

# Lecture 07: Condvars and Semaphores

CS343 – Operating Systems  
Branden Ghen a – Fall 2022

Some slides borrowed from:

Stephen Tarzia (Northwestern), and Shivaram Venkataraman (Wisconsin)

# Administrivia

- Scheduler lab due by end-of-day today
  - Remember that slip days are automatic, no need to ask
- PC lab available late tonight
  - Practices concurrency issues: detecting and resolving them

# Today's Goals

- Consider another source of concurrency: interrupts
- Understand how we can apply locks to gain correctness and maintain performance
  - Counter
  - Data Structures (bonus, if time is available)
- Signaling between threads to enforce ordering
  - Condition Variables
  - Semaphores

# Review: Locks/Mutexes

- Simple mutual exclusion primitive
- `Init()`, `Acquire()/Lock()`, `Release()/Unlock()`
- Implementations trade complexity, fairness, and performance
  - Spinlocks
  - Ticket locks
  - Yielding locks
  - Queueing locks

# Outline

- **Interrupts**
- Applying Locks
- Ordering with Condition Variables
- Semaphores

# Where else does concurrency come from?

- Processors introduce it for performance reasons by running multiple processes and threads
- Interactions with the outside world introduce it because events occur whenever they feel like it
  - Network request arriving
  - User presses a key
  - Motion sensor triggers
- Also, we need some way to deal with errors that occur when executing instructions
  - No pathway for returning an error from an instruction

# Interrupts

A way for the CPU to be, well, *interrupted*.

- CPU hardware switches to privileged mode
  - Now any instruction can be executed, including privileged ones.
- Execution jumps to a predefined location
  - Handler specified in the CPU's interrupt vector table
  - Lets the kernel deal with whatever the event was
- Used to support asynchronous I/O
  - Lets a hardware device tell the CPU that some data is ready
  - Remember that a disk operation is millions of times slower than an *add*.
- CPU has electrical pin(s) for hardware interrupts.
- There is also an instruction for *software* interrupts (like traps!)

# Interrupt Vector Table

Table 6-1. Exceptions and Interrupts

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode. <sup>1</sup>
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction. <sup>2</sup>
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory. <sup>3</sup>
18	#MC	Machine Check	Error codes (if any) and source are model dependent. <sup>4</sup>
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction <sup>5</sup>
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT <i>n</i> instruction.

Table actually lives in memory somewhere, with function pointers for each vector number

```
match interrupt {
    nvic::ASTALARM => ast::AST.handle_interrupt(),

    nvic::USART0 => usart::USART0.handle_interrupt(),
    nvic::USART1 => usart::USART1.handle_interrupt(),
    nvic::USART2 => usart::USART2.handle_interrupt(),
    nvic::USART3 => usart::USART3.handle_interrupt(),

    nvic::PDCA0 => dma::DMA_CHANNELS[0].handle_interrupt(),
    nvic::PDCA1 => dma::DMA_CHANNELS[1].handle_interrupt(),
    nvic::PDCA2 => dma::DMA_CHANNELS[2].handle_interrupt(),
    nvic::PDCA3 => dma::DMA_CHANNELS[3].handle_interrupt(),
    nvic::PDCA4 => dma::DMA_CHANNELS[4].handle_interrupt(),
    nvic::PDCA5 => dma::DMA_CHANNELS[5].handle_interrupt(),
    nvic::PDCA6 => dma::DMA_CHANNELS[6].handle_interrupt(),
    nvic::PDCA7 => dma::DMA_CHANNELS[7].handle_interrupt(),
    nvic::PDCA8 => dma::DMA_CHANNELS[8].handle_interrupt(),
    nvic::PDCA9 => dma::DMA_CHANNELS[9].handle_interrupt(),
    nvic::PDCA10 => dma::DMA_CHANNELS[10].handle_interrupt(),
    nvic::PDCA11 => dma::DMA_CHANNELS[11].handle_interrupt(),
    nvic::PDCA12 => dma::DMA_CHANNELS[12].handle_interrupt(),
    nvic::PDCA13 => dma::DMA_CHANNELS[13].handle_interrupt(),
    nvic::PDCA14 => dma::DMA_CHANNELS[14].handle_interrupt(),
    nvic::PDCA15 => dma::DMA_CHANNELS[15].handle_interrupt(),
}
```

Example from Tock for SAM4L chip (in Rust)



# Interrupt Vector Table

Table 6-1. Exceptions and Interrupts

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode. <sup>1</sup>
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction. <sup>2</sup>
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory. <sup>3</sup>
18	#MC	Machine Check	Error codes (if any) and source are model dependent. <sup>4</sup>
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction <sup>5</sup>
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT <i>n</i> instruction.

