Lecture 07: Condvars and Semaphores

CS343 – Operating Systems

Branden Ghena – Spring 2024

Some slides borrowed from:
Stephen Tarzia (Northwestern), and Shivaram Venkataraman (Wisconsin)

Northwestern

Administrivia

- Scheduler lab due by end-of-day today
  - Remember that slip days are automatic, no need to ask
- Get started on PCLab right away as midterm practice

Midterm Exam 1

- Exam Details
  - In class, Thursday April 25. Starts at 12:30 sharp. 80-minute exam
  - Covers all lectures through next week Tuesday
    - (1. Introduction through 8. Synchronization Bugs)
  - You may bring ONE 8.5”x11” sheet of paper with notes on front and back
    - Handwritten, typeset, whatever you want
  - No calculators or other notes

- Review materials
  - Posted to Canvas homepage: practice problems + prior exams
  - Review session: Monday 5-6 pm in Annenberg G21
    - Will practice some problems from those materials

Today's Goals

- 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

- Consider types of synchronization issues that can occur

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

- **Applying Locks**
- Ordering with Condition Variables
- Semaphores
- Synchronization Bugs

Review: Need to enforce mutual exclusion on critical sections

```c
#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;
}
```

```c
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

```c
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;
    }
  }
```

```c
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

Outline

- Applying Locks
- **Ordering with Condition Variables**
- Semaphores
- Synchronization Bugs

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!

```
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

CV for child wait

- Must use mutex to protect “done” flag and condvar

- **Parent** calls thr_join()
  - wait()’s until done==1

```
int done   = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c   = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

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

```
int done   = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Check your understanding: why doesn't this work?

Incorrect Code

Child

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}
```

Parent

```
void thr_join() {
    Pthread_mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Correct Code

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}
void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Consider if an ordering exists that would lead to incorrect behavior

- Lock means that only one critical section will run at a time

Buggy attempts to wait for a child, no flag

Incorrect Code

```
// Child
void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

// Parent
void thr_join() {
    Pthread_mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Correct Code

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}
void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Without *done* variable:

1) The child could run first and signal
2) Before the parent starts waiting for the child
3) Parent waits forever…

Check your understanding: is a lock necessary?

Incorrect Code

Child

```
void thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}
```

Parent

```
void thr_join() {
    if (done == 0)
        Pthread_cond_wait(&c);
}
```

Correct Code

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}
void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

What could go wrong?

- Without the lock, these lines could be interleaved in any way

Buggy attempts to wait for a child, no mutex

Incorrect Code

Child

```
void thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}
```

Parent

```
void thr_join() {
    if (done == 0)
        Pthread_cond_wait(&c);
}
```

Correct Code

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}
void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Without the lock:

1) Parent could see done == 0 and enter the if statement
2) Child could then exit, setting done to 1 and signaling
3) Parent then calls wait (missed the signal) and waits forever

