Goals for This Lecture...

- Basic Problem Definition
- Mechanisms to Control Access to Shared Resources
  - Low Level Mechanisms:
    - Disabling interrupts
    - Busy-Waiting (Spin) Locks
    - Self-Blocking Locks
  - High Level Mechanisms:
    - Semaphores
    - Condition Variables and Monitors

Another Attempt: Spin Locks (User Mode)

- A 2nd attempt at a **user-space** locking solution: *Spin locks*
  - "Busy waiting"

- Spin lock definition:

```
typedef struct lock_t {
  int held = 0;
} lock_t;


void acquire(lock_t *L) {
  while (L->held)
     ; // Busy wait (spin)

  // We're in! Lock up!
  L->held = 1;
}


void release(lock_t *L) {
  L->held = 0;
}
```

- Example Usage:

```
// myLock shared among threads
lock_t myLock;


// pthreads run this
void *threadWork(void *args) {
   acquire(&myLock);

   // <do critical stuff>

  release(&myLock);
}
```

Spin Locks (Cont.)

- But does the busy-wait loop work?
- Let's say a thread **T1** is in the critical section
  - OS switches to **T2**
  - Then `acquire()` is called by thread **T2**
  - **T2** waits... eventually switches back to **T1**
  - **T1** calls `release()` on the lock:
    - `l->held` now 0
  - **T2** exits spin loop
    - But what if a switch back to T1 occurs *right before* T2 sets `l->held` to 1?

```
void acquire(lock_t *L) {
  while (L->held)
     ;
    Context Switch here (Noooooooo)
  L->held = 1;
}
```

acquire() and release() Need to Be Atomic!

- **Problem:** the code for `acquire()` and `release()` are *also critical sections!*
  - *Can't be broken up. There is a race condition!*

```
typedef struct lock_t {
  int held = 0; //shared
} lock_t;


void acquire(lock_t *L) {
  while (L->held) {
     ;
  }
  L->held = 1;
}


void release(lock_t *L) {
  L->held = 0;
}
```

There's a critical section in `acquire()` !!

This block of *"test it, then set it"* code needs to be atomic!

`release()` is already atomic.

(Recall assignment statements are guaranteed atomic by the CPU!)

Need User-Mode Hardware Support!!!

- Most CPUs provide a **`test_and_set()`** op to be *an atomic* user *operation!*

```
int test_and_set(int *flag) {
  int res = *flag;
  *flag = 1;
  return res;
}
```

CPU hardware guarantees this code block cannot be broken up (i.e., it is atomic)

- Need to combine the **while-test** and the **assignment statement** into **one** indivisible (atomic) CPU instruction!

```
void acquire(lock_t *L) {
   while (L->held) {
      ;
   }
   L->held = 1;
}
```

re-written atomically

```
void acquire(lock_t *L) {
   while (test_and_set(&L->held))
      ;
}
```

A working user-mode spin lock! (Yes!!)

Using Spin-Locks in C: API

- The **pthread** Library supports spin locks
  - (OS X doesn't support it due to energy efficiency)

```c
#include <pthread.h>

/**
 * Initializes a spin lock.
 * @param pshared PTHREAD_PROCESS_SHARED if shared in multiple processes, or
 *                PTHREAD_PROCESS_PRIVATE if lock is shared between threads
 *.               in same process.
 */
pthread_spin_init(pthread_spinlock_t *lock, int pshared);

/** Destroys (frees resources for) a lock */
pthread_spin_destroy(pthread_spinlock_t *lock);

/** Acquire */
pthread_spin_lock(pthread_spinlock_t *lock);

/** Release */
pthread_spin_unlock(pthread_spinlock_t *lock);
```

