Concurrency

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Why Concurrency?

During the past two weeks, we have explored that processes and threads give the perception of continuous execution.

This week, we will explore **why** we care about concurrency and go into the technical about the various forms of “concurrency”.

Lots of Concurrency In Use

Servers: Many connections handled simultaneously

Parallel Algorithms: Achieve better performance

UI Threads: Achieve fast user-responsiveness

Blocking Operation: Networking/Disk operations happening in the background, in parallel with other tasks

Concurrency

There are six similar, but varied terms used:

1. Sequential execution
2. Concurrent execution
3. Parallel execution
4. Concurrent but not parallel
5. Parallel but not concurrent
6. Parallel and concurrent

One Thread

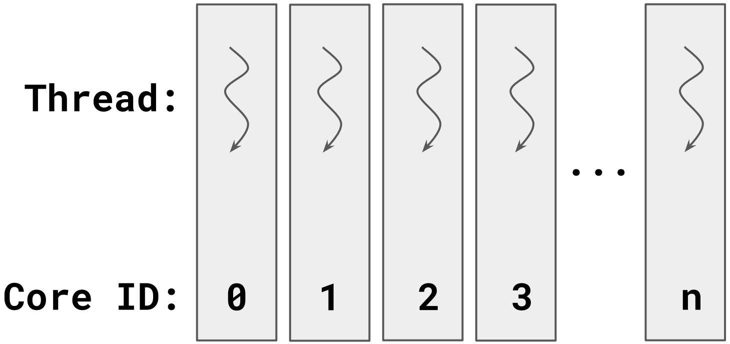
A **single thread** is a single execution sequence:

- Intuitive, familiar, easy to understand.
- Scheduled independently of everything else on the system.
- A single threaded process is (generally) isolated from other events in the system by the operating system.

Many Threads

A **multi-threaded** program provides an abstraction for the programmer:

Abstraction:



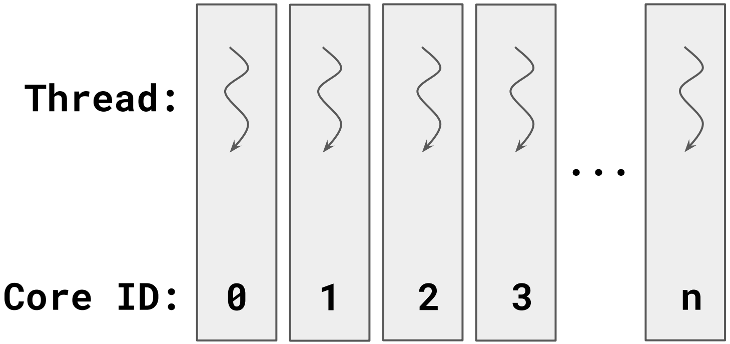
★ Every thread is running on its own CPU, continuously.

Reality

Many Threads

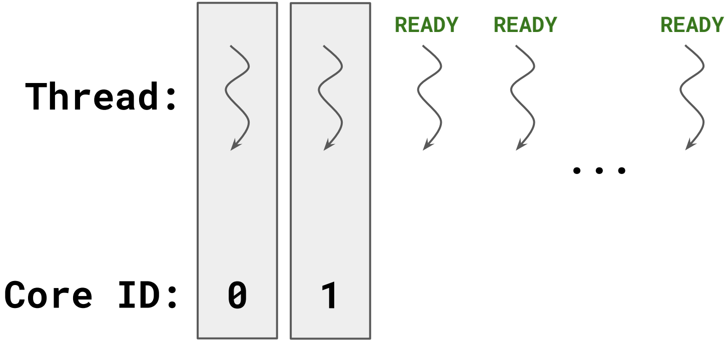
A **multi-threaded** program provides an abstraction for the programmer:

Abstraction:



★ Every thread is running on its own CPU, continuously.

Reality



★ Limited CPU, threads are sitting in the **READY** state.

