Operating Systems
CSCI 3150

Lecture 5: Synchronization (I) -- Critical region and lock

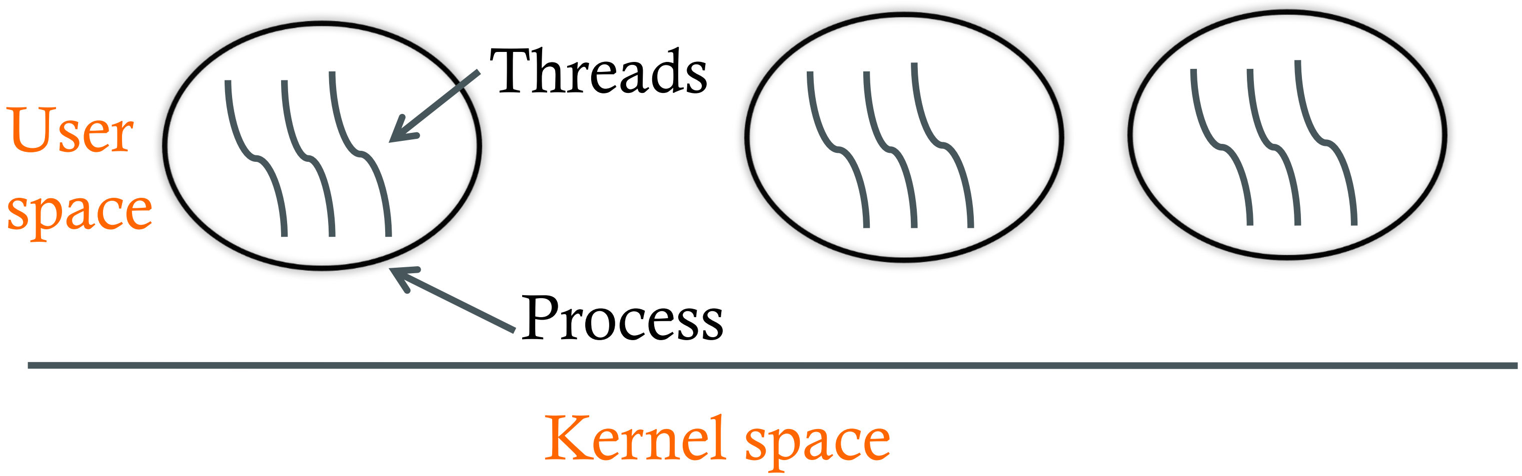
Hong Xu

Review of last lecture

- Multithreading is also very useful for applications
  - Efficient multithreading requires fast primitives
  - Processes are too heavyweight
- Solution is to separate threads from processes
  - Kernel-level threads much better, but still significant overhead
  - User-level threads even better, but not well integrated with OS
- Now, how do we get our threads to correctly cooperate with each other?
  - Synchronization…

Synchronization: why?

- A running computer has multiple processes and each process may have multiple threads



- Need proper sequencing
- Analogy: two people talking at the same time

A simple game

- Two volunteers to play two threads
  - Producer: produce 1 snack bar per iteration
    - Step1: increment the counter on the board
    - Step2: put one cookie on the table
  - Consumer:
    - Step1: read the counter LOUD
    - Step2a: if the counter is zero, go back to step1
    - Step2b: if the counter is nonzero, take a snack bar from the table
    - Step 3: decrement counter on the board
  - Rule: only one should "operate" at any time
- You are the OS
  - You decide who should operate, who should freeze
  - Can you get them into "trouble" before snack bars run out?

A simple game (cont.)

- Producer: produce 1 cookie per iteration
  - Step1: increment the counter on the board
  - Step2: put one snack bar on the table
- Consumer:
  - Step1: read the counter LOUD
  - Step2a: if the counter is zero, go back to step1
  - Step2b: if the counter is nonzero, take a cookie from the table
  - Step 3: decrement counter on the board

Switch to consumer, what will happen?

