Week 5: Synchronization in Concurrency

TABLE OF CONTENTS

Agenda

1. Intro, Announcements
2. Lab 2 Reviews/Walkthroughs
3. Review last week
4. More on Synchronization
5. Wrap-up/Summary

Weekly Summary and Where are we?

Topics

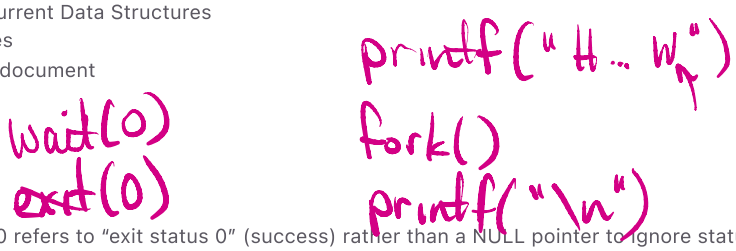
- **Last Week:** Intro to Concurrency and Synchronization
- **This Week:** More about Synchronization
- **Next Week:** Finish up Synchronization, Introduce Virtual Memory

Assignments

- **Lab 3** is out; due 13 Feb 2026.
- **HW 4** is due on Friday
- **Lab 2** grades will be released this Friday
- **HW 3** grades will be released Friday
- Reminder: Regrade requests can be requested until the Friday after the grades are released.

Reading Summary

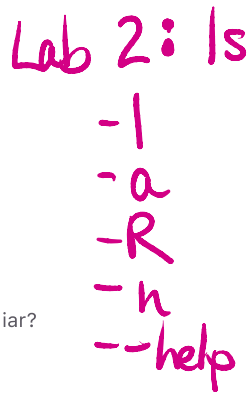
- From last week, still relevant:
  - OSTEP Chapter 25: Dialogue
  - OSTEP Chapter 26: Concurrency and Threads
  - OSTEP Chapter 27: Thread API
  - OSTEP Chapter 28: Locks
- For this week:
  - OSTEP Chapter 29: Lock-Based Concurrent Data Structures
  - OSTEP Chapter 30: Condition Variables
  - Dahlin's "Programming with Threads" document



Announcements

- HW2 grading is done and out
  - Q1a: some student incorrectly stated 0 refers to "exit status 0" (success) rather than a NULL pointer to ignore status.
  - Q2a: some students missed the core concept of the buffer being duplicated
  - Q2b: some students failed to explain the reason for the output (non-determinism)

Lab 2 Reviews



- Who wants to volunteer?
- Short reminder:
  - Implementing `ls`
  - Working with flags, multiple options
  - What did you learn?
  - What was tricky?
  - What of this have you done before/was familiar?

Synchronization

Review from last week

List 5 things!

- 1)
- 2)
- 3)
- 4)
- 5)

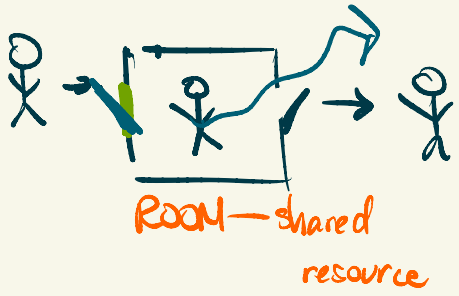
5 Things

- Using flags to lock variables → SHARED RESOURCES
- Race condition
- 2 threads ... touching same variable
- Update: 1 line of code → many assembly instructions
- Non deterministic

Synchronization

Critical Section
- Code
- Modifying shared state
- Concurrency
- Atomically

} Protecting.



- Mutual Exclusion
- Progress
- Bounded Waiting

Critical Sections

> **DEFINITION**
> ***Critical Section*** is a sequence of code that atomically accesses shared state.