Many Threads

★ In the real system, there's so many different combination of possible execution interactions:

Abstraction:

```
...
x = x + 1;
y = y + x;
z = x + (5 * y);
...
```

Many Threads

★ In the real system, there's so many different combination of possible execution interactions:

Abstraction:

```
...
x = x + 1;
y = y + x;
z = x + (5 * y);
...
```

Possible Executions:

```
...
x = x + 1;
y = y + x;
z = x + (5 * y);
...
```

```
...
x = x + 1;
```

Timer interrupt results in scheduler suspending thread; other threads run until re-scheduled.

```
y = y + x;
z = x + (5 * y);
...
```

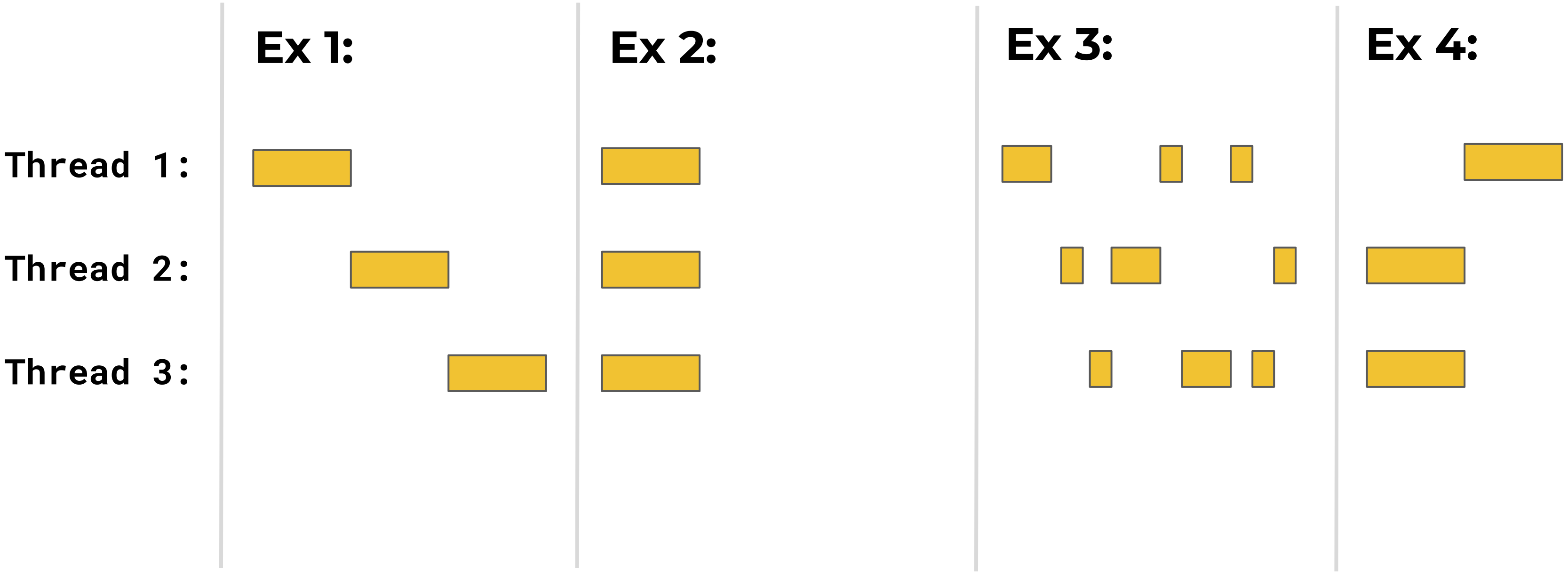
```
...
x = x + 1;
y = y + x;
```

Timer interrupt results in scheduler suspending thread; other threads run until re-scheduled.

```
z = x + (5 * y);
...
```

Many Threads

★ The possibilities grow exponentially larger as we consider the interleaving of threads:




ALMA

Thread Example

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

```
#define NTHREADS 10
thread_t threads[NTHREADS];
main() {
  for (i = 0; i < NTHREADS; i++) {
    thread_create(&threads[i], &go, i);
  }

  for (i = 0; i < NTHREADS; i++) {
    exitValue = thread_join(threads[i]);
    printf("Thread %d returned with %ld\n", i, exitValue);
  }

  printf("Main thread done.\n");
}

void go (int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
}
```

```
#define NTHREADS 10
thread_t threads[NTHREADS];
main() {
  for (i = 0; i < NTHREADS; i++) {
    thread_create(&threads[i], &go, i);
  }

  for (i = 0; i < NTHREADS; i++) {
    exitValue = thread_join(threads[i]);
    printf("Thread %d returned with %ld\n", i, exitValue);
  }

  printf("Main thread done.\n");
}

void go (int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
}
```

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

```c
#define NTHREADS 10
thread_t threads[NTHREADS];
main() {
  for (i = 0; i < NTHREADS; i++) {
    thread_create(&threads[i], &go, i);
  }

  for (i = 0; i < NTHREADS; i++) {
    exitValue = thread_join(threads[i]);
    printf("Thread %d returned with %ld\n", i, exitValue);
  }

  printf("Main thread done.\n");
}

void go (int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
}
```

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

Q: The “Thread X returned” printed in order. Does that always happen?

```c
#define NTHREADS 10
thread_t threads[NTHREADS];
main() {
  for (i = 0; i < NTHREADS; i++) {
    thread_create(&threads[i], &go, i);
  }

  for (i = 0; i < NTHREADS; i++) {
    exitValue = thread_join(threads[i]);
    printf("Thread %d returned with %ld\n", i, exitValue);
  }

  printf("Main thread done.\n");
}

void go (int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
}
```

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

Q: What is the **maximum** number of threads that will return before "Hello" from thread 5?

```
#define NTHREADS 10
thread_t threads[NTHREADS];
main() {
  for (i = 0; i < NTHREADS; i++) {
    thread_create(&threads[i], &go, i);
  }

  for (i = 0; i < NTHREADS; i++) {
    exitValue = thread_join(threads[i]);
    printf("Thread %d returned with %ld\n", i, exitValue);
  }

  printf("Main thread done.\n");
}

void go (int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
}
```

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

Q: What is the **minimum** number of threads that will return before "Hello" from thread 5?

```
#define NTHREADS 10
thread_t threads[NTHREADS];
main() {
  for (i = 0; i < NTHREADS; i++) {
    thread_create(&threads[i], &go, i);
  }

  for (i = 0; i < NTHREADS; i++) {
    exitValue = thread_join(threads[i]);
    printf("Thread %d returned with %ld\n", i, exitValue);
  }

  printf("Main thread done.\n");
}

void go (int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
}
```

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

Q: Why are none of the print statements interrupted mid-string?

Synchronization

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Why Synchronization?

How do we allow multiple threads, that may run any in random order for any length of time, do something useful together?

Can We Crash?

★ Assumption: **`q`** will always be non-NULL if **`initResource()`** has returned.

Thread 1:

```
...
r = initResource();
r_init = true;
...
```

Thread 2:

```
...
while (!r_init) { }

q = fetchResult(r);
if (!q) {
   /* Program crashes */
}
```

Can We Crash?

★ Compiler optimizations may see no relationship between r and r_init, reordering thread1:

Thread 1:

```
...
r_init = true;
r = initResource();
...
```

Thread 2:

```
...
while (!r_init) { }

q = fetchResult(r);
if (!q) {
   /* Program crashes */
}
```

Can We Crash?

★ Not just compilers -- some hardware operations may pre-fetch values that may be changed by another core.