Switch to producer, what will happen?

Data races

- Why are we having this problem?
- Reason:
  - concurrency
  - data sharing
- What are shared in this game?
  - Share the counter
  - Share the cookie

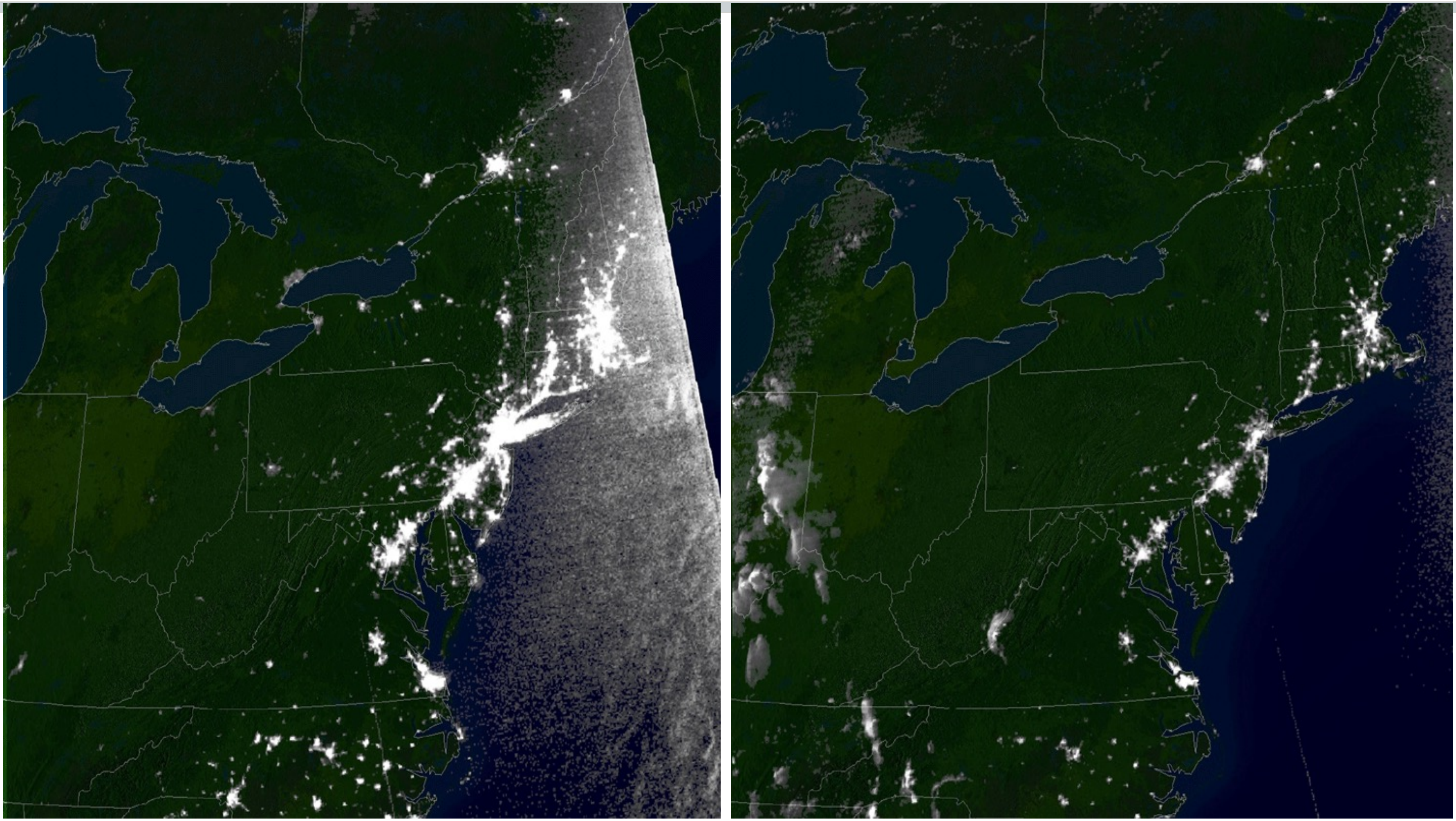
Shared Resources

- The problem is that two concurrent threads (or processes) accessed a shared resource without any synchronization
  - Known as a race condition (memorize this buzzword)
- We need mechanisms to control access to these shared resources in the face of concurrency
  - So we can reason about how the program will operate
- Shared data structure
  - Buffers, queues, lists, hash tables, etc.

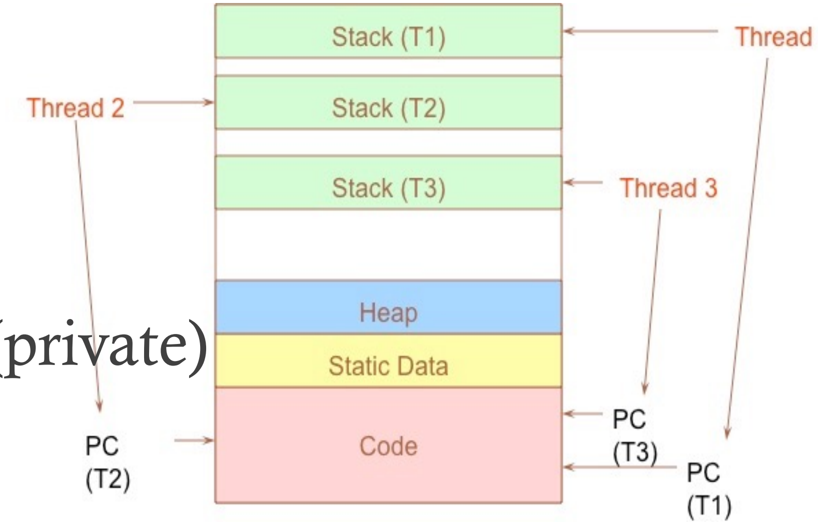
What is data race?

A *race* occurs when correctness of the program depends on one thread reaching point x before another thread reaches point y

2003 Northeast blackout is caused by data race



When are resources shared?



- Local variables are not shared (private)
  - Stored on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on the stack for thread T1 to another thread T2
- Global variables and static objects are shared
  - Stored in the static data segment, accessible by any thread
- Dynamic objects and other heap objects are shared
  - Allocated from heap with malloc/free or new/delete
- Accesses to shared data need to be synchronized

Why synchronize?

- Interleaving by an access from another thread to the same shared data between two subsequent accesses can result in errors

Write X

Write X

Read X

Analogy

- Synchronization is like traffic signals
  - Each thread is like a car----it can make progress independently with its own speed
  - Road or intersection is the shared resource
- https://www.youtube.com/watch?v=ho1cxThxEy8

Classic Example

- Suppose we have to implement a function to handle withdrawals from a bank account:

```
withdraw (account, amount) {
   balance = get_balance(account);
   balance = balance – amount;
   put_balance(account, balance);
   return amount;
}
```

- Now suppose that you and your significant other share a bank account with a balance of $1000.

- Then you each go to separate ATM machines and simultaneously withdraw $100 from the account.

Example Continued

- We'll represent the situation by creating a separate thread for each person to do the withdrawals
- These threads run on the same bank machine:

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return amount;
}
```

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return amount;
}
```

- What's the problem with this implementation?
  - Think about potential schedules of these two threads

Interleaved Schedules

- The problem is that the execution of the two threads can be interleaved:

Execution sequence seen by CPU

```
balance = get_balance(account);
balance = balance – amount;

balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);

put_balance(account, balance);
```

Context switch

- What is the balance of the account now?
- Is the bank happy with our implementation?
  - What if this is not withdraw, but deposit?