We have 2 things to be aware of:

- Protecting critical sections
  - Since critical sections use shared state, we need to make sure we synchronize access
- Implementing critical sections

> **DEFINITION**
> ***Atomicity*** ensures that the processor executes a sequence of instructions without interrupting them— we can guarantee all the instructions in the atomic section are executed completely without interruption.

- Analogy: a room that only allows one person
  - people: threads
  - room: critical section
- Critical section should satisfy 3 properties:

1. **Mutual Exclusion**: only one thread can be in the critical section at a time [this is the notion of ***atomicity***]
2. **Progress**: if no threads are executing in c.s., one of the threads trying to enter a given c.s. will eventually get in
3. **Bounded Waiting**: once a thread `T` starts trying to enter the critical section, there is a bound on the number of other threads that may enter the critical section before `T` enters

- Note on progress vs. bounded waiting
  - If no thread can enter C.S., don't have progress
  - If thread `A` is waiting to enter C.S. while `B` repeatedly leaves and re-enters C.S. ad infinitum, we don't have bounded waiting
  - We will be mostly concerned with ***mutual exclusion*** (in fact, real-world synchronization/concurrency primitives often don't satisfy bounded waiting.)
- Protecting critical sections:
  - want `lock()`/`unlock()` or `enter()`/`leave()` or `acquire()`/`release()`
    - lots of names for the same idea
    - in each case, the semantics are that once the thread of execution is executing inside the critical section, no other thread of execution is executing there
  - In our analogy:
    - mutex => door for the room
    - lock/acquire => close the door (prevent anyone else entering the room)
    - unlock/release => open the door (let the next person enter the room)

Here's our code from last time:

```c
int x;

int main(int argc, char** argv) {

    tid tid1 = thread_create(f, NULL);
    tid tid2 = thread_create(g, NULL);

    thread_join(tid1);
    thread_join(tid2);

    printf("%d\n", x);

}

void f() {
    x = 1;
    thread_exit();
```

```
}

void g() {
    x = 2;
    thread_exit();
}
```

Okay, now let's say that `f()` and `g()` are defined as below:

```
int y = 12;

f() { x = y + 1; }
g() { y = y * 2; }
```

One more time, with `f()` and `g()` defined as below:

```
int x = 0;

f() { x = x + 1; }
g() { x = x + 2; }
```

QUESTION

If we put `f()`/`g()` into critical sections, will it make a difference? what about 1.c?

Answer

Implementing Critical Sections

- The "easy" way, assuming a uniprocessor machine:
  - `enter()` -> disable interrupts
  - `leave()` -> reenable interrupts
  - Are we convinced that this provides mutual exclusion?

IMPORTANT

A ***critical section*** can be protected by a ***lock*** which provides ***mutual exclusion***.

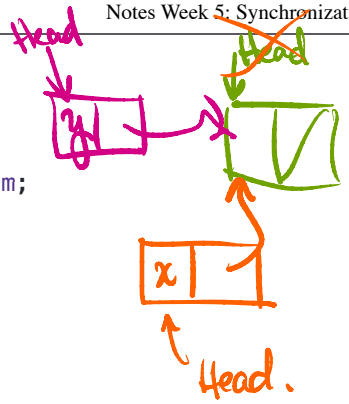
Locks and Mutual Exclusion

- We use locks or mutexes to protect our critical sections
  - Mutex: mutual exclusion object
- Usage (in C):
  - `mutex_t m`
  - `mutex_init(mutex_t* m)`
  - `acquire(mutex_t* m)`
  - `release(mutex_t* m)`
- The `pthread` implementation: `pthread_mutex_init()`, `pthread_mutex_lock()`, ...