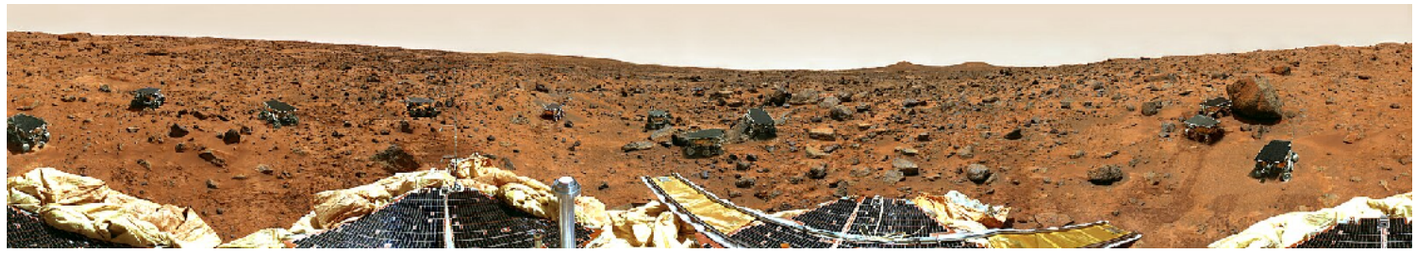
Summary of Spin Locks

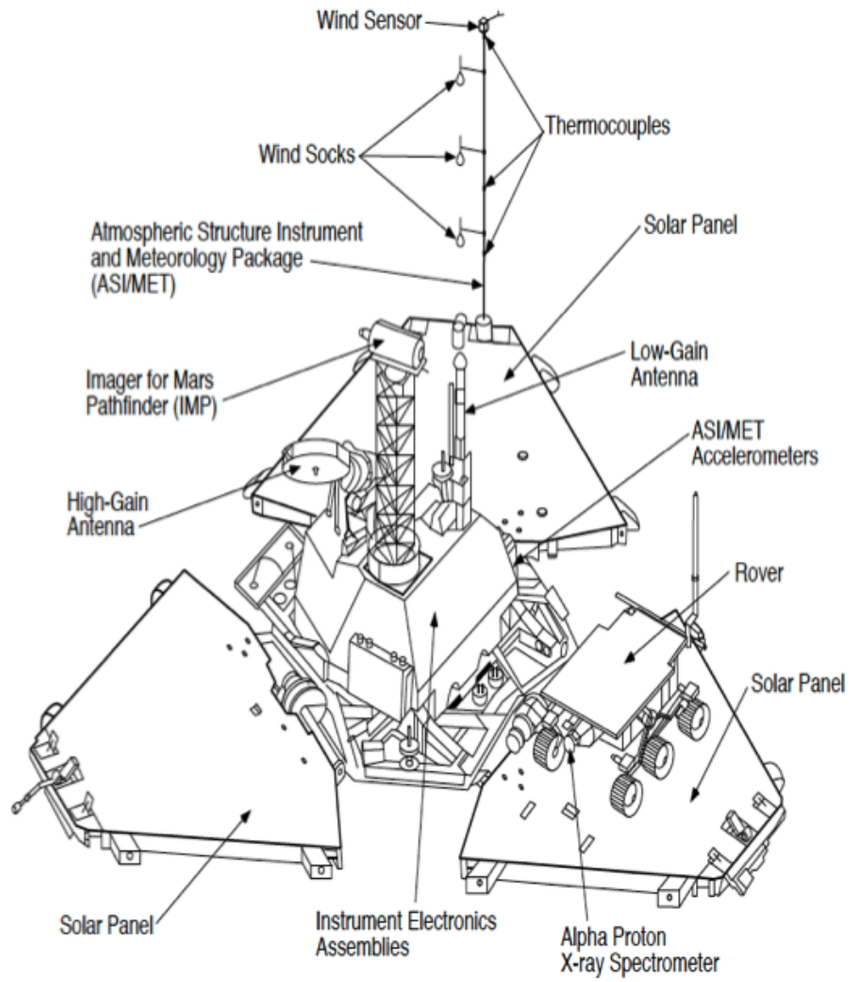
- Pros:
  - Easy to implement with help from CPU hardware
    - Exploit atomic user operations like test_and_set(), cmpxchg(), ...
  - **All done in user mode**! Fast!!
  - Don't have to manually disable interrupts!
    - Works on multi-core processors
    - Can be used to lock out individual data elements
      - Use a different lock for each array element (instead of locking out entire array)
- Cons:
  - Busy waiting is wasteful!
  - Susceptible to the *Priority Inversion Problem*

Big Problem with Spin-Locks: Priority Inversion

- *Priority-Inversion Problem*
  - High priority thread **starves** because a low priority thread has a spin-lock that the high priority thread needs.

- Mars Pathfinder/Sojourner Mission (1997)
  - Repeated system reset while on Mars.





