COMP 409 Study guide

Francis Piché

February 24, 2019

Contents

Part I
Preliminaries

1 License Information

These notes are curated from Professor Clark Verbrugge COMP409 lectures at McGill University. They are for study purposes only. They are not to be used for monetary gain.

Part II
Review / Intro

2 Basic Concepts

2.1 Parallelism vs Concurrency

Theres a subtle (and often blurred) difference between parallel and concurrent programming. Often the two are used interchangeably. Both are models of multiprogramming.

Formally **parallelism** is the idea of having multiple processors each working on a task at a particular moment in time. Think of it as having two workers digging a hole at the same time.

Concurrency is more general. It simply requires that two tasks are being performed (not necessarily at the exact same moment in time). There may be multiple processors, but that does not mean that two or more are running the task at the same time. With concurrency, we might imagine digging two holes by yourself, by taking a shovel out of one hole, then the other, back and forth.

2.2 Multi-processing vs Multi-threading

As seen in Operating Systems, multi-processing is the idea of running multiple processes at the same time. A process contains much more information (and overhead) than a thread, and so is "heavier". Multi-threading, on the other hand is much more light weight since threads belong to processes. So a program wanting to multi-thread wouldn't need to spawn child processes, only threads (belonging to the parent process), and still achieve a high amount of

concurrency.

2.3 Asynchronous vs Synchronous Execution

Asynchronous execution is when two (or more) threads of execution don't need to wait for another to finish before continuing. There doesn't need to be any synchronization between the threads.

Synchronous execution on the other hand is when one thread may need to wait for another to reach a particular point in the program before continuing.

2.4 Threads

Every thread has:

- Thread ID
- Scheduling policy
- Priority
- Signal Mask
- Register Set
- Program counter (PC)
- Stack pointer

The first 4 are managed by the OS, while the last 3 are part of the thread context.

Threads are mainly used to speed things up. Even in a single processor machine, using threads (and switching between them) can make a program more responsive, since there is a lot of wasted time in regular single threaded programs (cache misses, waiting for IO etc).

There is a limit, however. **Amdahls Law** states that the potential benefit of multiprocessing is limited by the portion of the program which is parallelizable.

$$speedup = \frac{1}{s + \frac{p}{n}}$$

or

$$speedup = \frac{1}{(1-p) + \frac{p}{n}}$$

where s is the portion of the program that is serial, and p is the portion that is parallelizable (1-s).n is the number of processors.

Threads are good for **hiding latency**. A single threaded application may wait for a long time on cache misses, I/O etc. With multiple threads this time can be utilized for more work, or for increasing responsiveness.

Part III
Hardware

3 Hardware

3.1 Uniprocessor

This is just a simple 1-processor system. There's no coarse parallelism, but can still have low-level concurrency through pipelining (implemented in the hardware of the CPU). Can still have concurrency through fast context switches.

3.2 Multiprocessors

In a multiprocessor, multiple cpu cores can each handle workloads. They may each have their own cache, but then the caches must stay in sync. A **UMA** design (Uniform Memory Access) makes all memory accesses the same (and so the overhead is the same) for each core in the multiprocessor. NUMA (not-UMA) allows some or all cores to have a fast-access local memory,as well as the slower main memory.

Within the UMA processors there are :

- SMP (symmetric multiprocessor): simply two (or more) CPU's hooked up to one main memory.
- CMP (on Chip multiprocessor): CPU's are on the same chip, for easier communication

The drawback to both of these is that a lot of programs are still single threaded, and so there isn't always a benefit from multiple cores.

- CMT (Coarse grain multithreading) is the idea of having one big fast CPU with lots of register sets. This allows support for hardware threads.
- FMT (Fine grain multithreading) is similar, but instead of big context switches, do "Barrel processing", in which each instruction is a different thread.

- SMT (Simultaneous Multi-threading) this supports true parallelism, but doesn't suffer in single-threaded mode. It acts like a CMP, but in single threaded mode, uses resources from the other CPU. Also known as hyper-threading on Intel CPU's.

Part IV
Atomicity

In order to write concurrent systems, we need to know what will and won't go wrong. If there are many possible interleavings of instructions, some interleavings lead to different results. Atomicity is when an instruction cannot be interleaved, even at the machine code level.

Definitions: The **read set** of a thread is the set of variables a thread reads. The **write set** of a thread is the set of variables that the thread writes.

Two threads are independent if the write set of each is disjoint from the read and write sets of the other.

Even simple variable assignments may not be atomic. For example:

```
x = y + z // seems harmless

//but is really
load r1, y
load r2, z
add r3, r1, r2
store r3, x
```

which, depending on the order of the instructions can lead to different results.

We must also be careful of datatypes. If the type is not word-sized, for example a 64 bit double on a 32-bit machine. Then manipulating these takes several steps.

In Java, we assume a 32 bit machine, and can use the volatile keyword to make reads/writes atomic.

