Synchronization: Race Conditions

ICS332
Operating Systems

Henri Casanova (henric@hawaii.edu)

Context

- This whole module is a mere introduction to a large, complicated, and fundamental topic
- Most software is multi-threaded at some level, and threads need to “synchronize”
  - The term “synchronize” is a bit confusing
  - In this set of lecture notes it means “make sure threads don’t step on each other’s toes in RAM to ensure program correctness”
- Therefore, this topic is relevant to most software
- And it’s not easy!
  - Full hands-on experience in ICS 432
- We’ll only go through a subset of the material in OSTEP
  - 26.3, 26.4, 26.5
  - 28.1, 28.8, 28.12, 28.14

The main Pitfall of Concurrency

- “My machine is multicore, so I know I can have true concurrency, and I’ve learned how to program with threads! Let me implement a program that counts up to some value faster with more threads!!!”
  - As usual we start with something really useless :)
- One global variable: a counter that stores a value
- **`numThreads`** threads that each increment the counter by one over **`numIterations`** iterations
  - Let’s look at the code in CounterTestV1.java
- Let’s run this code for:
  - 1, 2, or many threads, small and large values of **`numIterations`**
  - What do we observe?

Understanding the Pitfall

- High-level programming languages (anything but assembly, and even not all assembly languages) hide the complexity of operations performed at the CPU level
- In C, incrementing a 4-byte value in RAM:

```
int *x;
*x += 1;
```

- Translates in (NASM) x86 assembly language to:

```
mov eax, [x]        // set register EAX to *x
inc eax             // increment register EAX
mov [x], eax        // set *x to the value of EAX
```

- In MIPS-like assembly, this would be like:

```
lw $t0, (x)         // set register t0 to *x
addi $t0, $t0, 1    // increment register t0
sw $t0, (x)         // set *x to the value of t0
```

Understanding the Pitfall

- High-level programming languages (anything but assembly, and even not all assembly languages) hide the complexity of operations performed at the CPU level
- In C, incrementing a 4-byte value in RAM:

```
    int *x;
    *x += 1;
```

- Translates in (NASM) x86 assembly language to:

```
    mov eax, [x]        // set register EAX to *x
    inc eax             // increment register EAX
    mov [x], eax        // set *x to the value of EAX
```

- In MIPS-like assembly, this would be like:

```
    lw $t0, (x)         // set register t0 to *x
    addi $t0, $t0, 1    // increment register t0
    sw $t0, (x)         // set *x to the value of t0
```

- The point: `x++` is done with 3 instructions

Understanding the Pitfall: 1 thread

- Execution with 1 thread

Instruction	Value of EAX	Value at [x]
	Undefined	0
load [x] into reg	0	0
increment reg	1	0
store reg into [x]	1	1
load [x] into reg	1	1
increment reg	2	1
store reg into [x]	2	2
load [x] into reg	2	2
increment reg	3	2
store reg into [x]	3	3
load [x] into reg	3	3
increment reg	4	3
store reg into [x]	4	4

Understanding the Pitfall: 2 threads

- Let’s play the role of the OS scheduler with a “blue” thread and a “red” thread

Instruction	Value of reg	Value at [x]
	Undefined	0
load [x] into reg	**0**	0
increment reg	**1**	0
store reg into [x]	**1**	1
load [x] into reg	**1**	1
Context Switch from blue to red Saved blue registers: **reg = 1, PC = …, etc.** Restored red registers: **reg = undef, PC = …, etc**		
	Undefined	1
load [x] into reg	**1**	1
increment reg	**2**	1
store reg into [x]	**2**	2
Context Switch from red to blue Saved red registers: **reg = 2, PC = …, etc.** Restored blue registers: **reg = 1, PC = …, etc**		
	1	2
increment reg	**2**	2
store reg into [x]	**2**	2

☠ This is wrong ☠

We executed 3 INCREMENT instructions

We executed 3 STORE instructions (just like the 1-thread execution)

Yet our final value in RAM is 2 and not 3!!

Just because the OS did Context Switch #2 at the “wrong” time!