Table actually lives in memory somewhere, with function pointers for each vector number

```
match interrupt {
    nvic::ASTALARM => ast::AST.handle_interrupt(),

    nvic::USART0 => usart::USART0.handle_interrupt(),
    nvic::USART1 => usart::USART1.handle_interrupt(),
    nvic::USART2 => usart::USART2.handle_interrupt(),
    nvic::USART3 => usart::USART3.handle_interrupt(),

    nvic::PDCA0 => dma::DMA_CHANNELS[0].handle_interrupt(),
    nvic::PDCA1 => dma::DMA_CHANNELS[1].handle_interrupt(),
    nvic::PDCA2 => dma::DMA_CHANNELS[2].handle_interrupt(),
    nvic::PDCA3 => dma::DMA_CHANNELS[3].handle_interrupt(),
    nvic::PDCA4 => dma::DMA_CHANNELS[4].handle_interrupt(),
    nvic::PDCA5 => dma::DMA_CHANNELS[5].handle_interrupt(),
    nvic::PDCA6 => dma::DMA_CHANNELS[6].handle_interrupt(),
    nvic::PDCA7 => dma::DMA_CHANNELS[7].handle_interrupt(),
    nvic::PDCA8 => dma::DMA_CHANNELS[8].handle_interrupt(),
    nvic::PDCA9 => dma::DMA_CHANNELS[9].handle_interrupt(),
    nvic::PDCA10 => dma::DMA_CHANNELS[10].handle_interrupt(),
    nvic::PDCA11 => dma::DMA_CHANNELS[11].handle_interrupt(),
    nvic::PDCA12 => dma::DMA_CHANNELS[12].handle_interrupt(),
    nvic::PDCA13 => dma::DMA_CHANNELS[13].handle_interrupt(),
    nvic::PDCA14 => dma::DMA_CHANNELS[14].handle_interrupt(),
    nvic::PDCA15 => dma::DMA_CHANNELS[15].handle_interrupt(),
}
```

Example from Tock for SAM4L chip (in Rust)

# Differences from traps

- When we performed a system call:
  - We knew it was about to happen.
  - Set up our registers in advance.
  - Performed what looked sort of like a function call.
- Interrupts can happen *whenever*.
  - This can get extremely complicated on modern systems with out-of-order execution, multiple cores and threads, and caches

# Interrupt handlers

- Interrupt context
  - Can't just enter the kernel like we did with system calls
  - Interrupt could have occurred while we were in the kernel
- Handler code
  - Execute some *quick* processing to deal with the interrupt
  - Return so the hardware can bring us back to our normal operation
  - Cannot pause to wait for something else to finish first because the entire core jumped to handling this interrupt
- Handled by the operating system
  - Processes are interrupted, but otherwise not normally involved

# Why are interrupts important to the kernel?

- Interrupts are a case where the kernel could have a data race with itself!!
  - Imagine being in the middle of an operation on a device
  - When an interrupt comes in for that same device
  - Data structures for the device could end up messed up
- Takeaway: concurrency isn't just about processes and threads
  - Many different software designs need to deal with it

# Data race fix for single-core machines: disable interrupts

```
void lock() {  
    disable_interrupts();  
}  
  
void unlock() {  
    enable_interrupts();  
}
```

- Disable interrupts to prevent preemption during critical section
  - Scheduler can't run if the OS never takes control
  - Also stops data races in interrupt handlers
- Problems
  - Doesn't work by itself on multicore machines
    - Need to use it AND mutexes
  - Bad Idea™ to let processes disable the OS
    - Process could freeze the entire computer
  - Might screw up timing for interrupt handling

# Outline

- Interrupts
- **Applying Locks**
- Ordering with Condition Variables
- Semaphores

# Review: Need to enforce mutual exclusion on critical sections

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

```
static volatile int counter = 0;
static const int LOOPS = 1e9;
```

```
void* mythread(void* arg) {
    printf("%s: begin\n", (char*)arg);
    for (int i=0; i<LOOPS; i++) {
        counter++;
    }
    printf("%s: done\n", (char*)arg);
    return NULL;
}
```

```
int main(int argc, char* argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    // wait for threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done with both (counter = %d, goal was
%d)\n", counter, 2*LOOPS);
    return 0;
}
```

# Naively locked counter example

```
static volatile int counter = 0;
static const int LOOPS = 1e9;
static pthread_mutex_t lock;
```

```
void* mythread(void* arg) {
    printf("%s: begin\n", (char*)arg);
    for (int i=0; i<LOOPS; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    printf("%s: done\n", (char*)arg);
    return NULL;
}
```

```
int main(int argc, char* argv[]) {
    pthread_t p1, p2;
    pthread_mutex_init(&lock, 0);
    printf("main: begin (counter = %d)\n", counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    // wait for threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done with both (counter = %d, goal
was %d)\n", counter, 2*LOOPS);
    return 0;
}
```



# Problem: locking overhead decreases performance

When iterating one billion times:

Single-threaded counter:	3.850 seconds
Multithreaded no-lock counter:	4.700 seconds (Broken!)
Naïve-locked counter:	80.000 seconds (Correct...)