IRL Example:

You and your roommate are working together to keep the supply of milk in the fridge:

	Person 1:	Person 2:
12:30pm	`Look at fridge. No milk. :(`	
12:35pm	`Leave for store.`	
12:40pm	`Arrive at store.`	
12:45pm	`Buy milk.`	
12:50pm	`Arrive home, place milk in fridge.`	
12:55pm		
1:00pm		

IRL Example:

You and your roommate are working together to keep the supply of milk in the fridge:

	Person 1:	**Person 2:**
12:30pm	`Look at fridge. No milk. :(`	
12:35pm	`Leave for store.`	
12:40pm	`Arrive at store.`	`Look at fridge. No milk. :(`
12:45pm	`Buy milk.`	`Leave for store.`
12:50pm	`Arrive home, place milk in fridge.`	`Arrive at store.`
12:55pm		`Buy milk.`
1:00pm		`Arrive home, place milk in...`
		`...how did the milk get here?!?`

Definitions

- **Race Condition**:
  - A race condition occurs when the output of a concurrent program depends on the order of operation between threads.

Definitions

- **Mutual Exclusion**:
  - Occurs when a single threads is running a specific task or code.

- **Critical Section**
  - A piece of code that only one thread can execute at a time.

Definitions

- **Lock:**
  - A shared resource that allows a single thread to advance only once it “holds” the lock.
  - All other threads will wait until the lock is “released” before advancing.

Correctness of Synchronization

- **Liveness:**
  - If a thread is requesting access to a critical section and no one is in it, the requesting thread must be able to advance to the critical section.

- **Safety**:
  - Only one thread can enter the critical section at any time.

Synchronization Solutions

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

IRL Example:

You and your roommate are working together to keep the supply of milk in the fridge:

	Person 1:	Person 2:
12:30pm	`Look at fridge. No milk. :(`	
12:35pm	`Leave for store.`	
12:40pm	`Arrive at store.`	`Look at fridge. No milk. :(`
12:45pm	`Buy milk.`	`Leave for store.`
12:50pm	`Arrive home, place milk in fridge.`	`Arrive at store.`
12:55pm		`Buy milk.`
1:00pm		`Arrive home, place milk in...`
		`...how did the milk get here?!?`

Potential Solution #1

Both you and your roommate use the same logic:

Person 1:

```
if (!note) {
  if (!milk) {
    leave note;
    buy milk;
    remove note;
  }
}
```

Person 2:

```
if (!note) {
  if (!milk) {
    leave note;
    buy milk;
    remove note;
  }
}
```

Potential Solution #1

Both you and your roommate use the same logic:

Person 1:

```
if (!note) {

  if (!milk) {

    leave note;

    buy milk;
    remove note;
  }
}
```

Person 2:

```
if (!note) {

  if (!milk) {

    leave note;
    buy milk;
    remove note;
  }
}
```

Potential Solution #1

Both you and your roommate use the same logic:

Person 1:

```
if (!note) {  //true, no note found
  if (!milk) {  //true, no milk
    leave note;

    buy milk;  //in critical section
    remove note;
  }
}
```

Person 2:

```
if (!note) {  //true, no note found

  if (!milk) {  //true, no milk

    leave note;
    buy milk;  //in critical section
    remove note;
  }
}
```

Potential Solution #1

Both you and your roommate use the same logic:

Person 1:

```
if (!note) {  //true, no note found

  if (!milk) {  //true, no milk

    leave note;

    buy milk;  //in critical section
    remove note;
  }
}
```

Person 2:

```
if (!note) {  //true, no note found

  if (!milk) {  //true, no milk

    leave note;
    buy milk;  //in critical section
    remove note;
  }
}
```

Safety Violation: Two threads are in the critical section *(you and your roommate are both buying milk)*.