Pathfinder Project

Priority Inversion (2)

- Shared resource: InfoBus (just a shared malloc'd array)
- Threads:
  - T_c: **Communication thread (Med priority)**
    - Slow, long running. Sends data and images to earth
      - » 1 packet @ speed of light, 3 min (closest) to 22 min (farthest)
    - Work is entirely unrelated to the InfoBus
  - T_I : **Info thread (High priority)**
    - Moves data in and out of InfoBus (need synchronization)
    - If this thread doesn't get to run periodically, system resets!
  - T_w : **Weather gathering thread (Low priority)**
    - Puts data into InfoBus (need synchronization)
    - Runs infrequently

Priority Inversion (3)

Tw (Low Priority)	Ti (High Priority)	Tc (Med Priority)
`acquire();`		
`// Got in! Write weather data to InfoBus`		
	(I need to access the InfoBus, pronto!)	
	`acquire();`	
	; (spin wait)	
		send cool image to earth
		still sending (it's big)
	; (Ti gets scheduling priority over Tw)	
	; (more spinning)	
	; (It can't make progress!)	
	; (It can't make progress!)	***Still sending...***
	SYSTEM RESET	

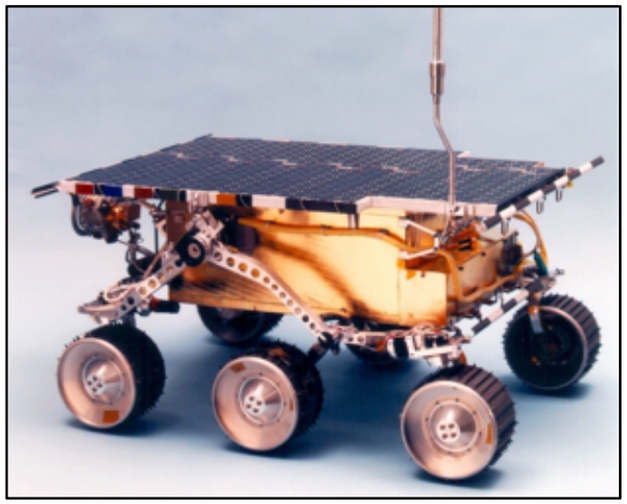
T_w starves because of priority scheduling.

T_i also starves because of T_w's hold on spin lock.

OS detects T_i isn't running. **Reboot!**

Big Problem with Spin-Locks: Priority Inversion

- *Priority Inversion Problem*
  - High priority thread **starves** because a low priority thread has lock on item that the high priority thread needs
    - (And Low priority thread can't get scheduled to release the lock!)



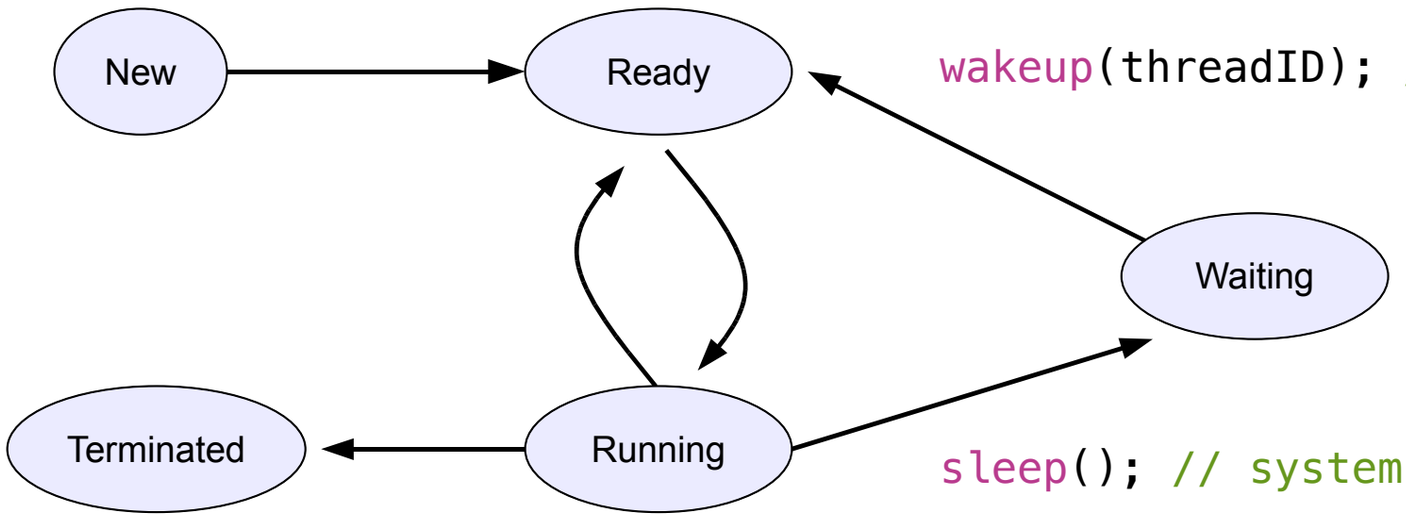
- Solution: use a *priority inheritance* policy
  - Low priority threads gets highest priority of all threads waiting on that same lock.