Race Condition

- The behavior on the previous slide is called a Race Condition
    - Which means we have a concurrency bug
    - In this case the bug is called a lost update
- The outcome depends on when context-switches occur
- When running our Java code, we witnessed many lost updates for large values of `numIterations`
- But:
    - The bug manifests itself differently for each execution
    - The bug may manifest itself very rarely for small values of n, **and yet the program is still buggy!**
- Such non-deterministic bugs make concurrent programming difficult
    - The whole “I tested the code 10,000 times, and then the user got a bug” problem...

Lost Update Example

- In general when a thread does x+=a and an another does x+=b three things can happen:
  - Both updates go through and x is incremented by a+b
  - The x+=a update is lost and x is incremented only by a
  - The x+=b update is lost and x is incremented only by b

- Example:
  - One variable: `a` initially set to 1
  - Thread #1: `a+=1;`
  - Thread #2: `a-=1;`
  - Once both threads are finished, the value of `a` is output
  - Question: What are the possible output?

Lost Update Example

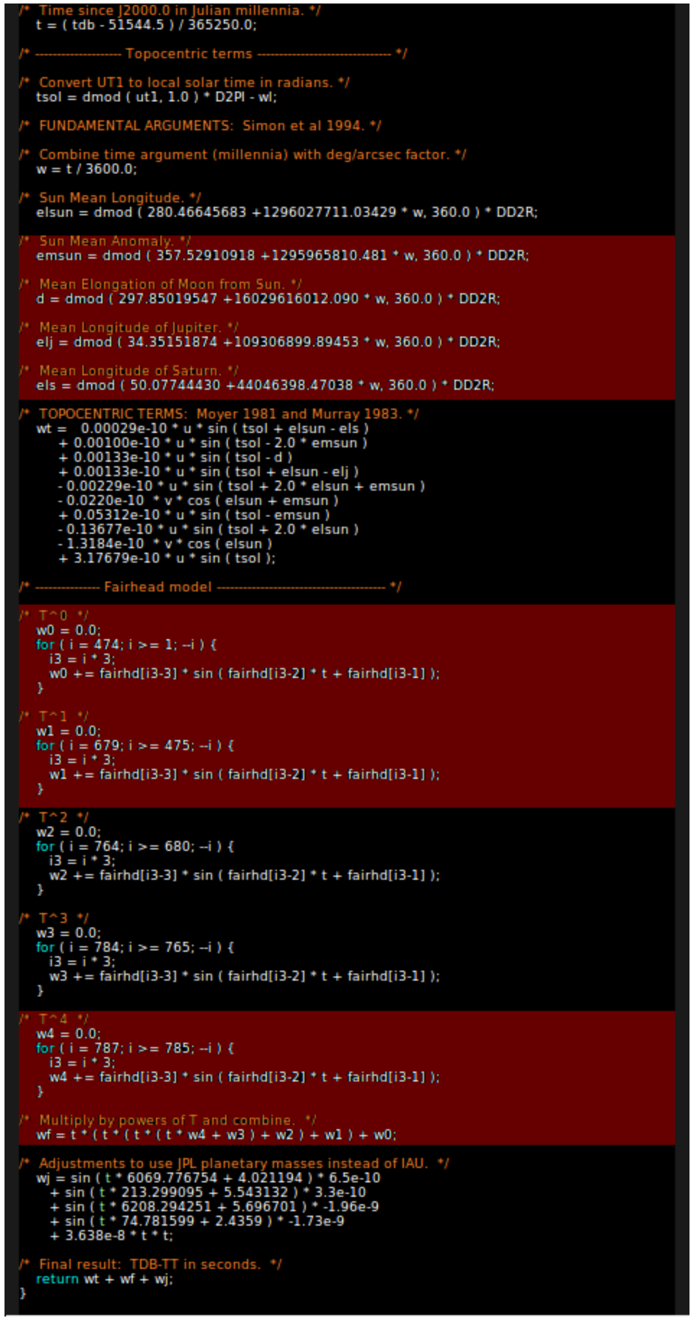
- The setup:
  - One variable: `a` initially set to 1
  - Thread #1: `a+=1;`
  - Thread #2: `a-=1;`
- There are four possible executions:
  - Blue before Red: final value of `a` is **1 (correct)**
  - Red before Blue: final value of `a` is **1 (correct)**
  - Blue starts, but gets context-switched out before writing to RAM, and so the red update is lost: final value of `a` is **2 (incorrect)**
  - Red starts, but gets context-switched out before writing to RAM, and so the blue update is lost: final value of `a` is **0 (incorrect)**
- So this program can output 0, 1, or 2
- And we cannot be 100% certain of what it will output

How do we fix this?