How Interleaved Can It Get?

How contorted can the interleavings be?

- We'll assume that the only atomic operations are reads and writes of words
  - Some architectures don't even give you that!
- We'll assume that a context switch can occur at any time
- We'll assume that you can delay a thread as long as you like as long as it's not delayed forever

.... get_balance(account);
balance = get_balance(account);
balance = ...
balance = balance – amount;
balance = balance – amount;
put_balance(account, balance);
put_balance(account, balance);

Mutual Exclusion

- We want to use mutual exclusion to synchronize access to shared resources
  - This allows us to have larger atomic blocks
- Code that uses mutual exclusion to synchronize its execution is called a critical region (or critical section)
  - Only one thread at a time can execute in the critical region
  - All other threads are forced to wait on entry
  - When a thread leaves a critical region, another can enter
  - Example: sharing your bathroom with housemates

Critical Region (Critical Section)

```
Process {
  while (true) {
      ENTER CRITICAL SECTION
       Access shared variables; // Critical Section;
      LEAVE CRITICAL SECTION
      Do other work
  }
}
```

- What requirements would you place on a critical section?

Critical Region Requirements (apply to both thread and process)

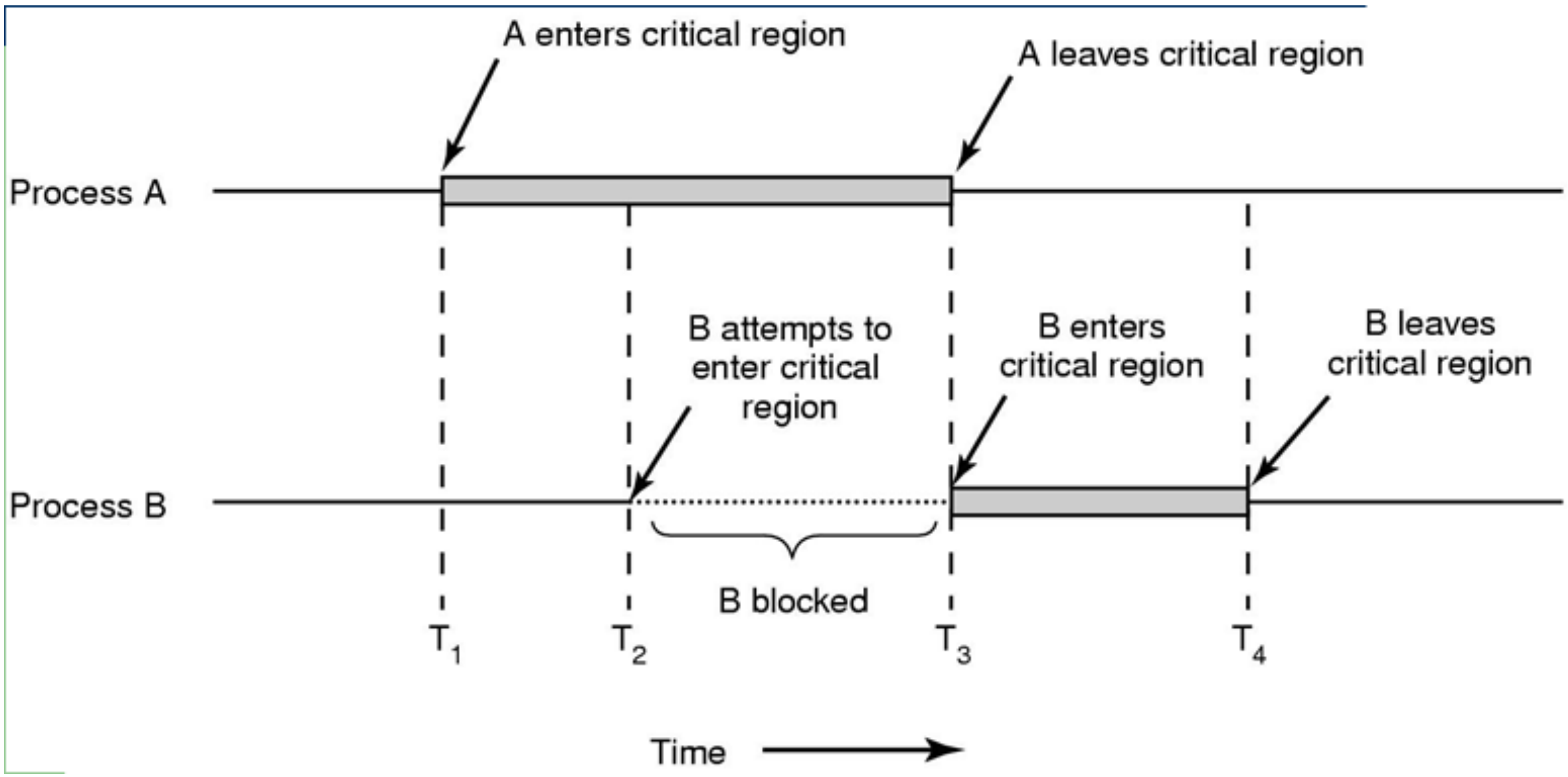
1) Mutual exclusion (mutex)
   - No other thread must execute within the critical region while a thread is in it