- Formerly loop contained 3 instructions (mov, add, mov)
- Now it has
  - Two function calls
  - Multiple instructions inside of those
  - Possibly even interaction with the OS...
  - 3 instructions -> 60 instructions

# Simple mutual exclusion: one big lock

- Simple solution “one big lock”
  - Find all the function calls that interact with shared memory
  - Lock at the start of each function call and unlock at the end
- Essentially, no concurrent access
  - Correct but poor performance
  - If you’ve forgotten all of this years from now, “one big lock” will still work

# Counter example with big lock technique

code posted with last  
lecture on canvas

```
static volatile int counter = 0;
static const int LOOPS = 1e9;
static pthread_mutex_t lock;

void* mythread(void* arg) {
    pthread_mutex_lock(&lock);
    printf("%s: begin\n", (char*)arg);
    for (int i=0; i<LOOPS; i++) {
        counter++;
    }
    printf("%s: done\n", (char*)arg);
    pthread_mutex_unlock(&lock);
    return NULL;
}
```

```
int main(int argc, char* argv[]) {
    pthread_t p1, p2;
    pthread_mutex_init(&lock, 0);
    printf("main: begin (counter = %d)\n", counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    // wait for threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done with both (counter = %d, goal  
was %d)\n", counter, 2*LOOPS);
    return 0;
}
```

# Problem: locking decreases performance

Single-threaded counter: 3.850 seconds

Multithreaded no-lock counter: 4.700 seconds (Broken!)

Naïve-locked counter: 80.000 seconds

Big lock counter: 3.895 seconds

- Big lock technique basically returned us to single-threaded execution time (and single-threaded implementation)
- Why is the no-lock multithreaded version so slow?
  - Not 100% certain
  - Likely something to do with hardware memory/cache consistency

# Reducing lock overhead

- We want to enable parallelism, but deal with less lock overhead
  - Need to increase the amount of work done when not locked
  - Goal: lots of parallel work per lock/unlock event
- “Sloppy” updates to global state
  - Keep local state that is operated on
  - Occasionally synchronize global state with current local state
- Counter example
  - Keep a local counter for each thread (not shared memory)
  - Add local counter to global counter periodically

# Sloppy counter example

code posted with last  
lecture on canvas

```
static volatile int counter = 0;
static const int LOOPS = 1e9;
static pthread_mutex_t lock;

void* mythread(void* arg) {
    int sloppy_count = 0;
    printf("%s: begin\n", (char*)arg);
    for (int i=0; i<LOOPS; i++) {
        sloppy_count++;
        if (i%1000 == 0) {
            pthread_mutex_lock(&lock);
            counter += sloppy_count;
            pthread_mutex_unlock(&lock);
            sloppy_count = 0;
        }
    }
}
```

```
int main(int argc, char* argv[]) {
    pthread_t p1, p2;
    pthread_mutex_init(&lock, 0);
    printf("main: begin (counter = %d)\n", counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    // wait for threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: done with both (counter = %d, goal
was %d)\n", counter, 2*LOOPS);
    return 0;
}
```

Offscreen Tail condition: don't forget to update  
"counter" again when the for loop is complete!

# Problem: locking decreases performance

Single-threaded counter: 3.850 seconds

Multi-threaded no-lock counter: 4.700 seconds (Broken!)

Naïve-locked counter: 80.000 seconds

Big lock counter: 3.895 seconds

Sloppy lock (synchronize every 100): 2.150 seconds

Sloppy lock (synchronize every 10000): 1.472 seconds

Sloppy lock (synchronize every 1000000): 1.478 seconds

Sloppy lock (synchronize every 1000000000): 1.500 seconds

- Optimal for this counter example will be synchronizing once, when entirely finished with the local sum

# Break + Open Question

- Avoiding data races is challenging
- Synchronization means we're running some code in parallel anyways
- **Is concurrency worth it? What kinds of problems work best?**



# Break + Open Question

- Avoiding data races is challenging
- Synchronization means we're running some code in parallel anyways
- **Is concurrency worth it? What kinds of problems work best?**
  - Problems that do not share data will still be HUGE wins!
    - No (or few) data races. Big concurrency performance gains.
  - Such problems are termed: *embarrassingly parallel*
    - [https://en.wikipedia.org/wiki/Embarrassingly\\_parallel#Examples](https://en.wikipedia.org/wiki/Embarrassingly_parallel#Examples)

# Outline

- Interrupts
- Applying Locks
- **Ordering with Condition Variables**
- Semaphores

# Requirements for sensible concurrency

- **Mutual exclusion**

- Prevents corruption of data manipulated in critical sections
- Atomic instructions → Locks → Concurrent data structures

- **Ordering** (B runs after A)

- By default, concurrency leads to a lack of control over ordering
- We can use mutex'd variables to control ordering, but it's inefficient:
  - `while(!myTurn) sleep(1);`
- We would like cooperating threads to be able to signal each other.
  - Park/unpark and futex could be used solve this problem
  - But we want a higher-level abstraction

# Barriers for all-or-nothing synchronization

- Barriers create synchronization points in the program
  - **All** threads must reach barrier before **any** thread continues
- `pthread_barrier_init(barrier_t)`
- `pthread_barrier_wait(barrier_t)`
- Use case: neural network processing
  - Spawn a pool of threads
  - Each thread handles a portion of the input data
  - Collect results from all threads at the end of the layer
  - Distribute results to appropriate threads for next layer