- Clearly, if we “just add threads” to a sequential program and have threads read/write the same memory locations, we’ll be in trouble
- Yet, we want them to read/write the same memory locations for them to co-operate
  - That’s the whole point of having threads that share an address space as opposed to processes that each live in their own world
- We need a new programming concept that ensures that threads do not “step on each other’s toes”
- This concept is called a critical section

Critical Section

- **A critical section is a region of code in which only one thread can be at a time**
  - If a thread is already executing code in the critical section then all other threads are “blocked” before being allowed to enter the critical section
  - Only one thread will be allowed to enter when a thread leaves the critical section
- A critical section does not have to be a contiguous section of code
  - In the example here, we have a 3-zone critical section (displayed in red)
- Real-life metaphor: a public bathroom

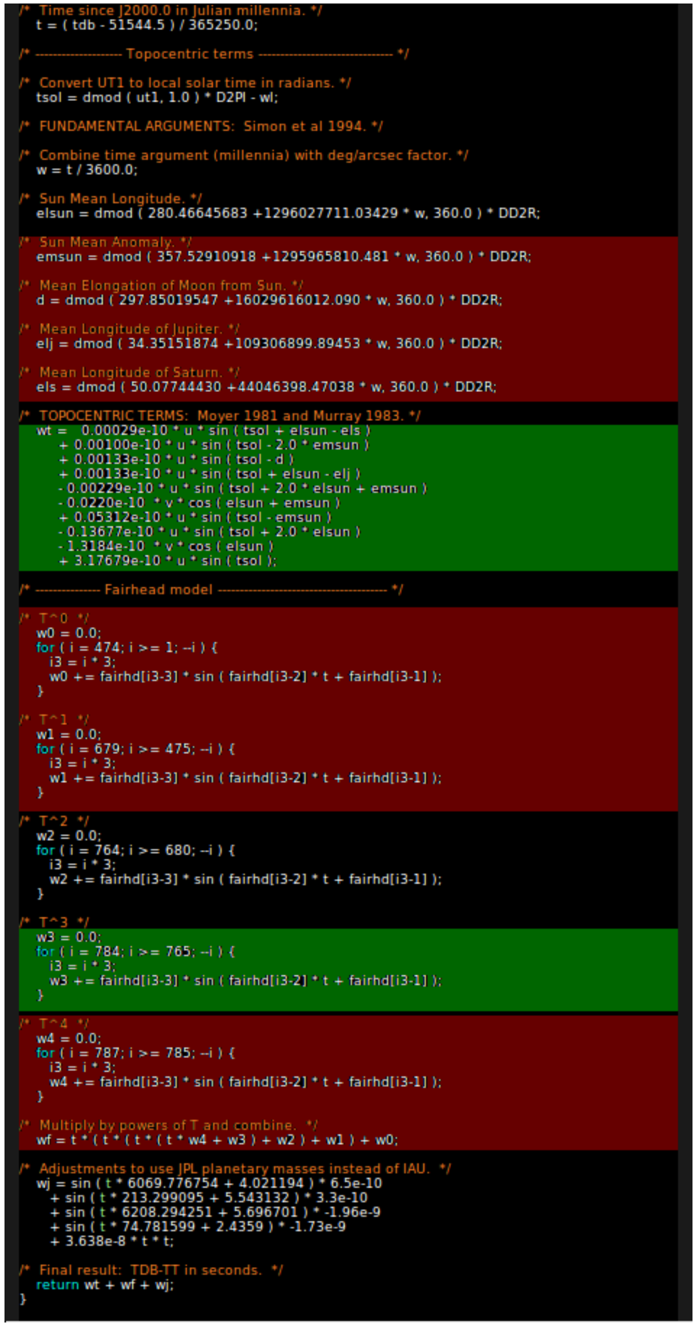


```
/*  Time since J2000.0 in Julian millennia. */
   t = ( tdb - 51544.5 ) / 365250.0;

/* ------------------ Topocentric terms ---------------------------- */

/*  Convert UT1 to local solar time in radians. */
   tsol = dmod ( ut1, 1.0 ) * D2PI - wl;

/*  FUNDAMENTAL ARGUMENTS:  Simon et al 1994. */

/*  Combine time argument (millennia) with deg/arcsec factor. */
   w = t / 3600.0;

/*  Sun Mean Longitude. */
   elsun = dmod ( 280.46645683 +1296027711.03429 * w, 360.0 ) * DD2R;

/*  Sun Mean Anomaly. */
   emsun = dmod ( 357.52910918 +1295965810.481 * w, 360.0 ) * DD2R;

/*  Mean Elongation of Moon from Sun. */
   d = dmod ( 297.85019547 +16029616012.090 * w, 360.0 ) * DD2R;

/*  Mean Longitude of Jupiter. */
   elj = dmod ( 34.35151874 +109306899.89453 * w, 360.0 ) * DD2R;

/*  Mean Longitude of Saturn. */
   els = dmod ( 50.07744430 +44046398.47038 * w, 360.0 ) * DD2R;

/*  TOPOCENTRIC TERMS:  Moyer 1981 and Murray 1983. */
   wt =   0.00029e-10 * u * sin ( tsol + elsun - els )