Always use a loop to check the flag variable

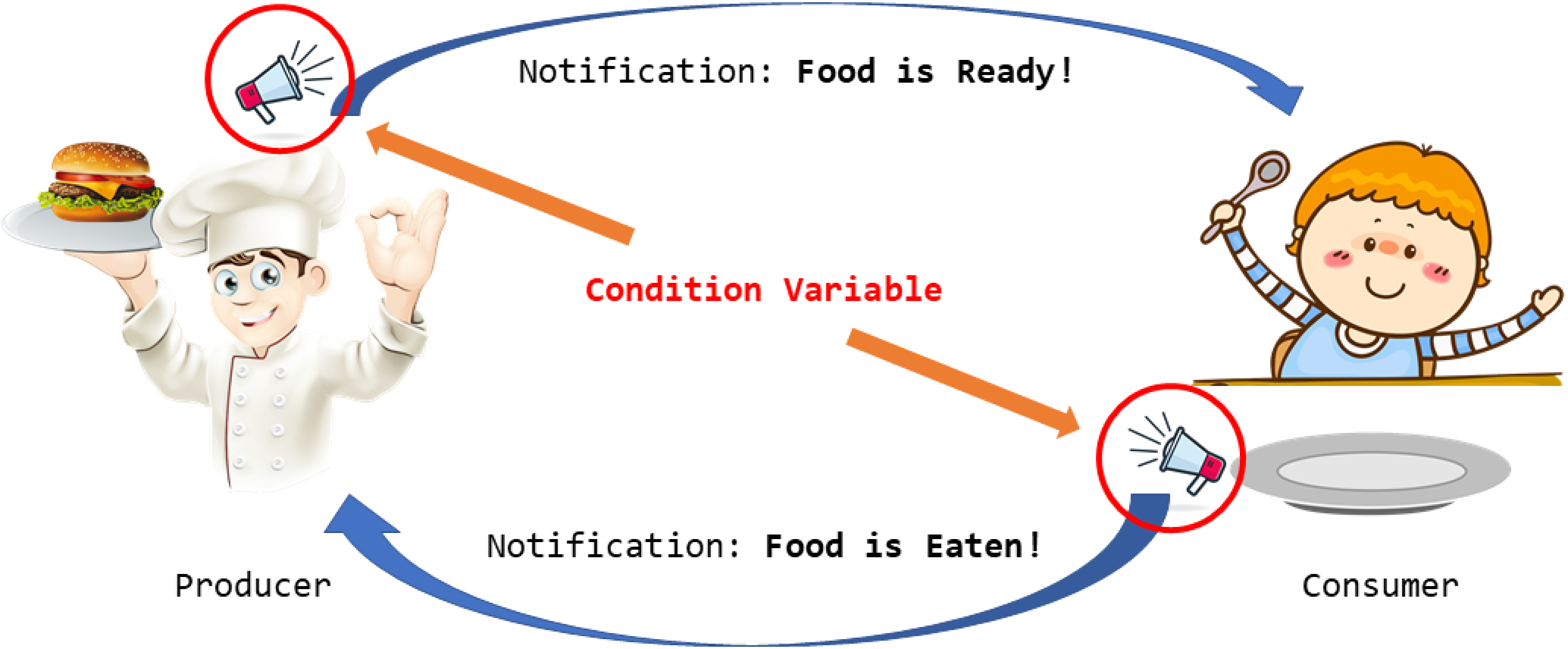
- It's possible for the thread to wake up from a wait, but the resource is not available!

```
void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

- Maybe another thread took the resource first
  - Another thread could run and claim it before the woken thread is scheduled

- Maybe a *spurious wakeup* occurred
  - Often other sources can cause wakeups to occur
    - Signals or Interrupts usually
  - Makes the implementation of condvar simpler, and we need to double-check the flag anyways, so it doesn't matter

Classical concurrency problem: Producer-Consumer



Produce/Consumer Example Details

- 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

```
int buffer[MAX];
int fill  = 0;
int use   = 0;
int count = 0;