# Basic Signaling with Condition Variable (condvar)

- Queue of waiting threads
  - Combine with a **flag** and a **mutex** to synchronize threads
- `wait(condvar_t, lock_t)`
  - Lock must be held when `wait()` is called
  - Puts the caller to sleep AND releases lock (atomically)
  - When awoken, reacquires lock before returning
- `signal(condvar_t)`
  - Wake a single waiting thread (if any are waiting)
  - Do nothing if there are no waiting threads
  - Called while holding the lock; action occurs after lock is released

# Waiting for a thread to finish

```
pthread_t p1, p2;  
  
// create child threads  
pthread_create(&p1, NULL, mythread, "A");  
pthread_create(&p2, NULL, mythread, "B");  
  
...  
  
// join waits for the child threads to finish  
thr_join(p1, NULL);  
thr_join(p2, NULL);  
  
return 0;
```

← How to implement join?

# CV for child wait

- Must use mutex to protect "done" flag and condvar
  - Done flag tracks the event
  - Condvar is used for ordering
- Mutex protects both!

```
1  int done = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

# CV for child wait

- Must use mutex to protect "done" flag and condvar
- **Parent** calls thr\_join()
  - wait()'s until done==1

```
1  int done = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```



# CV for child wait

- Must use mutex to protect "done" flag and condvar
- **Parent** calls thr\_join()
  - wait()'s until done==1
- **Child** calls thr\_exit()
  - sets done to 1
  - calls signal()
  - unlocks mutex

```
1  int done = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

# Buggy attempts to wait for a child, no flag

## Incorrect Code

```
Child
1 void thr_exit() {
2     Pthread_mutex_lock (&m);
3     Pthread_cond_signal (&c);
4     Pthread_mutex_unlock (&m);
5 }
6
Parent
7 void thr_join() {
8     Pthread_mutex_lock (&m);
9     Pthread_cond_wait (&c, &m);
10    Pthread_mutex_unlock (&m);
11 }
```

## Correct Code

```
5 void thr_exit() {
6     Pthread_mutex_lock (&m);
7     done = 1;
8     Pthread_cond_signal (&c);
9     Pthread_mutex_unlock (&m);
10 }
18 void thr_join() {
19     Pthread_mutex_lock (&m);
20     while (done == 0)
21         Pthread_cond_wait (&c, &m);
22     Pthread_mutex_unlock (&m);
23 }
```

- 1) Without *done* variable, the child could run first and signal before the parent starts waiting for the child. Parent waits forever...

# Buggy attempts to wait for a child, no mutex

## Incorrect Code

```
Child
1 void thr_exit() {
2     done = 1;
3     Pthread_cond_signal(&c);
4 }
5
Parent
6 void thr_join() {
7     if (done == 0)
8         Pthread_cond_wait(&c);
9 }
```

## Correct Code

```
5 void thr_exit() {
6     Pthread_mutex_lock(&m);
7     done = 1;
8     Pthread_cond_signal(&c);
9     Pthread_mutex_unlock(&m);
10 }
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
```

- 2) Without a lock, the parent could see `done==0`, then the child could finish and signal, then the parent would start waiting (after missing the signal). Parent waits forever...

# Must check condition with in a loop

1. Thread 1 calls `condvar_wait()`
  2. Thread 2 does some work, calls `condvar_signal()`
  3. Before Thread 1 is scheduled, Thread 3 executes and skips the wait because the resource is ready!
  4. Now Thread 1 will run, but the resource will not be ready
- There is no guarantee that the condition you've been waiting for is true when you are awoken
  - So, we must also use a "predicate loop." (*while*, not *if*)

# Spurious (fake) wakeups

- Pthreads allows wakeup to return not just when a signaled, but also when a ***timer expires*** or for ***no reason at all!***
- Spurious wakeups were included in the specification because they may allow some implementations be more efficient.
- Should be safe since we have to wait with a while loop anyways

```
1  int done = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

# Another Example: Produce/Consumer Problem

- We have multiple producers and multiple consumers that communicate with a shared queue (FIFO buffer).
  - Concurrent queue allows work to happen asynchronously.
  - Buffer has finite size (does not dynamically expand)
- Two operations:
  - *Put*, which should block (wait) if the buffer is **full**.
  - *Get*, which should block (wait) if the buffer is **empty**.
- This is more complex than a (linked-list-based) concurrent queue because of the finite size and waiting.
- Example scenario: request queue in a multi-threaded web server.

# Managing the buffer

```
1  int buffer[MAX];
2  int fill  = 0;
3  int use   = 0;
4  int count = 0;
5
6  void put(int value) {
7      buffer[fill] = value;
8      fill = (fill + 1) % MAX;
9      count++;
10 }
11
12 int get() {
13     int tmp = buffer[use];
14     use = (use + 1) % MAX;
15     count--;
16     return tmp;
17 }
```

- A simple implementation of a circular buffer that stores data in a fixed-size array.
- *fill* is the index of the tail
- *use* is the index of the head
- *count* =  $(fill - use) \% MAX$

This simple implementation assumes:

- Concurrency is managed elsewhere
- It will overwrite data if we try to put more than MAX elements.