      + 0.00100e-10 * u * sin ( tsol - 2.0 * emsun )
      + 0.00133e-10 * u * sin ( tsol - d )
      + 0.00133e-10 * u * sin ( tsol + elsun - elj )
      - 0.00229e-10 * u * sin ( tsol + 2.0 * elsun + emsun )
      - 0.0220e-10  * v * cos ( elsun + emsun )
      + 0.05312e-10 * u * sin ( tsol - emsun )
      - 0.13677e-10 * u * sin ( tsol + 2.0 * elsun )
      - 1.3184e-10  * v * cos ( elsun )
      + 3.17679e-10 * u * sin ( tsol );

/* ------------- Fairhead model ------------------------------------ */

/*  T^0  */
   w0 = 0.0;
   for ( i = 474; i >= 1; --i ) {
     i3 = i * 3;
     w0 += fairhd[i3-3] * sin ( fairhd[i3-2] * t + fairhd[i3-1] );
   }

/*  T^1  */
   w1 = 0.0;
   for ( i = 679; i >= 475; --i ) {
     i3 = i * 3;
     w1 += fairhd[i3-3] * sin ( fairhd[i3-2] * t + fairhd[i3-1] );
   }

/*  T^2  */
   w2 = 0.0;
   for ( i = 764; i >= 680; --i ) {
     i3 = i * 3;
     w2 += fairhd[i3-3] * sin ( fairhd[i3-2] * t + fairhd[i3-1] );
   }

/*  T^3  */
   w3 = 0.0;
   for ( i = 784; i >= 765; --i ) {
     i3 = i * 3;
     w3 += fairhd[i3-3] * sin ( fairhd[i3-2] * t + fairhd[i3-1] );
   }

/*  T^4  */
   w4 = 0.0;
   for ( i = 787; i >= 785; --i ) {
     i3 = i * 3;
     w4 += fairhd[i3-3] * sin ( fairhd[i3-2] * t + fairhd[i3-1] );
   }

/*  Multiply by powers of T and combine.  */
   wf = t * ( t * ( t * ( t * w4 + w3 ) + w2 ) + w1 ) + w0;

/*  Adjustments to use JPL planetary masses instead of IAU.  */
   wj = sin ( t * 6069.776754 + 4.021194 ) * 6.5e-10
     + sin ( t * 213.299095 + 5.543132 ) * 3.3e-10
     + sin ( t * 6208.294251 + 5.696701 ) * -1.96e-9
     + sin ( t * 74.781599 + 2.4359 ) * -1.73e-9
     + 3.638e-8 * t * t;

/*  Final result:  TDB-TT in seconds.  */
   return wt + wf + wj;
}
```

Critical Section

- **A source code can have multiple critical sections**
  - And they can overlap (not shown in this example)
  - Just like having multiple bathrooms
- **Common misconception**: A critical section corresponds to a variable
- This is incorrect: a critical section corresponds to section(s) of code (i.e., in the text segment)