Let $x = expr$ be a statement where x is a word sized variable, and $expr$ is an expression. We say that $expr$ has a **critical reference** if it uses a variable which another thread changes. We say that $x = expr$ has an "At-most-once" property if it either: has exactly 1 critical reference, and x is not read by another thread OR expr has no critical reference.

A statement with at-most-once can be treated as atomic.

4 Mutual Exclusion

Mutual Exclusion is the idea of ensuring that no two threads execute the same block of code at the same time. We need this since the number of possible interleavings is huge, and minimizing leads to easier reasoning about the programs.

4.1 Busy Waiting

One form of mutual exclusion is busy-waiting. This is when we simply use a spinning loop that does nothing but check a condition. Once the condition is false the lock is released.

Suppose we have two threads with ID's 0, 1. Let's look at several different approaches to mutual exclusion using busy-waiting.

```
init(){
   turn = 0
}

enter(id){
   while(turn != id);
}

exit(id){
   turn = 1 - turn
}
```

This will work, by enforcing that only one thread may enter at one time. However, Thread 0 must enter first (since turn is 0), which means that even if Thread 0 is not in the critical section, Thread 1 cannot enter, which is a waste. Also, we have strict alternation. This means that thread 0 and 1 must execute the same number of times.

```
init(){
   flag[0] = flag[1] = false
}

enter(id){
   while(flag[1-id]);
   flag[id] = true
}
```

```
exit(id){
   flag[id] = false
}
```

This will not work. This is because both threads might check flag[1-id] at the same time, and see that it is set to false. Then both would be able to enter the critical section.

```
init(){
   flag[0] = flag[1] = false
}

enter(id){
   flag[id] = true
   while(flag[1-id]);
}

exit(id){
   flag[id] = false
}
```

Note that this is basically the same as the previous, but with the operations swapped. This makes a big difference, since this solution enforces mutual exclusion. However, its possible that both threads set the flag at the same time, and both see that it is set to true, and so both get stuck waiting.

```
init(){
   flag[0] = flag[1] = false
}

enter(id){
   flag[id] = true
   while(flag[1-id]){
      flag[id] = false
      wait(random_time)
      flag[id] = true
   }

}

exit(id){
   flag[id] = false
}
```

Here, this works by "backing out" if the other thread is using the critical section, waiting

for a random amount of time, and trying again. It is possible that both threads wait for the same amount of time, although unlikely.

```
init(){
   flag[0] = flag[1] = false
   turn = 0
}

enter(id){
   flag[id] = true
   turn = id
   while(turn == id && flag[1-id]);
}

exit(id){
   flag[id] = false
}
```

This is known as Petersons algorithm. It works in all cases by using a combined condition.

4.2 Conditions

5 Java Threads

There are two ways of defining a thread in Java. Either by extending the Thread object, or defining a class that implements the Runnable interface.

The basic structure of a thread is as follows:

```
Thread {
   Runnable
   Thread(Runnable run){
      r = run; //keep track of the runnable
   }

   start() //native method to create a Thread (run self or the Runnable that was
       passed)
}
```

Calling start() calls run on either the current thread or the Runnable that was passed to it if it is not null.

You must define what the run() method actually does.

Extending Thread is nice but affects your hierarchy since you only have single inheritance in Java. This is what the Runnable interface helps, since you can then extend something else.

You should extend Thread if you're trying to change Thread behavior. Ie. trying to create a new "Type" of thread. But if you're just trying to run code inside a thread then it's fine to just implement Runnable.

You can create essentially as many threads as you want, (theres a high upper limit).

5.1 Useful API'S

- Thread.sleep(time: int) Goes to sleep for specified time. (Lower-bound)
- Thread.yield() Tells OS we are done, can schedule another thread. (only a hint, not concrete)
- Thread.currentThread() Get the current thread object
- Thread.isAlive() if we call this on current thread, then always alive. We can call on another thread to check to see if the thread has moved to a dead state. If we get a true, it may or may not be alive. It just means the thread was alive when the call was made. It's possible that by the time we get back from the call, the thread dies.

Note that there is no stop, destroy suspend or resume API methods, even if we often want to do this.

6 Ending a Multithreaded Program

How does a multithreaded program terminate? In a single threaded program it terminates when main() completes. In a multithreaded program, do we finish when the first thread finishes? When all threads finish?

All threads must finish. Ie, the first thread must finish main(), and then all spawned threads must complete their run() method.

To be more precise, all **non-daemon** threads must finish. Non-daemon is the default type of thread. **Daemon** threads on the other hand are meant to act like services. You start them, and they sit around waiting for other threads to use them. They do not actually "run" on their own. They do not keep the program alive. If all non-daemon threads have terminated, all the daemon threads will be killed.