Goals for This Lecture...

- Basic Problem Definition
- Mechanisms to Control Access to Shared Resources
  - Low Level Mechanisms:
    - Disabling interrupts
    - Busy-Waiting (Spin) Locks
    - Self-Blocking Locks
  - High Level Mechanisms:
    - Semaphores
    - Condition Variables and Monitors

Alternative to Spinning? Blocking

- Spin locks are wasteful and susceptible to priority inversion
  - Instead of spinning (thread in ***Running & Ready*** state)
  - Make threads waiting on locks go to sleep (or "self-block") to put thread in ***Waiting*** state.




```
wakeup(threadID); // system call to wake a job up
```

```
sleep(); // system call to self-suspend
```

Implementing Blocking Locks in User Mode

- Need to have a queue of process/threads waiting on the lock

```
typedef struct blockinglock_t {
  int held = 0;
  struct queue *waitq; //need a wait-queue of thread ids
} blockinglock_t;
```

```
void acquire(blockinglock_t *L) {
  if (L->held) {
    enqueue(getpid(), L->waitq);  //put myself on the lock's wait queue
    sleep();                      //put myself to sleep
  }
  else
    L->held = 1;
}
```

Implementing Blocking Locks in User Mode

- Need to have a queue of process/threads waiting on the lock

```
typedef struct blockinglock_t {
  int held = 0;
  struct queue *waitq; //need a wait-queue of thread ids
} blockinglock_t;
```

```
void acquire(blockinglock_t *L) {
  if (L->held) {
    enqueue(getpid(), L->waitq);  //put myself on the lock's wait queue
    sleep();                      //put myself to sleep
  }
  else
    L->held = 1;
}

void release(blockinglock_t *L) {
   if (isEmpty(L->waitq))
     L->held = 0;
   else
     wakeup(dequeue(l->waitq));   //dequeue next process, ask OS to make it ready
                                  //don't set l->held to 0 in else clause
}
```

But acquire and release have critical sections! Lock them up!

Implementing Blocking Locks in User Mode (Try 0)

- Use a **spinlock** to guard the critical sections of the blocking lock!

```
typedef struct blockinglock_t {
  int held = 0;
  spinlock_t *mutex;   // Use a spinlock to guard acquire() and release()
  struct queue *waitq; //need a wait-queue of thread ids
} blockinglock_t;
```

```
void acquire(blockinglock_t *L) {
  spinlock_acquire(L->mutex);
  if (L->held) {
      enqueue(getpid(), L->waitq);
      sleep();
  }
  else
      L->held = 1;
  spinlock_release(L->mutex);
}

void release(blockinglock_t *L) {
  spinlock_acquire(L->mutex);
  if (isEmpty(L->waitq))
      L->held = 0;
  else
      wakeup(dequeue(L->waitq));
  spinlock_release(L->mutex);
}
```

The Problem with "Try 0": Deadlock

T1	T2
`acquire() { ...`	
`  spinlock_acq()`	
`    L->held = 1`	
`  spinlock_rel()`	
`}`	
<<Crit. Sec.>>	
	`acquire() { ...`
	`  spinlock_acq()`
	`  enqueue(...)`
	`  sleep();`
`release() {`	
`  spinlock_acq()`	
(spin forever)	(sleep forever)

```
void acquire(blockinglock_t *L) {
  spinlock_acquire(L->mutex);
  if (L->held) {
      enqueue(getpid(), L->waitq);
      sleep();
  }
  else
      L->held = 1;
  spinlock_release(L->mutex);
}


void release(blockinglock_t *L) {
  spinlock_acquire(L->mutex);
  if (isEmpty(L->waitq))
      L->held = 0;
  else
      wakeup(dequeue(L->waitq));
  spinlock_release(L->mutex);
}
```