void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use];
    use = (use + 1) % MAX;
    count--;
    return tmp;
}
```

- 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*** is the number of items

This simple implementation assumes:

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

Managing the concurrency

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == MAX)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

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

Managing the concurrency

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == MAX)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

- 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

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == MAX)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

- 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

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == MAX)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

- 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 check that there really is a resource

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 >= size

- `pthread_cond_broadcast` wakes all threads
  - This approach may be inefficient, but it may be necessary to ensure progress

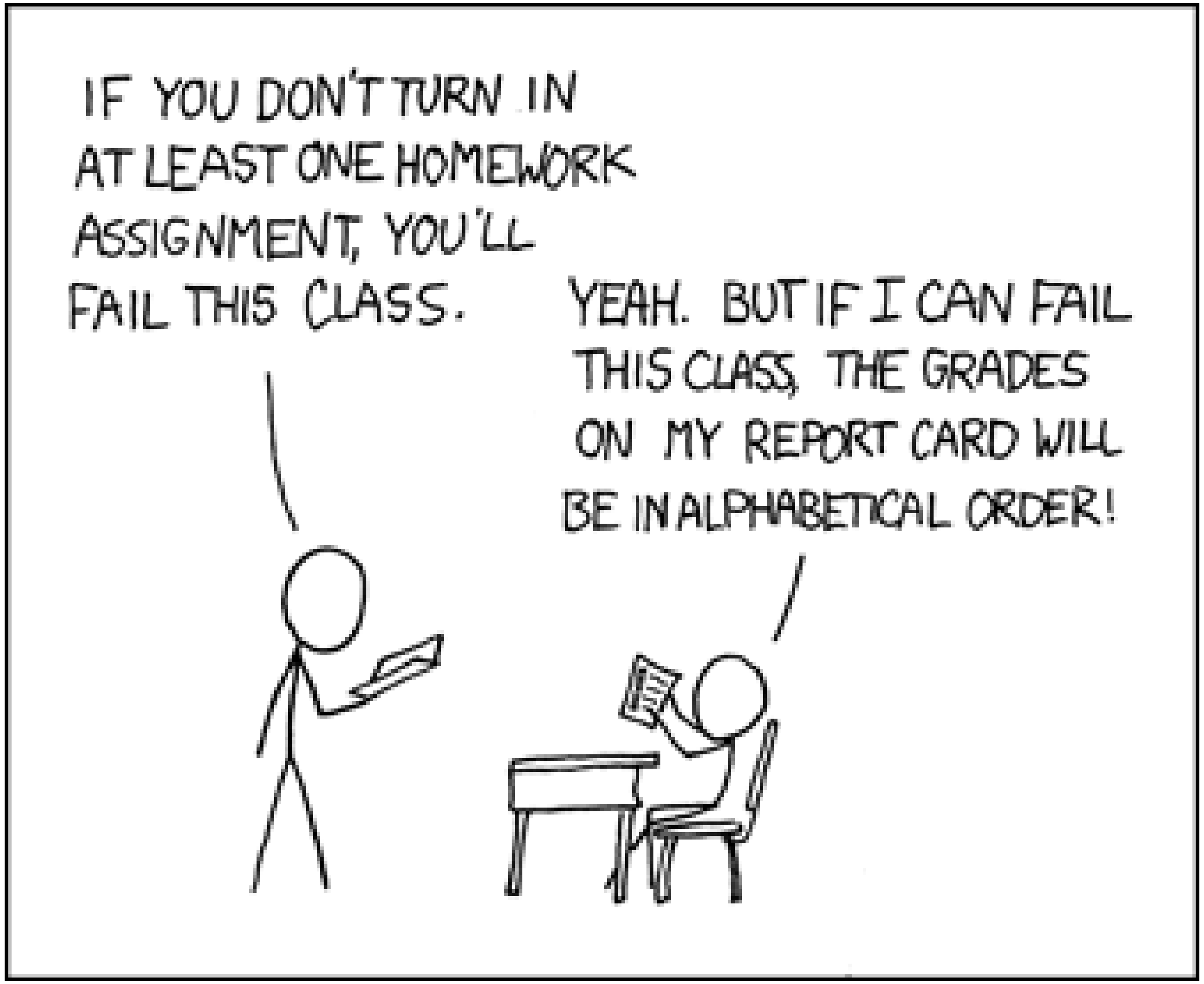
Condition Variable: rules of thumb

- 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 itself

- Use different condvars for different conditions
  - Sometimes, `cond_broadcast()` helps if you can’t find an elegant solution using `cond_signal()`

Break + xkcd (not relevant, just funny)



https://xkcd.com/336/

Outline

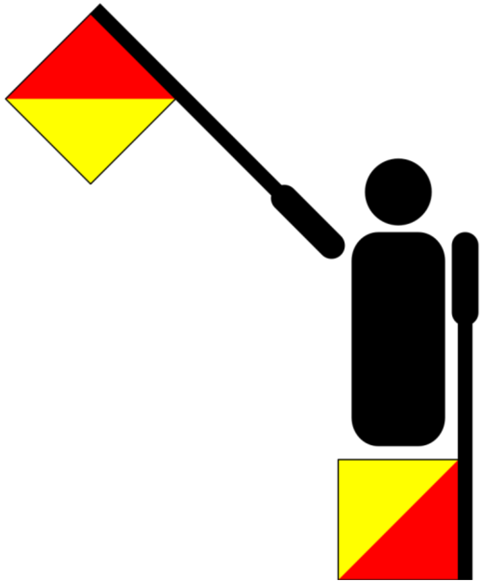
- Applying Locks
- Ordering with Condition Variables
- **Semaphores**
- Synchronization Bugs

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

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: build a mutex

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