Using Locks in Critical Sections

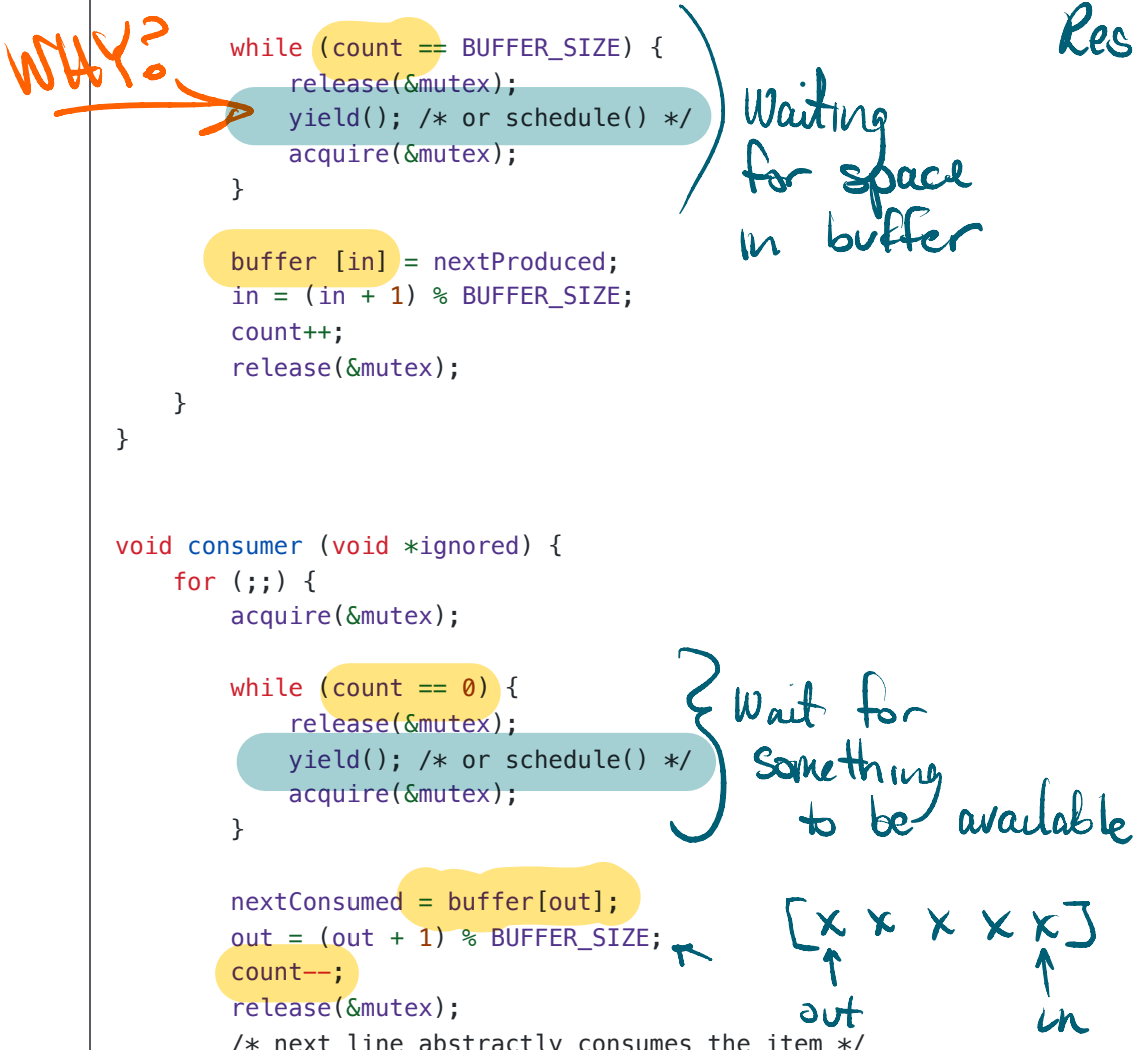
EXAMPLE: LINKED LIST WITH LOCKS

```
Mutex list_mutex;

insert(int data) {
    List_elem* l = new List_elem;
    l->data = data;

    acquire(&list_mutex);

    l->next = head;
    head = l;

    release(&list_mutex);
}
```