Implementing Blocking Locks in User Mode (Try 1)

- All right, then swap sleep and spinlock_release!

```
typedef struct blockinglock_t {
  int held = 0;
  spinlock_t *mutex;   // Use a spinlock to guard acquire() and release()
  struct queue *waitq; //need a wait-queue of thread ids
} blockinglock_t;
```

```
void acquire(blockinglock_t *L) {
  spinlock_acquire(L->mutex);
  if (L->held) {
      enqueue(getpid(), L->waitq);
      spinlock_release(L->mutex);
      sleep();
  } else {
      L->held = 1;
      spinlock_release(L->mutex);
  }
}

void release(blockinglock_t *L) {
  spinlock_acquire(L->mutex)
  if (isEmpty(L->waitq))
     L->held = 0;
  else
     wakeup(dequeue(L->waitq));
  spinlock_release(L->mutex);
}
```

Ah HA! Release spinlock first, *then* sleep!

The Problem with "Try 1": Starvation

T1	T2
`acquire() { ...`	
`  spinlock_acq()`	
`    L->held = 1`	
`  spinlock_rel()`	
`}`	
<<Crit. Sec.>>	
	`acquire() { ...`
	`  spinlock_acq()`
	`  enqueue(...)`
	`  spinlock_rel()`
`release() {`	
`  spinlock_acq()`	
`  dequeue(...)`	
`  wakeup(...)`	*<-- is lost! T2 not asleep yet!*
`  spinlock_rel()`	`  sleep();`
`}`	`  (sleep forever)`

```
void acquire(blockinglock_t *L) {
  spinlock_acquire(L->mutex);
  if (L->held) {
      enqueue(getpid(), L->waitq);
      spinlock_release(L->mutex);
      sleep();
  } else {
      L->held = 1;
      spinlock_release(L->mutex);
  }
}


void release(blockinglock_t *L) {
  spinlock_acquire(L->mutex)
  if (isEmpty(L->waitq))
     L->held = 0;
  else
     wakeup(dequeue(L->waitq));
  spinlock_release(L->mutex);
}
```

Implementing Blocking Locks in User Mode (Correct)

```
typedef struct blockinglock_t {
  int held = 0;
  spinlock_t *mutex;   // Use a spinlock to guard acquire() and release()
  struct queue *waitq; // Wait queue of pids
} blockinglock_t;
```

```
void acquire(blockinglock_t *L) {
   spinlock_acquire(L->mutex);
   if (L->held) {
    enqueue(getpid(), L->waitq);
    spinlock_release_and_sleep(L->mutex); // sleep() needs to release() all locks
                                          // being held by the thread atomically
  }
  else {
    L->held = 1;
    spinlock_release(L->mutex);
  }
}

void release(blockinglock_t *L) {
    spinlock_acquire(L->mutex);
    if (isEmpty(L->waitq))
        L->held = 0;
    else
        wakeup(dequeue(L->waitq));
    spinlock_release(L->mutex);
}
```

Need `sleep()` to also `release()` atomically!
(Ask CPU maker for atomic support)

Summary of Blocking Locks

- Pros:
  - No more busy-waiting. CPU is less taxed.
  - Avoids priority inversion problem! *(why?)*
  - Some of the implementation is in user-mode. Transitions to kernel-mode only when sleep() and wakeup() are called.

- Cons:
  - Need to manage a queue per lock
    - Queue management must be atomic, but it's a small critical section
      - Most OS will just use a spin lock for lock-queue management
  - `sleep()` and `wakeup()` need system calls, so there's an overhead
    - *Think: When would you prefer a spin lock over a blocking lock? When would you prefer the opposite?*

No Priority Inversion?

Tw (Low Priority)	Ti (High Priority)	Tc (Med Priority)
`acquire();`		
`write to InfoBus`		
`...`		
	`acquire() { ...`	
	`sleep();  // zzzzzzz`	
		`send cool image to earth`
		`....`
`...`		
`...`		
`release();`	**`(You woke me!)`**	
	`Yay I get to run. Rover happy.`	

Ti is no longer in contention with Tw in the ready queue!