- When people say “we need to protect variable x from race conditions” it really means “we need to put all the **code** that updates variable x into a critical section”
  - If software design is good, this shouldn’t be too much work



```
/*  Time since J2000.0 in Julian millennia. */
   t = ( tdb - 51544.5 ) / 365250.0;

/* ------------------ Topocentric terms ---------------------------- */

/*  Convert UT1 to local solar time in radians. */
   tsol = dmod ( ut1, 1.0 ) * D2PI - wl;

/*  FUNDAMENTAL ARGUMENTS:  Simon et al 1994. */

/*  Combine time argument (millennia) with deg/arcsec factor. */
   w = t / 3600.0;

/*  Sun Mean Longitude. */
   elsun = dmod ( 280.46645683 +1296027711.03429 * w, 360.0 ) * DD2R;

/*  Sun Mean Anomaly. */
   emsun = dmod ( 357.52910918 +1295965810.481 * w, 360.0 ) * DD2R;

/*  Mean Elongation of Moon from Sun. */
   d = dmod ( 297.85019547 +16029616012.090 * w, 360.0 ) * DD2R;

/*  Mean Longitude of Jupiter. */
   elj = dmod ( 34.35151874 +109306899.89453 * w, 360.0 ) * DD2R;

/*  Mean Longitude of Saturn. */
   els = dmod ( 50.07744430 +44046398.47038 * w, 360.0 ) * DD2R;

/*  TOPOCENTRIC TERMS:  Moyer 1981 and Murray 1983. */
   wt =   0.00029e-10 * u * sin ( tsol + elsun - els )
      + 0.00100e-10 * u * sin ( tsol - 2.0 * emsun )
      + 0.00133e-10 * u * sin ( tsol - d )
      + 0.00133e-10 * u * sin ( tsol + elsun - elj )
      - 0.00229e-10 * u * sin ( tsol + 2.0 * elsun + emsun )
      - 0.0220e-10  * v * cos ( elsun + emsun )
      + 0.05312e-10 * u * sin ( tsol - emsun )
      - 0.13677e-10 * u * sin ( tsol + 2.0 * elsun )
      - 1.3184e-10  * v * cos ( elsun )
      + 3.17679e-10 * u * sin ( tsol );

/* -------------- Fairhead model ------------------------------------- */

/*  T^0  */
   w0 = 0.0;
   for ( i = 474; i >= 1; --i ) {
     i3 = i * 3;
     w0 += fairhd[i3-3] * sin ( fairhd[i3-2] * t + fairhd[i3-1] );
   }

/*  T^1  */
   w1 = 0.0;
   for ( i = 679; i >= 475; --i ) {
     i3 = i * 3;
     w1 += fairhd[i3-3] * sin ( fairhd[i3-2] * t + fairhd[i3-1] );
   }

/*  T^2  */
   w2 = 0.0;
   for ( i = 764; i >= 680; --i ) {
     i3 = i * 3;
     w2 += fairhd[i3-3] * sin ( fairhd[i3-2] * t + fairhd[i3-1] );
   }

/*  T^3  */
   w3 = 0.0;
   for ( i = 784; i >= 765; --i ) {
     i3 = i * 3;
     w3 += fairhd[i3-3] * sin ( fairhd[i3-2] * t + fairhd[i3-1] );
   }

/*  T^4  */
   w4 = 0.0;
   for ( i = 787; i >= 785; --i ) {
     i3 = i * 3;
     w4 += fairhd[i3-3] * sin ( fairhd[i3-2] * t + fairhd[i3-1] );
   }

/*  Multiply by powers of T and combine.  */
   wf = t * ( t * ( t * ( t * w4 + w3 ) + w2 ) + w1 ) + w0;

/*  Adjustments to use JPL planetary masses instead of IAU.  */
   wj = sin ( t * 6069.776754 + 4.021194 ) * 6.5e-10
     + sin ( t * 213.299095 + 5.543132 ) * 3.3e-10
     + sin ( t * 6208.294251 + 5.696701 ) * -1.96e-9
     + sin ( t * 74.781599 + 2.4359 ) * -1.73e-9
     + 3.638e-8 * t * t;

/*  Final result:  TDB-TT in seconds.  */
   return wt + wf + wj;
}
```

Example

- Consider this code fragment, where threads can call functions **`f()`** and **`g()`** at any time

```
int a = 0;
int b = 2;
int x = 100;

void f() {
  for (int i=0; i < 1000; i++) {
    a++;
  }
}

void g() {
  b++;
  x--;
}
```

Example

- Consider this code fragment, where threads can call functions `f()` and `g()` at any time

```
int a = 0;
int b = 2;
int x = 100;

void f() {
  for (int i=0; i < 1000; i++) {
    a++;
  }
}

void g() {
  b++;
  x--;
}
```

- One brute-force solution is to put everything into a critical section
- Bad idea: no concurrency anymore!!