EXAMPLE: PRODUCER/CONSUMER WITH LOCKS


```
Mutex mutex;

void producer (void *ignored) {
    for (;;) {
        /* next line produces an item and puts it in nextProduced */
        nextProduced = means_of_production();
        acquire(&mutex);

        while (count == BUFFER_SIZE) {
            release(&mutex);
            yield(); /* or schedule() */
            acquire(&mutex);
        }

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        release(&mutex);
    }
}


void consumer (void *ignored) {
    for (;;) {
        acquire(&mutex);

        while (count == 0) {
            release(&mutex);
            yield(); /* or schedule() */
            acquire(&mutex);
        }

        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        release(&mutex);
        /* next line abstractly consumes the item */
```



```
            consume_item(nextConsumed);
        }
    }
```

- why are we doing this?
  - because *atomicity* is required if you want to reason about the correctness of concurrent code
  - atomicity requires **mutual exclusion**— it's a solution to critical sections
  - mutexes provide that solution!
- With mutexes, we don't have to worry about arbitrary interleavings.
  - Critical sections are interleaved, but those are much easier to reason about than individual operations.
- Why? because of ***invariants***.
  - E.g. "proof by loop invariant"
  - examples of invariants:
    - "list structure has integrity"
    - "'count' reflects the number of entries in the buffer"

One way to think about this:

- the meaning of `lock.acquire()` is that if and only if you get past that line, it's safe to violate the invariants.
- the meaning of `lock.release()` is that right *before* that line, any invariants need to be restored.
- An example:
  - Define the invariant to be: "list structure has integrity"
  - Protect the list with a mutex
  - Only manipulate the list after `acquire()`

QUESTION

Why aren't we worried about *processes* trashing each other's memory?

Answer

Condition Variables

DEFINITION

A ***condition variable*** provide a way for one thread to wait for another thread to take some action.

The thread waits on the desired condition, that some other thread will do something to change.

Motivating Example

- Producer/consumer queue
  - very common paradigm. also called "bounded buffer":
  - *producer* puts things into a shared buffer
  - *consumer* takes them out
  - *producer* must wait if the buffer is full; *consumer* must wait if the buffer is empty
- This shows up everywhere!
  - Soda machine: producer is delivery person, consumer is soda drinkers, shared buffer is the machine
  - OS implementation of `pipe()`
  - DMA (direct memory access) buffers
- Producer/consumer queue using just mutexes (see example code above)
  - what's the problem with that?
  - answer: a form of *busy waiting*
    - In the analogy we were using: the next person keeps knocking the door
- It is convenient to break synchronization into two types:
  - ***mutual exclusion***: allow only one thread to access a given set of shared state at a time
  - ***scheduling constraints***: wait for some other thread to do something (finish a job, produce work, consume work, accept a connection, get bytes off the disk, etc.)
- A condition variable is used to wait for change to a shared state
- A lock must always protect updates to shared state

Condition Variables

- Allows thread to wait for a condition to be true.
  - Mutex, condition.

- Mutual Exclusion
  Only 1 thread can access shared resource.

vs. Scheduling Constraints.
We have to wait for something else to happen before we continue.

- All three methods (`wait`, `signal` and `broadcast`) should only be called when the associated lock is held.

Using Conditional Variables

- The API:
  - `void cond_init (Cond *, ...);`
    - Initialize
  - `void cond_wait(Cond *c, Mutex* m);`
    - Atomically unlock `m` and suspends execution until `c` is signaled
    - Then re-acquire `m` and resume executing
  - `void cond_signal(Cond* c);`
    - Wake one thread waiting on `c`
    - [in some pthreads implementations, the analogous call wakes *at least* one thread waiting on `c`. Check the the documentation (or source code) to be sure of the semantics. But, actually, your implementation shouldn't change since you need to be prepared to be "woken" at any time, not just when another thread calls `signal()`. More on this below.]
  - `void cond_broadcast(Cond* c);`
    - Wake all threads waiting on `c`