```
typdef 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: build a mutex

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

```
typdef 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

Explanation of semaphore mutex implementation

```
typdef 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));
}
```

- The semaphore value represents the number of resources available
  - For a lock, there is 1 available initially
- Acquiring the lock might give it to you immediately
  - Or it might wait
  - Multiple threads could be waiting
- Releasing the lock only occurs after acquiring and resets it to 1

Semaphores reduce effort for numerical conditions

Condition Variable

Child

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}
```

Parent

```
void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Semaphore

```
void thr_exit() {
    sem_post(&s);
}

void thr_join() {
    sem_wait(&s);
}

// somewhere before all of this
sem_init(&s, 0);
```

- Want parent to wait immediately so initialize to 0
- If child thread finishes first, semaphore increments to 1
- Resource: number of threads completed

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

```
typedef struct _rwlock_t {
  sem_t lock;      // binary semaphore (basic lock)
  sem_t writelock; // used to allow ONE writer or MANY readers
  int   readers;   // count of readers reading in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1)
    sem_wait(&rw->writelock); // first reader acquires writelock
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0)
    sem_post(&rw->writelock); // last reader releases writelock
  sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

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.

```c
typedef struct _rwlock_t {
  sem_t lock;      // binary semaphore (basic lock)
  sem_t writelock; // used to allow ONE writer or MANY readers
  int   readers;   // count of readers reading in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1)
    sem_wait(&rw->writelock); // first reader acquires writelock
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0)
    sem_post(&rw->writelock); // last reader releases writelock
  sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

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

- 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

- Applying Locks
- Ordering with Condition Variables
- Semaphores
- **Synchronization Bugs**

Common synchronization bugs

- Atomicity violation
  - Critical section is violated (due to missing lock)
- Order violation
  - Something happens sooner (or later) than expected
- Deadlock
  - Two threads wait indefinitely on each other
- Livelock (not that common in practice)
  - Two threads repeatedly block each other from proceeding and retry

Atomicity violation

- It's not too bad to find and protect critical sections,
  - But often we forget to add locks around other uses of the shared data.
- Obvious critical section is here:
  - Two threads should not enter this at once
- But, we also have to make sure that *file* is not modified elsewhere.
- Even if this one-line *close* was atomic we have to make sure it doesn't run during the above critical section.

```
Main Thread
lock(lck);
if (file == NULL) {
  file = open("~/myfile.txt");
}
write(file, "hello file");
unlock(lck);

…
Some Other Thread
close(file);  // whoops!!
```

Order violation

- Code often requires a certain ordering of operations, especially:
  - Objects must be initialized before they're used
  - Objects cannot be freed while they are still in use

Parent

```
file = open("file.dat");
thread_create(child_fcn);
// do some work
…
close(file);
```

Child Thread