Example

- Consider this code fragment, where threads can call functions **`f()`** and **`g()`** at any time

```
int a = 0;
int b = 2;
int x = 100;

void f() {
  for (int i=0; i < 1000; i++) {
    a++;
  }
}

void g() {
  b++;
  x--;
}
```

- Some of the code in the critical section is not "critical" because it's about variables local to a thread, so we can make the critical section smaller, which is better for concurrency

Example

- Consider this code fragment, where threads can call functions **`f()`** and **`g()`** at any time

```
int a = 0;
int b = 2;
int x = 100;

void f() {
  for (int i=0; i < 1000; i++) {
    a++;
  }
}

void g() {
  b++;
  x--;
}
```

- We should also use different critical sections for lines of codes that update different variables
- This maximizes concurrency

Critical Section Definition

- Formally there are three requirements to execute critical sections:
  - Mutual Exclusion: If a thread is executing in the critical section, then no other thread can be executing in it
  - Progress: If a thread wants to enter the critical section, it will enter it at some point in the future
  - Bounded Waiting: Once a thread has declared intent to enter the critical section, there should be a bound on the number of threads that can enter the critical section before it
- Note that there is no assumption regarding the elapsed time spent by each involved thread in the critical section
- These are theoretical conditions: Programming Languages, OSes, Hardware are in charge of the "implementation details"

Critical Section Duration

- You should always try to make critical sections as short as possible
  - Not in number of lines of code, but **in time** to run these lines
- Long critical sections: only one thread can do work for a while, so we have reduced opportunities for concurrent execution
  - And thus reduced interactivity and/or performance
- Extreme situation: put the whole code in a critical critical section
  - Guaranteed to have no race condition, but only one thread can run at a time, so why use threads in the first place: No concurrency at all
- Instead, one should use possibly many very short critical sections (each protected by a different lock), so that many threads can do useful work simultaneously

The Kernel and Race Conditions

- Consider a process that places a system call
- It begins running kernel code
- And then a context switch happens!

- Modern kernels allow the above (they're called preemptive kernels)
- But that means we can have race conditions in the kernel!!
  - e.g., the list of open files is some data structure with a `size` variable. Say that right now 10 files are opened. One thread is opening a file, and is context-switched out right before storing value 11 into `size`. Another thread closes a file and updates `size` to 9. The first thread is context switched back in and sets `size` to 11. We have a lost update: There are 10 files open, but the kernel thinks there are 11! Down the line this will cause a Linux kernel panic, a Windows blue screen of death, etc.
- Preemptive kernels must deal with race conditions just like any other piece of code, using critical sections
- You can search for for "Google Is Uncovering Hundreds Of Race Conditions Within The Linux Kernel" …

Critical Section Mechanisms

- What we need to are **`enter_critical_section()`** and **`leave_critical_section()`** mechanisms, to lock and unlock access to the critical section
- There are some pure-software solutions (mostly historical)
  - They can be very complicated, and not guaranteed to work on modern architectures
  - See “Aside: Dekker’s and Peterson’s Algorithms” for details (OSTEP 28.5)
- One option could be to disable interrupts during critical sections (then there can be no context switches)
  - Very dangerous (what if the user “forgets” to re-enable them??)
  - Interrupts are useful for other things, not just context switches
  - Perhaps ok if done by the kernel occasionally
- The current solution: our CPUs provide atomic instructions
  - Instructions that can never be interrupted
  - Once a thread begins executing the instruction, it is guaranteed to finish it right away without the CPU doing anything else

Locks

- Without going into details, with atomic instructions it is possible to implement a lock data type
- A lock can be in one of two states: taken or not taken
- There are two fundamental operations:
  - `acquire()` or `lock()`: **atomically** acquires (i.e., puts it in the "taken state") the lock if it's not taken, otherwise fail
  - `release()` or `unlock()`: releases the lock (i.e., puts it in the "not taken" state)
- Real-life metaphor: a bathroom key on a hook in a coffee shop
  - Either it's taken (and somebody is using the bathroom)
  - Or it's not taken