EXAMPLE: PRODUCER/CONSUMER WITH MUTEXES AND CONDITION VARIABLES

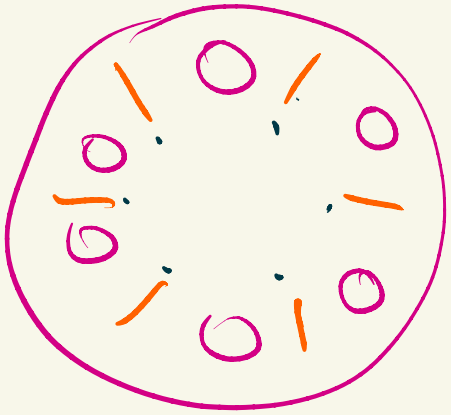
```
Mutex mutex;
Cond nonempty;
Cond nonfull;

void producer (void *ignored) {
    for (;;) {
        /* next line produces an item and puts it in nextProduced */
        nextProduced = means_of_production();

        acquire(&mutex);
        while (count == BUFFER_SIZE)
            cond_wait(&nonfull, &mutex);

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        cond_signal(&nonempty, &mutex);
        release(&mutex);
    }
}

void consumer (void *ignored) {
    for (;;) {
        acquire(&mutex);
        while (count == 0)
            cond_wait(&nonempty, &mutex);

        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        cond_signal(&nonfull, &mutex);
        release(&mutex);

        /* next line abstractly consumes the item */
```

Semaphore

Djikstra

Dining Philosophers

- Like a lock.
  - Has value: N=6
    - decrement
    - increment
  - P()
  - V().



- Give the shared resource an ordering.

Do Not Use

- Provides both Mutual Exclusion AND scheduling constraints.

Mutex + Conditional Variable =
"Monitor".

```
        consume_item(nextConsumed);
    }
}
```

Why does `cond_wait` need to both release the mutex and sleep? Why not:

```
while (count == BUFFER_SIZE) {
    release(&mutex);
    cond_wait(&nonfull);
    acquire(&mutex);
}
```

Some important points

(1) We MUST use "while", not "if". Why?

- Because we can get an interleaving like this:
  - The `signal()` puts the waiting thread on the ready list but doesn't run it
  - That now-ready thread is ready to `acquire()` the mutex (inside `cond_wait()`).
  - But a *different* thread (a third thread: not the signaler, not the now-ready thread) could `acquire()` the mutex, work in the critical section, and now invalidates whatever condition was being checked
  - Our now-ready thread eventually `acquire()`s the mutex...
  - ...with no guarantees that the condition it was waiting for is still true
- Solution is to use "while" when waiting on a condition variable
  - DO NOT VIOLATE THIS RULE; doing so will (almost always) lead to incorrect code
  - NOTE: NOTE: NOTE: There are two ways to understand `while`-versus-`if`: * It's the `while` condition that actually guards the program. * There's no guarantee when the thread comes out of wait that the condition holds.

(2) `cond_wait` releases the mutexes and goes into the waiting state in one function call (see example code block above, "Example: Producer/Consumer with Mutexes and Condition Variables").

- QUESTION: Why?
  - Answer: can get stuck waiting.

```
Producer: while (count == BUFFER_SIZE)
Producer: release()
Consumer: acquire()
Consumer: .....
Consumer: cond_signal(&nonfull)
Producer: cond_wait(&nonfull)
```

- Producer will never hear the signal!