2) Progress
   - A thread in the critical region will eventually leave the critical region
   - If some thread T is not in the critical region, then T cannot prevent some other thread S from entering the critical region

3) Bounded waiting (no starvation)
   - If some thread T is waiting on the critical region, then T should only have wait for a bounded number of other threads to enter and leave the critical region

4) No assumption
   - No assumption may be made about the speed or number of CPUs

Critical Region Illustrated



Mechanisms For Building Critical Sections

- Atomic read/write
  - Can it be done?
- Locks
  - Primitive, minimal semantics, used to build others
- Semaphores
  - Basic, easy to get the hang of, but hard to program with
- Monitors
  - High-level, requires language support, operations implicit
- Messages
  - Simple model of communication and synchronization based on atomic transfer of data across a channel
  - Direct application to distributed systems
  - Messages for synchronization are straightforward (once we see how the others work)

Mutual Exclusion with Atomic Read/Writes: First Try

```
int turn = 1;
```

```
while (true) {
    while (turn != 1) ;
    critical region
    turn = 2;
    outside of critical region
}
```

```
while (true) {
    while (turn != 2) ;
    critical region
    turn = 1;
    outside of critical region
}
```

This is called alternation

It satisfies mutex:

- If blue is in the critical region, then turn == 1 and if yellow is in the critical region then turn == 2 (why?)
- (turn == 1) ≡ (turn != 2)

It violates progress: the thread could go into an infinite loop outside of the critical section, which will prevent the yellow one from entering.

Easy to use? (what if more than 2 threads? what if we don't know how many threads?)

Locks

- A lock is an object in memory providing two operations
  - acquire(): before entering the critical region
  - release(): after leaving a critical region
- Threads pair calls to acquire() and release()
  - Between acquire()/release(), the thread holds the lock
  - acquire() does not return until any previous holder releases
  - What can happen if the calls are **not paired**?
- Locks can spin (a spinlock) or block (a mutex)

Using Locks

```
withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    release(lock);
    return amount;
}
```

Critical Region

```
acquire(lock);
balance = get_balance(account);
balance = balance – amount;
```

```
acquire(lock);
```

```
put_balance(account, balance);
release(lock);
```

```
balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);
release(lock);
```

- What happens when blue tries to acquire the lock?
- Why is the "return" outside the critical region? Is this OK?
- What happens when a third thread calls acquire?