```
child_fcn() {
  write(file, "hello");
}
```

Close must happen after *write*, but code does not enforce this ordering.

Why is this difficult?

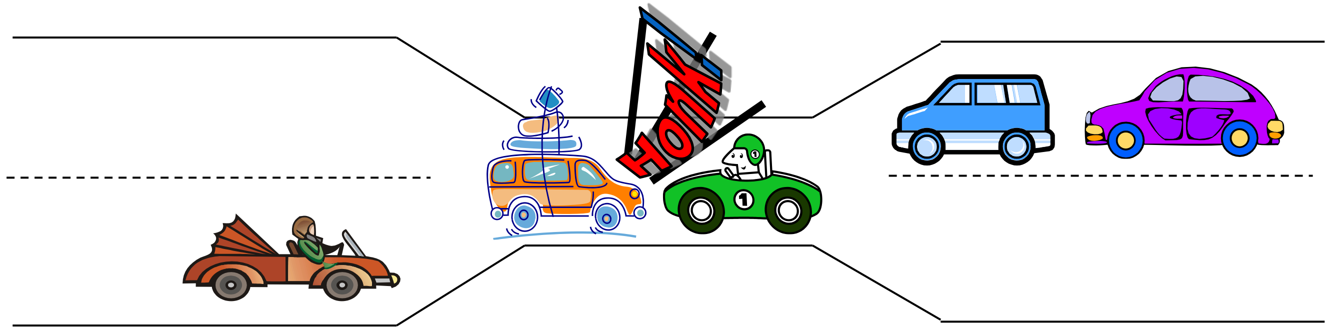
- It seems like we can just add lots of locks and semaphores to be safe, right?
  - Still tricky! Too many locks can cause ***deadlock*** – indefinite waiting.
- How about just one big lock?
  - (+) Cannot deadlock with one lock (unless there are interrupts)
  - (–) However, this would ***limit concurrency***
    - If every task requires the same lock, then unrelated tasks cannot proceed in parallel.
- Concurrent code is always difficult to write ☹
  - Although somewhat easier with *some* higher-level languages

Deadlock

- A concurrency bug arising when:
  - Two threads are each waiting for the other to release a resource.
  - While waiting, the threads cannot release the resource already held.
    - Or at least *do not* release it
  - So the two threads ***wait forever***.

- Can arise when ***multiple*** shared resources are used.
  - For example, acquiring two or more locks.

Deadlock versus starvation



- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into

- **Deadlock:** Two cars in opposite directions meet in middle
- **Starvation** (not deadlock): Eastbound traffic doesn't stop for westbound traffic

Locking granularity

- ***Coarse grained lock*:**
  - Use one (or a few) locks to protect all (or large chunks of) shared state
  - Linux kernel < version 2.6.39 used one "Big Kernel Lock"
  - Essentially only one thread could run kernel code
  - It's simple but there is much contention for this lock, and concurrency is limited

- ***Fine grained locks*:**
  - Use many locks, each protecting small chunks of related shared state
  - Leads to more concurrency and better performance
  - However, there is greater risk of *deadlock*

Outline

- Applying Locks
- Ordering with Condition Variables
- Semaphores
- Synchronization Bugs

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
malloc(), **free**(), **calloc**(), **realloc**()	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

```c
typedef struct __node_t {
    int                 value;
    struct __node_t    *next;
} node_t;

typedef struct __queue_t {
    node_t             *head;
    node_t             *tail;
    pthread_mutex_t     headLock;
    pthread_mutex_t     tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->headLock, NULL);
    pthread_mutex_init(&q->tailLock, NULL);
}
```

```c
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;

    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}

int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->headLock);
        return -1; // queue was empty
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->headLock);
    free(tmp);
    return 0;
}
```

Concurrent Queue

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

```
typedef struct __node_t {
    int              value;
    struct __node_t  *next;
} node_t;

typedef struct __queue_t {
    node_t           *head;
    node_t           *tail;
    pthread_mutex_t   headLock;
    pthread_mutex_t   tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->headLock, NULL);
    pthread_mutex_init(&q->tailLock, NULL);
}
```

```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;

    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}

int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->headLock);
        return -1; // queue was empty
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->headLock);
    free(tmp);
    return 0;
}
```

Concurrent Queue

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

```
typedef struct __node_t {
    int                 value;
    struct __node_t    *next;
} node_t;

typedef struct __queue_t {
    node_t              *head;
    node_t              *tail;
    pthread_mutex_t      headLock;
    pthread_mutex_t      tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->headLock, NULL);
    pthread_mutex_init(&q->tailLock, NULL);
}
```

```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;

    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}

int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->headLock);
        return -1; // queue was empty
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->headLock);
    free(tmp);
    return 0;
}
```

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.

```
#define BUCKETS (101)

typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
    int i;
    for (i = 0; i < BUCKETS; i++) {
        List_Init(&H->lists[i]);
    }
}

int Hash_Insert(hash_t *H, int key) {
    int bucket = key % BUCKETS;
    return List_Insert(&H->lists[bucket], key);
}

int Hash_Lookup(hash_t *H, int key) {
    int bucket = key % BUCKETS;
    return List_Lookup(&H->lists[bucket], key);
}
```

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
- 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

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;
}
```