# Managing the concurrency

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == MAX)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

- Always acquire *mutex*
  - Must use same mutex in both functions
- Use *two condvars*



# Managing the concurrency

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == MAX)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

- Always acquire *mutex*
  - Must use same mutex in both functions
- Use *two condvars*
- Producer waits on **empty** while the buffer is full
  - Producer signals **fill** after put

# Managing the concurrency

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == MAX)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

- Always acquire *mutex*
  - Must use same mutex in both functions
- Use *two condvars*
- Producer waits on **empty** while the buffer is full
  - Producer signals **fill** after `put`
- Consumer waits on **fill** while the buffer is empty
  - Consumer signals `empty` after `get`

# Managing the concurrency

```
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);
8          while (count == MAX)
9              Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```

- Always acquire *mutex*
  - Must use same mutex in both functions
- Use *two condvars*
- Producer waits on **empty** while the buffer is full
  - Producer signals **fill** after `put`
- Consumer waits on **fill** while the buffer is empty
  - Consumer signals `empty` after `get`
- Loops re-check count condition after breaking out of wait, to handle spurious wakeups.

# Broadcast makes more complex conditions possible

- Recall that *signal* wakes one waiting thread (FIFO)
  - But there are times when threads are not all equivalent
  - The signal may not be serviceable by any of the threads
- For example, consider memory allocation/free requests
  - An allocation can only be serviced by free of  $\geq$  size
- **pthread\_cond\_broadcast** wakes all threads
  - This approach may be inefficient, but it may be necessary to ensure progress

# Condition Variable: rules of thumb

- Shared state determines if condition is true or not
  - Check the state in a while loop before waiting on condvar
- Use a mutex to protect:
  - The shared state on which condition is based, and
  - Operations on the condvar
- Remember to acquire the mutex before calling `cond_signal()` and `cond_broadcast()`
- Use different condvars for different conditions
  - Sometimes, `cond_broadcast()` helps if you can't find an elegant solution using `cond_signal()`

# Break + Administrivia

- Midterm exam is coming soon!
  - Next week Thursday
  - I'll distribute a practice exam soon
- Covers all of class material through this week
  - All of Scheduling and all of Concurrency
- We'll have a review session in class next week Tuesday
  - It'll focus on some practice problems

# Outline

- Interrupts
- Applying Locks
- Ordering with Condition Variables
- **Semaphores**

# Generalizing Synchronization

- Condvars have no state or lock, just a waiting queue
  - The rest is handled by the programmer
- Semaphores are a generalization of condvars and locks
  - Includes internal (locked) state
  - A little harder to understand and use, but can do everything



# Semaphores (by Edsger Dijkstra, 1965)

- Keeps an internal integer value that determines what happens to a calling thread
- Init(val)
  - Set the initial internal value
  - Value cannot otherwise be directly modified
- Up/Signal/Post/V() (from Dutch *verhogen* "increase")
  - Increase the value. If there is a waiting thread, wake one.
- Down/Wait/Test/P() (from Dutch *proberen* "to try")
  - Decrease the value. Wait if the value is negative.



Dijkstra invented  
Dijkstra's Algorithm!

Also Semaphores and the  
*entire field of Concurrent  
Programming*

[https://en.wikipedia.org/  
wiki/Edsger W. Dijkstra](https://en.wikipedia.org/wiki/Edsger_W._Dijkstra)

# Semaphores vs Condition Variables

- Semaphores
  - *Up/Post*: increase value and wake one waiting thread
  - *Down/Wait*: decrease value and wait if it's negative
- Condition Variables
  - *Signal*: wake one waiting thread
  - *Wait*: wait
- Compared to CVs, Semaphores add an integer value that controls when waiting is necessary
  - *Value* counts the quantity of a shared resource currently available
  - *Up* makes a resource available, *down* reserves a resource
  - Negative value **-X** means that **X** threads are waiting for the resource

# Check your understanding

- How would we build a mutex out of a semaphore?

```
typedef struct {  
    sem_t sem;  
} lock_t;  
init(lock_t* lock){  
  
}  
acquire(lock_t* lock) {  
  
}  
release(lock_t* lock) {  
  
}
```

```
sem_init(sem_t*, int initial)  
sem_wait(sem_t*): Decrement, wait until  
                  value >= 0  
sem_post(sem_t*): Increment value then  
                  wake a single waiter
```

# Check your understanding

- How would we build a mutex out of a semaphore?

```
typedef struct {  
    sem_t sem;  
} lock_t;  
init(lock_t* lock){  
    sem_init(&(lock->sem), 1);  
}  
acquire(lock_t* lock) {  
    sem_wait(&(lock->sem));  
}  
release(lock_t* lock) {  
    sem_post(&(lock->sem));  
}
```