Using Blocking Locks in C

- The Blocking Lock type is **`pthread_mutex_t`**

```
#include <pthread.h>

pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutex_attr *attr)

pthread_mutex_destroy(pthread_mutex_t *mutex)

pthread_mutex_lock(pthread_mutex_t *mutex)

pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Usage Example

```c
//shared lock
pthread_mutex_t *lock;

int main() {
    lock = (pthread_mutex_t*) malloc(sizeof(pthread_mutex_t));
    pthread_mutex_init(lock, NULL);  //sets it to "unlocked" state

    // ... more code ...
    // (creating threads...)

    // ... more code ...
    // (joining threads)
    pthread_destroy(lock);
    return 0;
}

void* threadWork(void* args) {
    pthread_mutex_lock(lock);  //try to acquire the lock

    ////////////////////////////////////////
    // << critical section >>
    ////////////////////////////////////////

    pthread_mutex_unlock(lock); //release
}
```

Full Circle (Remember This Race Condition?)

```c
#include <pthread.h>
#include <stdio.h>

int x = 0;

void* doStuff(void*);

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, doStuff, NULL);
    pthread_create(&t2, NULL, doStuff, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Value of x is %d\n", x);

    return 0;
}

void* doStuff(void* args) {
    int i;
    for (i = 0; i < 1000000; i++)
        x++;
    return NULL;
}
```

I want to increment x to 2,000,000 but split the work across 2 cores!

Output:

```
$ gcc add2m.c -lpthread -o add2m
$ ./add2m
Value of x is 1029184

$ ./add2m
Value of x is 1224682

$ ./add2m
Value of x is 1004207

$ ./add2m
Value of x is 1003645
```

Full Circle - Using **Self-Blocking** Locks

```c
int x = 0;
pthread_mutex_t *lock;

void* doStuff(void*);

int main() {
    //initialize the lock
    lock = (pthread_mutex_t*) malloc(sizeof(pthread_mutex_t));
    pthread_mutex_init(lock, NULL);

    pthread_t t1, t2;
    pthread_create(&t1, NULL, doStuff, NULL);
    pthread_create(&t2, NULL, doStuff, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Value of x is %d\n", x);
    pthread_mutex_destroy(lock);
    return 0;
}


void* doStuff(void* args) {
    int i;
    for (i = 0; i < 1000000; i++) {
        pthread_mutex_lock(lock);
        x++;
        pthread_mutex_unlock(lock);
    }
    return NULL;
}
```

Fixes race condition, but now much slower than sequential version! (Ugh... Amdahl's Law)

Full Circle - Using **Spin** Locks

```c
int x = 0;
pthread_spinlock_t *lock;

void *doStuff(void *);

int main() {
  // initialize the lock
  lock = (pthread_spinlock_t *)malloc(sizeof(pthread_spinlock_t));
  pthread_spin_init(lock, PTHREAD_PROCESS_PRIVATE);

  pthread_t t1, t2;
  pthread_create(&t1, NULL, doStuff, NULL);
  pthread_create(&t2, NULL, doStuff, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("Value of x is %d\n", x);
  pthread_spin_destroy(lock);

  return 0;
}

void *doStuff(void *args) {
  int i;
  for (i = 0; i < 1000000; i++)
  {
    pthread_spin_lock(lock);
    x++;
    pthread_spin_unlock(lock);
  }
  return NULL;
}
```

This version will run faster than blocking locks.... (why?)

Administrivia 3/14

- Today: Start Chapter 6
  - Race conditions
    - Concurrency leads to nondeterministic outcomes
    - Interleaving schedules (i.e., context switching is the culprit)
      - But context switching is a good thing
      - We can't just go back to batch processing!
  - Critical section problem

Administrivia 3/24

- Announcements:
  - Hwk 6 posted! Due Wednesday, 4/9
- Last time...
  - Enforcing atomicity
  - The critical section problem
- Today:
  - Review the "Too Much Milk" Problem
  - Building locks: Spin locks (busy waiting)
  - Building self-blocking locks (not covered in book)

Administrivia 3/26

- Announcements:
  - Hwk 6 posted! Due Wednesday, 4/9
- Last time...
  - Peterson's Solution to the TMM problem (only 2 threads)
  - Building user-mode locks: Spin locks (busy waiting)
    - Importance of test-and-set() as an unprivileged atomic instruction
- Today
  - Spinlocks continued: Priority inversion problem
    - In C: pthread_spinlock_t
  - Building self-blocking locks (not covered in book)
    - In C, pthread_mutex_t