

Considering Concurrency

Concurrency is great...

```
int a=1, b=2;
main() {
    createThread(fn1, 4);
    createThread(fn2, 5);
    thread_join(all);
}
```

```
fn1(int arg1){
    if(a) b++;
}
```

```
fn2(int arg1){
    a=arg1;
}
```

What are the values of a and b after execution?

- A. a=1, b=2
- B. a=1, b=3
- C. a=5, b=2
- D. a=5, b=3

... but can be problematic

```
int a=1, b=2;
main() {
    createThread(fn1, 4);
    createThread(fn2, 5);
    thread_join(all);
}

fn1(int arg1){
    if(a) b++;
}

fn2(int arg1){
    a=0;
}
```

What are the values of a and b after execution?

- A. a=0, b=2
- B. a=0, b=3
- C. a=1, b=2
- D. a=1, b=3

Question

```
int flag1=0, flag2=0;
```

```
int main(){  
    tid id=thread_create(p1, NULL);  
    p2(); thread_join(id);  
}
```

```
void p1 (void *ignored){  
    flag1=1;  
    if(!flag2){  
        critical_section_1();  
    }  
}
```

```
void p2(void * ignored){  
    flag2=1;  
    if(!flag1){  
        critical_section_2();  
    }  
}
```

Can both critical sections execute during a single execution of the code?

- A. Yes
- B. No

Atomicity

- Required to reason about multi-threaded code without considering all interleavings
- Requires mutual exclusion
- Locks provide that solution
- Looked at lock implementation
 - Requires waiting
 - Requires hardware support
- Use software abstractions
 - Semaphores
 - Monitors (lock+condition variables)

Cost of Concurrency

Tradeoff and Problems: Difficult to Get Right

- Ensure safety
- Ensure liveness
- No race conditions
- No starvation
- No priority inversion
- No deadlock

In Addition... the Cost of Parallelization

```
for (k = 0; k < n; k++)  
    a[k] = b[k]*c[k] + d[k]*e[k];
```

How would you parallelize this?

How many threads?

Writing Multi-threaded (User-Level) Code

Designing Multithreaded Programs

- Building a shared object class (or pseudo-class) involves familiar steps:
 - Decompose the problem into objects
 - For each object:
 - define a clear interface
 - implement methods that manipulate the state appropriately
- Then add the new steps...

Managing Locks

- Add a lock as a member variable for each object in the class to enforce mutual exclusion on the object's shared state
- Acquire a lock at the start of each public method
- Release the lock at the end of each public method
 - You will be tempted to acquire/release a lock midway through a method.
 - RESIST!

Identifying Condition Variables

- Ask yourself: when should the thread executing this function wait?
- Map each opportunity for waiting to a condition variable
 - full and empty in producers/consumers
- You may be able to manage with fewer condition variables
 - such as somethingChanged
 - but you must call broadcast() and not signal() (why?)

Waiting Using Condition Variables

- Every call to `Condition::Wait()` should be enclosed in a loop
- Loop tests the appropriate resource
- When `Condition::Wait()` is called, **all invariants must hold**
 - Remember, you are releasing the lock!

The Six Commandments

- Thou shalt always do things the same way
 - Habit allows you to focus on the core problem
 - Easier to review, maintain, and debug your code
- Thou shalt always synchronize with locks and condition variables
 - Condition variables and locks make code clearer

The Six Commandments

- Thou shalt always acquire the lock at the beginning of a function and release it at the end
 - Put a chunk of code that requires a lock into its own function
- Thou shalt always hold lock when operating on a condition variable
 - Condition variables are useless without shared state
 - Shared state should only be accessed using a lock

The Six Commandments

- Thou shalt always wait in a while loop
 - `while ( )` works every time `if ( )` does
 - Makes signals hints
 - Protects against spurious wake ups
- (Almost) Never `sleep()`
 - Use `sleep ( )` only if an action should occur at a specific real time
 - Never use `sleep ( )` to wait for an event

Thread Coding Standards

The Six Commandments

- Thou shalt always do things the same way
- Thou shalt always synchronize with locks and condition variables
- Thou shalt always acquire the lock at the beginning of a function and release it at the end
- Thou shalt always hold lock when operating on a condition variable
- Thou shalt always wait in a while loop
- (Almost) Never sleep()

Why Thread Coding Standards?

- History has tested this approach
- If you follow these commandments, you will find it easier to write correct code.
- In this class, you must use them or lose points.
- We highly recommend that you continue to do so after this class

But...

- After this class, if you can come up with something better, please use it!
- BUT...
 - Lots of really smart people have thought really hard about this already, so a day or two of thought is unlikely to change the best practice
 - The consequences of getting code wrong can be atrocious
 - People who are confident about their abilities tend to perform **worse**. If you think you are a Threading and Concurrency Ninja and truly understand, then you may wish to re-evaluate...
 - Dunning-Kruger effect

In this class...

- Six commandments
 - For application code
- Coarse-grained locking
 - In general
- Order your resources
 - Semaphores
 - Locks in the kernel