```
sem_init(sem_t*, int initial)  
sem_wait(sem_t*): Decrement, wait until  
                  value >= 0  
sem_post(sem_t*): Increment value then  
                  wake a single waiter
```

# Implementing a lock with a semaphore

- Choose an appropriate initial value for the semaphore
- To implement a **Lock**:
  - Initialize to 1 (access to the critical section is the one shared resource)
  - **Lock** → **Down**: (decreases the value and waits if negative)
    - Will decrease the value to 0 if it lock *is not* already taken
    - Will decrease the value to -1 and wait if the lock *is* taken (value was 0)
  - **Unlock** → **Up**: (increases the value and wakes one waiting thread)
    - If value was 0, then no thread was waiting, and no thread is woken
    - If value was -1, then one thread was waiting, and it is woken
    - If value was -x, then x threads are waiting, one is woken, value becomes -(x-1).
  - If value is already 1, *Up* should not be called. (Unlock before lock?!)

# Semaphores reduce effort for numerical conditions

	<b>Condition Variable</b>	<b>Semaphore</b>
Child	<pre>5 void thr_exit() { 6     Pthread_mutex_lock (&amp;m); 7     done = 1; 8     Pthread_cond_signal (&amp;c); 9     Pthread_mutex_unlock (&amp;m); 10 }</pre>	<pre>void thr_exit() {     sem_post(&amp;s); }</pre>
Parent	<pre>18 void thr_join() { 19     Pthread_mutex_lock (&amp;m); 20     while (done == 0) 21         Pthread_cond_wait (&amp;c, &amp;m); 22     Pthread_mutex_unlock (&amp;m); 23 }</pre>	<pre>void thr_join() {     sem_wait(&amp;s); }  sem_init(&amp;s, 0);</pre>

- Want parent to wait immediately so initialize to 0
- If child thread finishes first, semaphore increments to 1

# Readers-Writers Problem

- Some resources don't need strict mutual exclusion, especially if they have many *read-only* accesses. (eg., a linked list)
- Any number of readers can be active simultaneously, but
- Writes must be mutually exclusive AND cannot happen during read
- API:
  - `acquire_read_lock()`, `release_read_lock()`
  - `acquire_write_lock()`, `release_write_lock()`

# Reader-writer Lock

- “lock” semaphore used as a mutex

```
1 typedef struct _rwlock_t {
2     sem_t lock; // binary semaphore (basic lock)
3     sem_t writelock; // used to allow ONE writer or MANY readers
4     int readers; // count of readers reading in critical section
5 } rwlock_t;
6
7 void rwlock_init(rwlock_t *rw) {
8     rw->readers = 0;
9     sem_init(&rw->lock, 0, 1);
10    sem_init(&rw->writelock, 0, 1);
11 }
12
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14    sem_wait(&rw->lock);
15    rw->readers++;
16    if (rw->readers == 1)
17        sem_wait(&rw->writelock); // first reader acquires writelock
18    sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22    sem_wait(&rw->lock);
23    rw->readers--;
24    if (rw->readers == 0)
25        sem_post(&rw->writelock); // last reader releases writelock
26    sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30    sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34    sem_post(&rw->writelock);
35 }
```



# Reader-writer Lock

- “writelock” must be held during read to block writes or during write to block reads.
- During reads
  - Number of active readers is counted.
  - First/last reader handles acquiring/releasing writelock.

```
1 typedef struct _rwlock_t {
2     sem_t lock;          // binary semaphore (basic lock)
3     sem_t writelock;    // used to allow ONE writer or MANY readers
4     int  readers;      // count of readers reading in critical section
5 } rwlock_t;
6
7 void rwlock_init(rwlock_t *rw) {
8     rw->readers = 0;
9     sem_init(&rw->lock, 0, 1);
10    sem_init(&rw->writelock, 0, 1);
11 }
12
13 void rwlock_acquire_readlock(rwlock_t *rw) {
14     sem_wait(&rw->lock);
15     rw->readers++;
16     if (rw->readers == 1)
17         sem_wait(&rw->writelock); // first reader acquires writelock
18     sem_post(&rw->lock);
19 }
20
21 void rwlock_release_readlock(rwlock_t *rw) {
22     sem_wait(&rw->lock);
23     rw->readers--;
24     if (rw->readers == 0)
25         sem_post(&rw->writelock); // last reader releases writelock
26     sem_post(&rw->lock);
27 }
28
29 void rwlock_acquire_writelock(rwlock_t *rw) {
30     sem_wait(&rw->writelock);
31 }
32
33 void rwlock_release_writelock(rwlock_t *rw) {
34     sem_post(&rw->writelock);
35 }
```

# Classical concurrency problems

- Note that this particular solution could starve writers
  - There might always be readers in the critical section
- Full solution to readers-writers problem with progress guarantee
  - [https://en.wikipedia.org/wiki/Readers%E2%80%93writers\\_problem](https://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem)
- Generally: try to map your problem to one of these solved problems
  - Producers/Consumers or Readers/Writers
  - There are MANY solutions to these problems available online

# Outline

- Interrupts
- Applying Locks
- Ordering with Condition Variables
- Semaphores

# Outline

- **Bonus: Concurrent Data Structures**

# Thread-safe data structures

- “Thread safe” – works even if used by multiple threads concurrently
  - Can apply to various libraries, functions, and data structures
- Simple data structures implementations are usually not thread safe
  - Some global state needs to be shared among all threads
  - Need to protect critical sections
- Challenge: multiple function calls each access same shared structure
  - Need to identify the critical section in each and lock it with shared lock

# Linked List