Potential Solution #2

Let's update it so we each leave a note, checking for each other's:

Person 1:

```
leave noteA;
if (!noteB) {
  if (!milk) {
    buy milk;
  }
}
remote noteA;
```

Person 2:

```
leave noteB;
if (!noteA) {
  if (!milk) {
    buy milk;
  }
}
remote noteB;
```

Person 1:

```
leave noteA;

if (!noteB) {
    if (!milk) {
        buy milk;
    }
}

remote noteA;
```

Person 2:

```
leave noteB;

if (!noteA) {
    if (!milk) {
        buy milk;
    }
}
remote noteB;
```

Person 1:

```
leave noteA;

if (!noteB) {  //false, noteB found
    ~~if (!milk) {~~
        ~~buy milk;~~
    ~~}~~
}

remote noteA;
```

Person 2:

```
leave noteB;

if (!noteA) {  //false, noteA found
    ~~if (!milk) {~~
        ~~buy milk;~~
    ~~}~~
}
remote noteB;
```

Person 1:

```
leave noteA;

if (!noteB) {   //false, noteB found
    if (!milk) {
        buy milk;
    }
}

remote noteA;
```

Person 2:

```
leave noteB;

if (!noteA) {   //false, noteA found
    if (!milk) {
        buy milk;
    }
}
remote noteB;
```

Liveness Violation: A thread requested access to the critical section, but failed to get it. (No one bought milk!)

Potential Solution #3

Person 1:

```
leave noteA;
while (noteB) { }

if (!milk) {
  buy milk;
}

remote noteA;
```

Person 2:

```
leave noteB;

if (!noteA) {
  if (!milk) {
    buy milk;
  }
}

remote noteB;
```

Person 1:

```
leave noteA;
while (noteB) { }

if (!milk) {
  buy milk;
}

remote noteA;
```

Person 2:

```
leave noteB;

if (!noteA) {
  if (!milk) {
    buy milk;
  }
}

remote noteB;
```

Person 1:

```
leave noteA;

while (noteB) { }








if (!milk) {
  buy milk;
}

remote noteA;
```

Person 2:

```
leave noteB;

if (!noteA) {
  if (!milk) {
    buy milk;
  }
}

remote noteB;
```

Person 1:

```
leave noteA;
while (noteB) { }




if (!milk) {
  buy milk;
}

remote noteA;
```

Person 2:

```
leave noteB;

if (!noteA) {
  if (!milk) {
    buy milk;
  }
}

remote noteB;
```

Takeaways

★ **Solution is Complex:**
  - Obvious solution has bugs.

★ **We Assumed Code Wasn't Reordered**
  - Optimization may reorder our code, making reasoning even more difficult than it is already!

★ **Needs Generalization**
  - Our solution assumed only 2 threads, how can we generalize this further? (See: Peterson's Solution)

Locks

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Definitions

- **Lock:**
  - A shared resource that allows a single thread to advance only once it “holds” the lock.
  - All other threads will wait until the lock is “released” before advancing.

Operations

- ★ **lock::acquire**
  - Waits until the lock is available, then takes the lock.
- ★ **lock::release**
  - Release the lock, allowing someone waiting to acquire the lock.

Correctness of Synchronization

- **Safety**
  - Locks allows only a single thread into the critical section.

- **Liveness**
  - Locks ensure that a thread may enter as soon as the lock has been released by the previous owner.

  - *We will always assume the programmer did not create a bug in forgetting to release the lock!*

Potential Solution #4

Person 1:

```
acquire_lock(&lock);
buy milk;
release_lock(&lock);
```

Person 2:

```
acquire_lock(&lock);
buy milk;
release_lock(&lock);
```

Rules for Using Locks

- ★ Locks are defined to be available on initialization (“un-owned”).
- ★ You must acquire a lock before accessing any shared data.
- ★ You must always release the lock after accessing any shared data.
  - *Only the lock “owner” should release the lock.*
  - *Never throw the lock to someone else to release it later.*
- ★ **Never** access shared data without a lock.