An Aside for Legacy Purposes: Semaphores

- **In this class, we require you to use condition variables and mutexes; semaphores will be considered incorrect.**
- Advice: don't use these. We're mentioning them only for completeness and for historical reasons: they were the first general-purpose synchronization primitive, and they were the first synchronization primitive that Unix supported.
- Introduced by Edsger Dijkstra in late 1960s
- Semaphore is initialized with an integer, `N`
- Two functions:
  - `Down()` and `Up()` [also known as `P()` and `V()`. Why? Dijkstra was Dutch.]
  - The guarantee is that `Down()` will return only `N` more times than `Up()` is called
  - Basically a counter that, when it reaches `0`, causes a thread to `sleep()`

- Another way to say the same thing:
  - Semaphore holds a count
  - `Down()` is an atomic operation that waits for the count to become positive; it then decrements the count by 1
  - `Up()` is an atomic operation that increments the count by 1 and then wakes up a thread waiting on Down(), if any
- Reasons we recommend against their use:
  - semaphores are dual-purpose (for mutual exclusion and scheduling constraints), so hard to read code and hard to get code right
  - semaphores have hidden internal state
  - getting a program right requires careful interleaving of "synchronization" and "mutex" semaphores

Monitors

Monitor = mutex + condition variables

- High-level idea: an object (as in object-oriented systems) in which:
  - methods do not execute concurrently
  - it has one or more condition variables
- Every method call starts with `acquire(&mutex)`, and ends with `release(&mutex)`
  - Technically, these `acquire()` / `release()` are invisible to the programmer because it is the programming language (i.e., the compiler+run-time) that is implementing the monitor
  - So, technically, a monitor is a programming language concept
- But technical definition isn't hugely useful because no programming languages in widespread usage have true monitors
  - Java has something close: a class in which every method is set by the programmer to be "synchronized" (i.e., implicitly protected by a mutex)
    - Not exactly a monitor because there's nothing forcing every method to be synchronized
  - And we can *use* mutexes and condition variables to implement our own manual versions of monitors, though we have to be careful
- Given the above, we are going to use the term "monitor" more loosely to refer to both the technical definition and also a "manually constructed" monitor, wherein:
  - all method calls are protected by a mutex (that is, the programmer inserts those `acquire()` / `release()` on entry and exit from every procedure *inside* the object)
  - synchronization happens with condition variables whose associated mutex is the mutex that protects the method calls
- In other words, we will use the term "monitor" to refer to the programming conventions that you should follow when building multithreaded applications

EXAMPLE: BOUNDED BUFFER WITH A MONITOR

```
// This is pseudocode that is inspired by C++.
// Don't take it too literally.

class MyBuffer {
    public:
        MyBuffer();
        ~MyBuffer();
        void Enqueue(Item);
        Item = Dequeue();
    private:
        int count;
        int in;
        int out;
        Item buffer[BUFFER_SIZE];
        Mutex* mutex;
        Cond* nonempty;
        Cond* nonfull;
}

void
MyBuffer::MyBuffer()
{
    in = out = count = 0;
```

```
    mutex = new Mutex;
    nonempty = new Cond;
    nonfull = new Cond;
}

void
MyBuffer::Enqueue(Item item)
{
    mutex.acquire();
    while (count == BUFFER_SIZE)
        cond_wait(&nonfull, &mutex);

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;
    cond_signal(&nonempty, &mutex);
    mutex.release();
}

Item
MyBuffer::Dequeue()
{
    mutex.acquire();
    while (count == 0)
        cond_wait(&nonempty, &mutex);

    Item ret = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;
    cond_signal(&nonfull, &mutex);
    mutex.release();
    return ret;
}
int main(int, char**)
{
    MyBuffer buf;
    int dummy;
    tid1 = thread_create(producer, &buf);
    tid2 = thread_create(consumer, &buf);

    // never reach this point
    thread_join(tid1);
    thread_join(tid2);
    return -1;
}

void producer(void* buf)
{
    MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
    for (;;) {
        /* next line produces an item and puts it in nextProduced */
        Item nextProduced = means_of_production();
        sharedbuf->Enqueue(nextProduced);
    }
}
```

```
void consumer(void* buf)
{
    MyBuffer* sharedbuf = reinterpret_cast<MyBuffer*>(buf);
    for (;;) {
        Item nextConsumed = sharedbuf->Dequeue();

        /* next line abstractly consumes the item */
        consume_item(nextConsumed);
    }
}
```

Key point: *Threads* (the producer and consumer) are separate from the *shared object* (`MyBuffer`). The synchronization happens in the *shared object*.