Let’s go back to this example

- Let’s rewrite it with locks

```
int a = 0;
int b = 2;
int x = 100;

void f() {
  for (int i=0; i < 1000; i++) {
    a++;
  }
}

void g() {
  b++;
  x--;
}
```

- We have three critical sections, so let’s use 3 locks

Let’s go back to this example

- Let’s rewrite it with locks

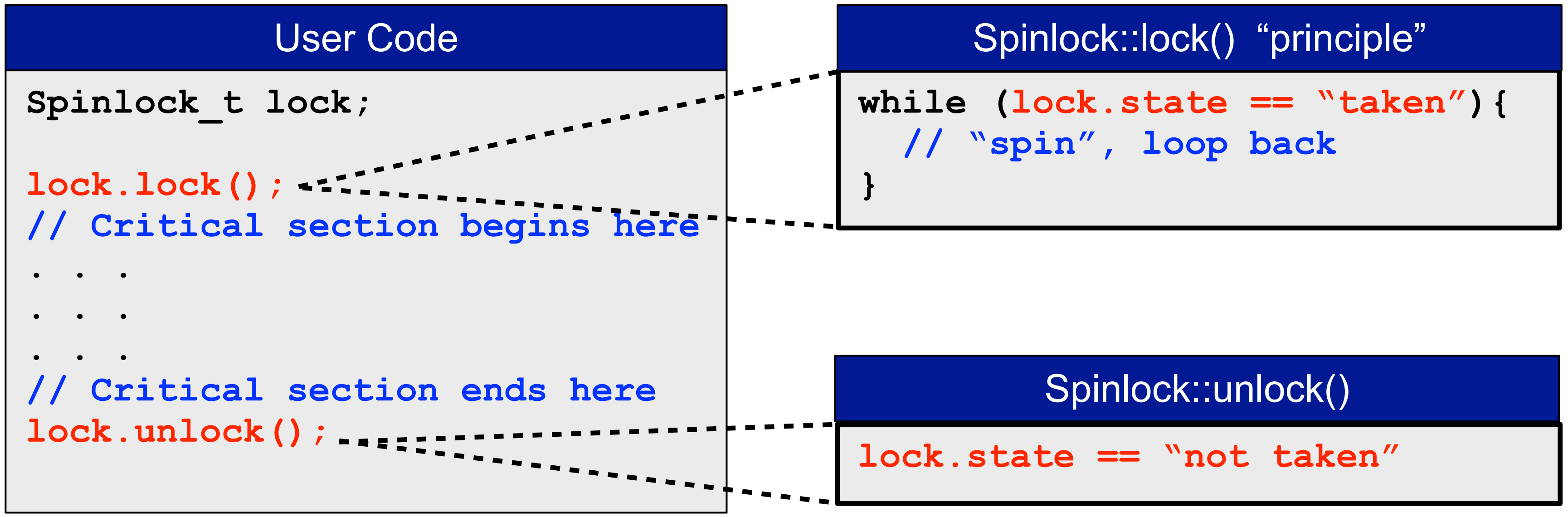
```
int a = 0;
int b = 2;
int x = 100;
lock_t lock_a, lock_b, lock_x;

void f() {
  for (int i=0; i < 1000; i++) {
    lock_a.lock();
    a++;
    lock_a.unlock();
  }
}
```

```
void g() {
  lock_b.lock();
  b++;
  lock_b.unlock();
  lock_x.lock();
  x--;
  lock_x.unlock();
}
```

Lock implementations

- Turns out there are several ways to implement locks internally
- The two main kinds are:
  - Spinlocks
  - Blocking locks
- The semantics and API are the same, and so you can use either kind in your program and you'll prevent race conditions
  - They are all "locks"
- But different kinds of lock lead to different performance, CPU cycle wastes, and overhead
- Let's look at easy-to-understand pseudo-code for the spinlock and the blocking lock implementation
  - But the real implementation is different because it uses atomic instructions, which the pseudo-code doesn't show at all

Spinlock



User Code

```
Spinlock_t lock;

lock.lock();
// Critical section begins here
. . .
. . .
. . .
// Critical section ends here
lock.unlock();
```

Spinlock::lock() “principle”

```
while (lock.state == “taken”){
  // “spin”, loop back
}
```

Spinlock::unlock()

```
lock.state == “not taken”
```

- The good:
  - A thread will enter the critical section as soon as another has left it
  - Very little overhead (the OS is not involved)
- The bad:
  - If the critical section is long and a thread is already in it, a thread wanting to get in will spin for a long time
  - This wastes CPU cycles, power, and generates heat
  - Think of the real-life coffeeshop metaphor....