Example:

Thread 1:

```
if (p == NULL) {
  acquire_lock(&lock);
  if (p == NULL) {
    p = malloc(sizeof(p));
    p->val1 = ...;
    p->val2 = ...;
    ...
  }
  release_lock(&lock);
}

// use p
```

Thread 2:

```
if (p == NULL) {
  acquire_lock(&lock);
  if (p == NULL) {
    p = malloc(sizeof(p));
    p->val1 = ...;
    p->val2 = ...;
    ...
  }
  release_lock(&lock);
}

// use p
```

Thread 1:

```
if (p == NULL) {
  acquire_lock(&lock);
  if (p == NULL) {
    p = malloc(sizeof(p));









    p->val1 = ...;
    p->val2 = ...;
    ...
```

Thread 2:

```
if (p == NULL) {
  acquire_lock(&lock);
  if (p == NULL) {
    p = newP();
  }
  release_lock(&lock);
}

// use p
```

Thread 1:

```
if (p == NULL) {
  acquire_lock(&lock);
  if (p == NULL) {
    p = malloc(sizeof(p));




    p->val1 = ...;
    p->val2 = ...;
    ...
```

Thread 2:

```
if (p == NULL) {
  acquire_lock(&lock);
  if (p == NULL) {
    p = newP();
  }
  release_lock(&lock);
}

// use p
```

Thread 1:

```
if (p == NULL) {
  acquire_lock(&lock);
  if (p == NULL) {
    p = malloc(sizeof(p));







    p->val1 = ...;
    p->val2 = ...;
    ...
```

Thread 2:

```
if (p == NULL) {
  acquire_lock(&lock);
  if (p == NULL) {
    p = newP();
  }
  release_lock(&lock);
}

// use p
```

Seg Fault: p is allocated but not initialized.
...this stuff is tricky!

Bounded Queue Example:

get function:

```
tryget() {
  item = NULL;
  lock.acquire();
  if (front < tail) {
    item = buf[ front % MAX ];
    front++;
  }
  lock.release();
  return item;
}
```

put function:

```
tryput(item) {

  lock.acquire();
  if ( (tail-front) < size) {
    buf[tail % MAX] = item;
    tail++;
  }
  lock.release();

}
```

Initially: front == 0, back == 0, and MAX is capacity.

Q: When tryget() returns NULL, what do we know?

get function:

```
tryget() {
  item = NULL;
  lock.acquire();
  if (front < tail) {
    item = buf[ front % MAX ];
    front++;
  }
  lock.release();
  return item;
}
```

put function:

```
tryput(item) {

  lock.acquire();
  if ( (tail-front) < size) {
    buf[tail % MAX] = item;
    tail++;
  }
  lock.release();

}
```

Q: What is the problem with this user code? ⇒

```
do {
  item = tryget();
} while (!item);
```

get function:

```
tryget() {
  item = NULL;
  lock.acquire();
  if (front < tail) {
    item = buf[ front % MAX ];
    front++;
  }
  lock.release();
  return item;
}
```

put function:

```
tryput(item) {

  lock.acquire();
  if ( (tail-front) < size) {
    buf[tail % MAX] = item;
    tail++;
  }
  lock.release();

}
```

Kernel Lock Implementation

Kernel Lock Implementation

Acquire:

```
Lock::acquire() {
    disableInterrupts();
    if (value == BUSY) {
        waiting.add(myTCB);
        sch_move_to_blocked(myTCB);

        myTCB->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts();
}
```

Release:

```
Lock::release() {
    disableInterrupts();
    if (!waiting.empty()) {
        nextTCB = waiting.remove();
        sch_move_to_ready(nextTCB);


    } else {
        value = FREE;
    }
    enableInterrupts();
}
```

Conditional Variables

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Conditional Variables

- What if we need to wait inside of a critical section?
  - (Ex: Waiting depends on a shared variable's state.)

get function:

```
tryget() {
  item = NULL;
  lock.acquire();
  if (front < tail) {
    item = buf[ front % MAX ];
    front++;
  }
  lock.release();
  return item;
}
```