Implementing Locks (1)

- How do we implement locks? Here is one attempt:

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}
void release (lock) {
    lock->held = 0;
}
```

busy-wait (spin-wait)
for lock to be released

- This is called a spinlock because a thread spins waiting for the lock to be released
- Does this work?

Implementing Locks (2)

- No. Two independent threads may both notice that a lock has been released and thereby acquire it.

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (lock->held);
    lock->held = 1;
}
void release (lock) {
    lock->held = 0;
}
```

A context switch can occur here, causing a race condition

Implementing Locks (3)

- The problem is that the implementation of locks has critical sections, too
  - How do we stop the recursion?
- The implementation of acquire/release must be atomic
  - An atomic operation is one which executes as though it could not be interrupted
  - Code that executes "all or nothing"
- How do we make them atomic?
- Need help from hardware
  - Atomic instructions (e.g., test-and-set)
  - Disable/enable interrupts (prevents context switches)

Atomic Instructions: Test-And-Set

- The semantics of test-and-set are:
  - Record the old value
  - Set the value to TRUE
  - Return the old value
- Hardware executes it **atomically**!

```
bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

- When executing test-and-set on "flag"
  - What is value of flag afterwards if it was initially False? True?
  - What is the return result if flag was initially False? True?

Using Test-And-Set

- Here is our lock implementation with test-and-set:

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0;
}
```

- When will the while return? What is the value of held?
- Does it work? What about multiprocessors?

Problems with Spinlocks

- The problem with spinlocks is that they are wasteful
  - If a thread is spinning on a lock, then the thread holding the lock cannot make progress
- Solution 1:
  - If cannot get the lock, call thread_yield to give up the CPU
- Solution 2: sleep and wakeup
  - When blocked, go to sleep
  - Wakeup when it is OK to retry entering the critical region

Disabling Interrupts

- Another implementation of acquire/release is to disable interrupts:

```
struct lock {
}
void acquire (lock) {
    disable interrupts;
}
void release (lock) {
    enable interrupts;
}
```

- Note that there is no state associated with the lock
- Can two threads disable interrupts simultaneously?

On Disabling Interrupts

- Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)
  - This is what OS161 uses as its primitive
- In a “real” system, this is only available to the kernel
  - Why?
- Disabling interrupts is insufficient on a multiprocessor
  - Back to atomic instructions

Critical regions without hardware support?

- So far, we have seen how to implement critical regions (lock) with hardware support
  - Atomic instruction
  - Disabling interrupt
- Can we implement lock ***without*** HW support?
  - Software only solution?
- Yes, but…
  - Complicated (easy to make mistake)
  - Poor performance
  - Production OSes use hardware support

Mutex without hardware support: Peterson's Algorithm

```
int turn = 1;
bool try1 = false, try2 = false;
```

```
while (true) {
    try1 = true;
    turn = 2;
    while (try2 && turn != 1) ;
    critical section
    try1 = false;
    outside of critical section
}
```

```
while (true) {
    try2 = true;
    turn = 1;
    while (try1 && turn != 2) ;
    critical section
    try2 = false;
    outside of critical section
}
```

Did I execute “turn=2” before thread 2 executed “turn=1”?

Has thread 2 executed “try2=true?”. If not, I am safe. If yes, let’s see…

- Does it work?
    - Yes!
- Try all possible interleavings

Summarize Where We Are

- Goal: Use mutual exclusion to protect critical sections of code that access shared resources
- Method: Use locks (spinlocks or disable interrupts)
- Problem: Critical sections can be long

Spinlocks:

- Threads waiting to acquire lock spin in test-and-set loop
- Wastes CPU cycles
- Longer the CS, the longer the spin
- Greater the chance for lock holder to be interrupted

```
acquire(lock)
…
Critical section
…
release(lock)
```

Disabling Interrupts:

- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)

If you only remember one thing from this lecture…

- When you have ***concurrency*** & ***shared resources***, protect your critical region with synchronization primitives (e.g., locks, semaphore (next lecture), etc.)
  - You don't want to go to that crazy intersection in Russia.