Blocking Lock

- If the critical section is long (in terms of the time it takes for a thread to execute it), spinlocks are probably a bad idea
  - “The bad” from the previous slide
- If the critical section is long, then a thread shouldn’t be spinning Instead, it should “sleep” or be “blocked”
- The main idea:
  - If the lock cannot be acquired, then ask the OS to put me to sleep (to the WAITING / BLOCKED state, not in the Ready Queue anymore)
  - Whenever the lock is released, then the OS will wake me up (to the READY state, back into the Ready Queue, so that later I can try again)
- Real-life metaphor: if the bathroom key is taken, ask the barista to come “wake you up” at your table whenever the key is ready
- Let’s see pseudo-code...

Blocking Lock

User Code

```
BlockingLock_t lock;

lock.lock();
// Critical section begins here
. . .
. . .
. . .
// Critical section ends here
lock.unlock();
```

BlockingLock::lock() "principle"

```
while (lock.state == "taken"){
  // Do a system call to ask the OS
  // to put me to sleep
  // At some point I will be awakened,
  // put back in the ready queue,
  // scheduled, and loop back to
  // try again.
}
```

BlockingLock::unlock()

```
lock.state == "not taken"
```

- The good: No wasted CPU cycles
  - Which is great if the wait is long
- The bad: High overhead due to the system call
  - Which is bad if the wait is short
  - Again think of the real-life coffeeshop metaphor

Spinlocks and Blocking Locks

- In the same program, for different critical sections you can use different kinds of locks at will

Critical Sections

```
SpinLock     s_lock;
BlockingLock b_lock;

s_lock.lock();
// Short critical section begins here
...
// Short critical section ends here
s_lock.unlock();


...

b_lock.lock();
// Long critical section begins here
...
// Long critical section ends here
b_lock.unlock();
```

Fixing our Java Example

- Java provides locks in `java.util.concurrent.locks.ReentrantLock`
  - This is a “smart” lock, which I won’t say much about
- We can thus create a critical section as:

Fixing our Java program

```
ReentrantLock lock = new ReentrantLock();

public void increment() {
       this.lock.lock();
        this.counter += 1;
        this.lock.unlock()
}
```

- Let’s look at and run the code in CounterTestV2.java

Java `synchronized`

- A common bug is to forget to call **`unlock()`**
- Java provides a convenient **`synchronized`** keyword

Using Java’s synchronized keyword

```
public synchronized void increment() {
    this.counter += 1;
}
```

- Let’s look at and run the code in CounterTestV3.java

Locks in OSes

- All OSes provide spinlocks and blocking locks, in one shape or another

- Many provide smart adaptive locks
  - Will spin for a short while, and then will block
  - A “perhaps I’ll be lucky” approach
  - Totally fits the real-life bathroom key metaphor for some of us

- There are other kinds of locks with different properties/behaviors (e.g., reader-writer locks)

Race Conditions occur in many Shapes/Forms

- Let’s look at a recent LinkedIn post from a UH alumni [link]

Main Takeaways

- A common problem in concurrent programs is race conditions
  - Because high-level language operations are non-atomic, and consists of sequences of several machine-level instructions
  - An example of a bug caused by race conditions is a lost update
- The solution: create critical sections
- The main mechanism: locks
  - Public bathroom analogy
- Creating more critical sections, while preserving correctness, increases concurrency and thus performance
- Two kinds of locks: Spinlocks and Blocking locks
- Java: the synchronized keyword

Conclusion

- Synchronization is a critical and difficult topic
  - Both in practice and in theory
  - We only scratched the surface in these lecture notes
  - There are many other topics (Condition variables, Semaphores)
- **Bottom line:** take ICS 432 if you want to find out more and gain hands-on experience
  - Both with writing code and with looking at code and figuring out what’s wrong with it
- Onward to Deadlocks...