Busy Waiting:

```
do {
  item = tryget();
} while (!item);
```

Bounded Queue Example:

get function:

```
get() {
  lock.acquire();
  while (front == tail) {
    empty.cond_wait(lock);
  }

  item = buf[ front % MAX ];
  front++;

  full.signal(lock);
  lock.release();
  return item;
}
```

put function:

```
put(item) {
  lock.acquire();
  while ( (tail-front) == MAX ) {
    full.cond_wait(lock);
  }

  buf[tail % MAX] = item;
  tail++;

  empty.cond_signal(lock);
  lock.release();
}
```

Initially: front == 0, back == 0, and MAX is capacity.

Q: What is the state when we enter the critical section?

get function:

```
get() {
  lock.acquire();
  while (front == tail) {
    empty.cond_wait(lock);
  }

  item = buf[ front % MAX ];
  front++;

  full.signal(lock);
  lock.release();
  return item;
}
```

put function:

```
put(item) {
  lock.acquire();
  while ( (tail-front) == MAX ) {
    full.cond_wait(lock);
  }

  buf[tail % MAX] = item;
  tail++;

  empty.cond_signal(lock);
  lock.release();
}
```

...allows for proofs of correctness.

General CV Usage:

wait function:

```
methodThatWaits() {
  lock.acquire();
  // Pre-condition: State is consistent

  while (!testSharedState()) {
    cv.cond_wait(&lock);
  }

  // == Critical Section ==
  // Shared state may have changed from
  // the start of the function.  But
  // testSharedState is TRUE and
  // pre-condition is true.
  ...

  lock.release();
}
```

signal function:

```
methodThatSignals() {
    lock.acquire();
    // Pre-condition: State is consistent

    // ...access shared state...

    // If testSharedState is now true:
    cv.cond_signal(&lock);
    // ...note: signal keeps lock

    lock.release();
}
```

Initially: front == 0, back == 0, and MAX is capacity.

Principles for Conditional Variables:

- ★ ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - ALWAYS hold lock when accessing shared state

- ★ Condition variable is memoryless
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up

- ★ Wait atomically releases lock.

Principles for Conditional Variables:

- When a thread is woken up from wait, it may not run immediately
    - Signal/broadcast put thread on ready list
    - When lock is released, anyone might acquire it

- Wait MUST be in a loop

- Simplifies implementation
    - Of condition variables and locks
    - Of code that uses condition variables and locks

Conditional Variables: MESA vs. Hoare

CS 423 - University of Illinois

Wade Fagen-Ulmschneider

(Slides built from Adam Bates and Tianyin Xu previous work on CS 423.)

Mesa vs. Hoare Semantics

- **Mesa**
  - Signal puts waiter on ready list
  - Signaller keeps lock and processor

- **Hoare**
  - Signal gives processor and lock to waiter
  - When waiter finishes, processor/lock given back to signaller
  - Nested signals possible!

Mesa vs. Hoare Semantics

- **Mesa**
  - *Matches pthread_cond_*() functionality.*

- **Hoare**
  - Works significantly differently from pthread_cond_* functions.

Hoare Semantics

wait function:

```
methodThatWaits() {
  lock.acquire();
  // Pre-condition: State is consistent

  while (!testSharedState()) {
    cv.cond_wait(&lock);
  }

  // == Critical Section ==
  // Shared state may have changed from
  // the start of the function.  But
  // testSharedState is TRUE and
  // pre-condition is true.
  ...

  lock.release();
}
```

signal function:

```
methodThatSignals() {
    lock.acquire();
    // Pre-condition: State is consistent

    // ...access shared state...






    // If testSharedState is now true:
    cv.cond_signal(&lock);
    // ANOTHER THREAD RUNS NOW!!

    lock.release();
}
```

Initially: front == 0, back == 0, and MAX is capacity.