```
void List_Insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        return; // fail
    }
    new->key = key;
    new->next = L->head;
    L->head = new;
    return; // success
}
```

# Concurrent Linked List – Big lock approach

```
void List_Insert(list_t *L, int key) {  
    pthread_mutex_lock(&L->lock);  
    node_t *new = malloc(sizeof(node_t));  
    if (new == NULL) {  
        perror("malloc");  
        pthread_mutex_unlock(&L->lock);  
        return; // fail  
    }  
    new->key = key;  
    new->next = L->head;  
    L->head = new;  
    pthread_mutex_unlock(&L->lock);  
    return; // success  
}
```

Most important part  
of this example.  
Don't forget to unlock  
if returning early.

- Much better than counter example, because we are only serializing the list itself. Hopefully the rest of the code can run concurrently.

# Better Concurrent Linked List – Only lock critical section

```
void List_Insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        return; // fail
    }
    new->key = key;
    new->next = L->head;
    L->head = new;
    return; // success
}
```

**Check your understanding:**

Where is the critical section here?



# Better Concurrent Linked List – Only lock critical section

```
void List_Insert(list_t *L, int key) {  
    node_t *new = malloc(sizeof(node_t));  
    if (new == NULL) {  
        perror("malloc");  
        return; // fail  
    }  
    new->key = key;  
    new->next = L->head;  
    L->head = new;  
    return; // success  
}
```

**Check your understanding:**

Where is the critical section here?

# What about malloc? Is that safe to use??

```
void List_Insert(list_t *L, int key) {  
    node_t *new = malloc(sizeof(node_t));  
    if (new == NULL) {  
        perror("malloc");  
        return; // fail  
    }  
    new->key = key;  
    new->next = L->head;  
    L->head = new;  
    return; // success  
}
```

- Thread-safe functions
  - Capable of being called concurrently and still functioning correctly
  - (Because they use locks!)
- How would we know if malloc is thread-safe?
  - Must check the documentation

Must check the library documentation to determine thread safety

- <https://man7.org/linux/man-pages/man3/malloc.3.html>

- Malloc (and free) is indeed thread-safe

**ATTRIBUTES** [top](#)

For an explanation of the terms used in this section, see [attributes\(7\)](#).

Interface	Attribute	Value
<code>malloc()</code> , <code>free()</code> , <code>calloc()</code> , <code>realloc()</code>	Thread safety	MT-Safe

- If it wasn't, we would have to consider it another shared resource that needs to be locked

# Better Concurrent Linked List – Only lock critical section

```
void List_Insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        return; // fail
    }
    new->key = key;
    pthread_mutex_lock(&L->lock);
    new->next = L->head;
    L->head = new;
    pthread_mutex_unlock(&L->lock);
    return; // success
}
```

- Now new node is created locally in parallel
- Only actual access to the linked list is serialized

# Concurrent Queue

- Separate head & tail locks
- Allows concurrent add & remove
  - Up to 2 threads can access without waiting

```
1  typedef struct __node_t {
2      int          value;
3      struct __node_t *next;
4  } node_t;
5
6  typedef struct __queue_t {
7      node_t      *head;
8      node_t      *tail;
9      pthread_mutex_t headLock;
10     pthread_mutex_t tailLock;
11 } queue_t;
12
13 void Queue_Init(queue_t *q) {
14     node_t *tmp = malloc(sizeof(node_t));
15     tmp->next = NULL;
16     q->head = q->tail = tmp;
17     pthread_mutex_init(&q->headLock, NULL);
18     pthread_mutex_init(&q->tailLock, NULL);
19 }
```

```
21 void Queue_Enqueue(queue_t *q, int value) {
22     node_t *tmp = malloc(sizeof(node_t));
23     assert(tmp != NULL);
24     tmp->value = value;
25     tmp->next = NULL;
26
27     pthread_mutex_lock(&q->tailLock);
28     q->tail->next = tmp;
29     q->tail = tmp;
30     pthread_mutex_unlock(&q->tailLock);
31 }
32
33 int Queue_Dequeue(queue_t *q, int *value) {
34     pthread_mutex_lock(&q->headLock);
35     node_t *tmp = q->head;
36     node_t *newHead = tmp->next;
37     if (newHead == NULL) {
38         pthread_mutex_unlock(&q->headLock);
39         return -1; // queue was empty
40     }
41     *value = newHead->value;
42     q->head = newHead;
43     pthread_mutex_unlock(&q->headLock);
44     free(tmp);
45     return 0;
46 }
```

# Concurrent Queue

- “tailLock” controls adding elements
- Looks similar to ListInsert

```
1  typedef struct __node_t {
2      int          value;
3      struct __node_t  *next;
4  } node_t;
5
6  typedef struct __queue_t {
7      node_t      *head;
8      node_t      *tail;
9      pthread_mutex_t  headLock;
10     pthread_mutex_t  tailLock;
11 } queue_t;
12
13 void Queue_Init(queue_t *q) {
14     node_t *tmp = malloc(sizeof(node_t));
15     tmp->next = NULL;
16     q->head = q->tail = tmp;
17     pthread_mutex_init(&q->headLock, NULL);
18     pthread_mutex_init(&q->tailLock, NULL);
19 }
```