Thread Programming: Dahlin's Standard Approach

Why do we have to have standards for thread programming?

- Lots of really smart people have thought really hard about the right abstractions!
  - Your few days of thinking about a new approach is unlikely to yield an advance over the best practices.
- The consequences of getting code wrong can be atrocious.
  - For example:
    - Therac-25: NYTimes
    - Therac-25:
    - Therac-25: Wikipedia
- People who are confident about their abilities tend to perform *worse*, so if you are confident you are a Threading and Concurrency Warrior and/or you think you truly understand how these things work, then you may wish to reevaluate...
  - Dunning-Kruger effect

Dahling's Coding Standards

Overall, refer to the reading that goes into more detail!

RULE 1: ALWAYS DO THINGS THE SAME WAY.

RULE 2: ALWAYS USE **MONITORS**

RULE 3: ALWAYS HOLD THE LOCK WHEN OPERATING ON A CONDITION VARIABLE

- hold lock when doing condition variable operations
- Some people [for example, Andrew Birrell in this paper] will say: "for experts only, no need to hold the lock when signaling". IGNORE THIS. Putting the signal outside the lock is only a small performance optimization, and it is likely to lead you to write incorrect code.

RULE 4: ALWAYS ACQUIRE AT THE BEGINNING OF THE PROCEDURE; RELEASE RIGHT BEFORE RETURN

- acquire/release at beginning/end of methods

RULE 5: ALWAYS USE `while`, NOT `if`

RULE 6: (ALMOST) NEVER `sleep()`

=======

- RULE:
  - a thread that is in `wait()` must be prepared to be restarted at any time, not just when another thread calls "`signal()`".
  - why? because the implementor of the threads and condition variables package *assumes* that the user of the threads package is doing `while(){wait()}`.

An Approach to Concurrent Programming

We'll use our earlier Producer/Consumer with Monitors as an example.

(This is adapted from Operating Systems and Practices, Anderson and Dahlin)

1. Start as we do with the class design for most single-thread problems
   - Decompose the problem into objects
   - For each object:
     - Define a clean interface
     - Identify the correct internal state and invariants to support the interface
     - Implement methods to manipulate that state
2. Identify the shared resources. See below for more details.
3. Add a lock
4. Add code to `acquire` and `release` the lock
5. Identify and add condition variables
6. Add loops to wait using the condition variables
7. Add `signal` and `broadcast` calls

The step to identify and protect the shared resources is the most difficult part.

1. GETTING STARTED:

- **Identify units of concurrency**.
  - Make each a thread with a `go()` method or main loop.
  - Write down the actions a thread takes at a high level.
- **Identify shared chunks of state**.
  - Make each shared *thing* an object.
    - Identify the methods on those objects, which should be the high-level actions made *by* threads *on* these objects.
    - Plan to have these objects be protected by mutexes.
- Write down the high-level main loop of each thread.

Advice:

- **Stay high level here.** Don't worry about synchronization yet. Let the objects do the work for you.
- **Separate threads from objects**. The code associated with a thread should not access shared state directly (and so there should be no access to locks/condition variables in the "main" procedure for the thread).
  - Shared state and synchronization should be encapsulated in shared objects.
- **QUESTION**: how does this apply to the example?
  - separate loops for `producer()`, `consumer()`, and synchronization happens inside MyBuffer.