```
21 void Queue_Enqueue(queue_t *q, int value) {
22     node_t *tmp = malloc(sizeof(node_t));
23     assert(tmp != NULL);
24     tmp->value = value;
25     tmp->next = NULL;
26
27     pthread_mutex_lock(&q->tailLock);
28     q->tail->next = tmp;
29     q->tail = tmp;
30     pthread_mutex_unlock(&q->tailLock);
31 }
32
33 int Queue_Dequeue(queue_t *q, int *value) {
34     pthread_mutex_lock(&q->headLock);
35     node_t *tmp = q->head;
36     node_t *newHead = tmp->next;
37     if (newHead == NULL) {
38         pthread_mutex_unlock(&q->headLock);
39         return -1; // queue was empty
40     }
41     *value = newHead->value;
42     q->head = newHead;
43     pthread_mutex_unlock(&q->headLock);
44     free(tmp);
45     return 0;
46 }
```

# Concurrent Queue

- Head lock controls removing elements from front
- Needs to lock almost entire function

```
1  typedef struct __node_t {
2      int          value;
3      struct __node_t  *next;
4  } node_t;
5
6  typedef struct __queue_t {
7      node_t      *head;
8      node_t      *tail;
9      pthread_mutex_t  headLock;
10     pthread_mutex_t  tailLock;
11 } queue_t;
12
13 void Queue_Init(queue_t *q) {
14     node_t *tmp = malloc(sizeof(node_t));
15     tmp->next = NULL;
16     q->head = q->tail = tmp;
17     pthread_mutex_init(&q->headLock, NULL);
18     pthread_mutex_init(&q->tailLock, NULL);
19 }
```

```
21 void Queue_Enqueue(queue_t *q, int value) {
22     node_t *tmp = malloc(sizeof(node_t));
23     assert(tmp != NULL);
24     tmp->value = value;
25     tmp->next = NULL;
26
27     pthread_mutex_lock(&q->tailLock);
28     q->tail->next = tmp;
29     q->tail = tmp;
30     pthread_mutex_unlock(&q->tailLock);
31 }
32
33 int Queue_Dequeue(queue_t *q, int *value) {
34     pthread_mutex_lock(&q->headLock);
35     node_t *tmp = q->head;
36     node_t *newHead = tmp->next;
37     if (newHead == NULL) {
38         pthread_mutex_unlock(&q->headLock);
39         return -1; // queue was empty
40     }
41     *value = newHead->value;
42     q->head = newHead;
43     pthread_mutex_unlock(&q->headLock);
44     free(tmp);
45     return 0;
46 }
```

# Concurrent Hash Table

- Each bucket is implemented with a Concurrent List
  - We don't have to define any locks!
  - (Locks are in the lists)
- A thread can access a bucket without blocking other threads' access to *other* buckets.
- Hash tables are ideal for concurrency.
  - Hash (bucket id) can be calculated without accessing a shared resource.
  - *Distributed hash tables* are used for huge NoSQL databases.

```
1  #define BUCKETS (101)
2
3  typedef struct __hash_t {
4      list_t lists[BUCKETS];
5  } hash_t;
6
7  void Hash_Init(hash_t *H) {
8      int i;
9      for (i = 0; i < BUCKETS; i++) {
10         List_Init(&H->lists[i]);
11     }
12 }
13
14 int Hash_Insert(hash_t *H, int key) {
15     int bucket = key % BUCKETS;
16     return List_Insert(&H->lists[bucket], key);
17 }
18
19 int Hash_Lookup(hash_t *H, int key) {
20     int bucket = key % BUCKETS;
21     return List_Lookup(&H->lists[bucket], key);
22 }
```



# Lock-free data structures

- In our original example, we put a lock around `counter++`
  - We could have instead used `atomic_fetch_and_add` to update counter
  - Lock-free and *still* atomic!!
- This is possible with more complex data structures as well
  - Often based on a compare-and-swap (CAS) approach
  - [https://www.cs.cmu.edu/~410-s05/lectures/L31\\_LockFree.pdf](https://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf)
- Warning: these are not to be taken lightly
  - Atomic instructions have performance costs on processors
  - Getting this correct involves really understanding hardware
  - [https://abseil.io/docs/cpp/atomic\\_danger](https://abseil.io/docs/cpp/atomic_danger)

# Break + Question: Where is the critical section for vector?

```
typedef struct {
    size_t size;
    size_t count;
    int** data;
} vector_t;

void vector_add(vector_t* v, int* item) {
    if (v->count == v->size) {
        v->size *= 2;
        v->data = realloc(v->data, sizeof(int*) * v->size);
    }
    v->data[v->count++] = item;
}
```

# Break + Question: Where is the critical section for vector?

```
typedef struct {  
    size_t size;  
    size_t count;  
    int** data;  
} vector_t;
```

```
void vector_add(vector_t* v, int* item) {  
    if (v->count == v->size) {  
        v->size *= 2;  
        v->data = realloc(v->data, sizeof(int*) * v->size);  
    }  
    v->data[v->count++] = item;  
}
```