2. FOR EACH OBJECT, WRITE DOWN CONSTRAINTS

- For each object, write down the synchronization constraints on the solution. Identify the type of each constraint: mutual exclusion or scheduling.
  - For scheduling constraints, ask, "when does a thread wait"?
  - NOTE: usually, the mutual exclusion constraint is satisfied by the fact that we're programming with monitors.
- **QUESTION**: how does this apply to the example?
  - Only one thread can manipulate the buffer at a time (mutual exclusion constraint)
  - Producer must wait for consumer to empty slots if all full (scheduling constraint)
  - Consumer must wait for producer to fill slots if all empty (scheduling constraint)

3. CREATE LOCKS

- Create the locks or condition variables corresponding to each constraint
- **QUESTION**: how does this apply to the example?
  - Answer: need a lock and two condition variables. (lock was sort of a given from the fact of a monitor).

4. WRITE THE METHODS,...

...using locks and condition variables for coordination

[thanks to David Mazieres, Mike Dahlin, and Mike Walfish for content in portions of this lecture.]

Examples

EXAMPLE: A DATABASE WITH MULTIPLE READERS AND WRITERS

The high–level goal here is (a) to give a writer exclusive access (a single active writer means there should be no other writers and no readers) while (b) allowing multiple readers.

```
// This is psuedocode!

// assume that these variables are initialized in a constructor
state variables:
AR = 0; // # active readers
AW = 0; // # active writers
WR = 0; // # waiting readers
WW = 0; // # waiting writers

Condition okToRead = NIL;
Condition okToWrite = NIL;
Mutex mutex = FREE;

Database::read() {
    startRead(); // first, check self into the system
    Access Data
    doneRead();
}

Database::startRead() {
    acquire(&mutex);
    while((AW + WW) > 0){
        WR++;
        wait(&okToRead, &mutex);
        WR--;
    }
    AR++;
    release(&mutex);
}

Database::doneRead() {
    acquire(&mutex);
    AR--;
    if (AR == 0 && WW > 0) { // if no other readers still
        signal(&okToWrite, &mutex);
    }
    release(&mutex);
}

Database::write(){ // symmetrical
    startWrite(); // check in
    Access Data
    doneWrite(); // check out
}

Database::startWrite() {
    acquire(&mutex);
    while ((AW + AR) > 0) { // check if safe to write.
        // if any readers or writers, wait
        WW++;
        wait(&okToWrite, &mutex);
```

```
        WW--;
    }
    AW++;
    release(&mutex);
}

Database::doneWrite() {
    acquire(&mutex);
    AW--;
    if (WW > 0) {
        signal(&okToWrite, &mutex); // give priority to writers
    } else if (WR > 0) {
        broadcast(&okToRead, &mutex);
    }
    release(&mutex);
}
```

NOTE: what is the starvation problem here?

EXAMPLE: SHARED LOCKS

```
struct sharedlock {
    int i;
    Mutex mutex;
    Cond c;
};

void AcquireExclusive (sharedlock *sl) {
    acquire(&sl->mutex);
    while (sl->i) {
        wait (&sl->c, &sl->mutex);
    }
    sl->i = -1;
    release(&sl->mutex);
}

void AcquireShared (sharedlock *sl) {
    acquire(&sl->mutex);
    while (sl->i < 0) {
        wait (&sl->c, &sl->mutex);
    }
    sl->i++;
    release(&sl->mutex);
}

void ReleaseShared (sharedlock *sl) {
    acquire(&sl->mutex);
    if (!--sl->i)
        signal (&sl->c, &sl->mutex);
    release(&sl->mutex);
}

void ReleaseExclusive (sharedlock *sl) {
    acquire(&sl->mutex);
```

```
    sl->i = 0;
    broadcast (&sl->c, &sl->mutex);
    release(&sl->mutex);
}
```

QUESTIONS

1 There is a starvation problem here. What is it?

Answer

1 How could you use these shared locks to write a cleaner version of the code in the prior item? Note, however, that the starvation properties would be different.

Answer

Summary/Wrap-Up

- 1)
- 2)
- 3)
- 4)
- 5)

Let's put these in Khanmigo and get a summary.