7 Execution

When we create a new thread, it exists inside a *started* state. Once we call run(), we move into the *ready* state. The OS then schedules the threads, moving them to a *running* state. The OS can also de-schedule the thread back to a ready state. We can also reach a *sleeping*

state. Here, the thread exists, but does not want to run. The threads may wake up, and go back to the ready state. We can stop a thread to move it to a *dead* state (from which it can never return).

In general, more threads can be ready to execute than we have CPU's. Say we have 100 threads, but only 2 cores, we clearly need to switch between them.

Java uses a priority system to decide which thread gets to execute. This is a **"nominally priority preemptive**". Which means that higher priority gets executed preferentially. It's nominally means that not much is formally guaranteed. (It says its priority based but is more of a name than a rule).

For example, suppose we have 3 priority levels, high, med and low. Each level has a list of threads which exist at that level. Say threads A, B, C are high, D, E are med, F, G, H are low. Suppose we have 3 cores. We would first schedule A, B and C for as long as they still need to run. It's possible that the rest of the cores never get to execute, if A, B and C never finish. If we had only 2 cores, then A, B and C would be time-sliced. After some time, one of A, B would be replaced by C, and so on. Again, the med-low priority are starved until A, B, C complete.

If we had 4 cores, 3 cores would be used for A, B, C and the last core would be split between the medium priority cores.

If a higher priority thread stops, goes to sleep or finishes, then some thread from the next level can run.

However, none of this is guaranteed. (Only nominally priority based) We assume time-slicing, but we could be on a system which does not have time-slicing. We assume that our priorities are respected, but it's not necessarily true.

8 Synchronized

```
synchronized(x){
   ...
}
```

Every object in Java has a lock associated with it. (Including static Class objects). calling synchronized means "aquire" the lock for the object x. The lock is released when synchronized method is completed. This give mutual exclusion since only one thread can hold the lock on the object at one time.

Suppose we don't want anyone to execute ANY method of an object at one time, (across methods), we can do the following.

```
class foo{

   m1(){
      synchronized(this)
   }

   m2(){
      synchronized(this)
   }
}
```

Now the foo object cannot execute m1 and m2 at the same time. However there's a short hand:

```
class foo{
   synchronized m1(){...}
   synchronized m2(){...}
}
```

What if we use recursion in these?

```
synchronized int fib(n){
   ...
   fib(n-1)
}
```

The synchronized keyword ensures that only one thread holds a lock, but that thread may hold the lock as many times as it wants. So recursive locking is ok. But we can also just call m1 from inside m2 and have the same effect (double locking).

The **volatile** keyword can also be used to make 64 bit reads/writes atomic. There is another use for it. Consider this example:

```
static int x;
while(x != 1);
```

Where some other thread does x=1. The compiler might try to do some optimization by putting x into a register, and will never check memory to see if some other thread changed the value of x. Or worse, it might think that because it's static, it will never be updated. Using the volatile keyword tells the compiler that the value might change, so it will always read from memory rather than using some register.

So, we MUST use the volatile keyword for variables that will be accessed by multiple threads.

9 Race Conditions

A race condition is a statement in which the order in which the threads reach the statement determines the outcome. So, since the order in which they arrive is non-deterministic, then the result will be non-deterministic. Even with locks, we will STILL have non-determinism. For example, two threads doing x++ is non-deterministic, but locks will make it deterministic, which is not a race condition.

"A data-race in a Multithreaded program occurs when 2 threads access the same memory location with no ordering constraints such that at least one of them is a write."

We could use the volatile keyword to enforce mutual exclusion in data-race situations.

10 PThreads

PThreads are a library based approach to threading. They are based on the POSIX standard.

Creating a pthread:

```
pthread_create(
   thread handle,
   attributes,
   start routine,
   args
)
```

We can create threads that are **joinable** (default) in which case we must join otherwise we get a resource leak. Or, they can be **detracted**.

We can also specify the scheduling policy to be round robin, first in first out, or other.

10.1 PThread Synchronization

Mutex can be used for mutual exclusion. We create one using: `pthread_create_mutex()`. And lock/unlock it using: `pthread_mutex_lock(m)` and `pthread_mutex_unlock(m)`

Note that thread unlocking is not syntactically guaranteed to be the same thread that locked it. If we need this condition, then it is implementation defined.

Also not that it does not by default support **recursive locking**. We can use vendor extensions if we want this.

11 Locks

Recall Peterson's 2-process tie-breaker algorithm. Can we extend this to n threads?

11.1 Tournament Lock

The tournament lock is a generalization of Petersons algorithm. It uses Petersons 2-thread algorithm in $n-1$ stages.

At stage n at least one thread entering the stage succeeds. If more than 1 thread tries to enter stage n, then at least one is blocked.

```
init(){
   int stage[n] //Id of the highest stage each thread is trying to get into. Each
       thread has an entry in this array.

   int waiting[n] //Which thread was the last to get into stage i. Each stage has
       an entry in this array
}

enter(id){
   for(i=1; i<n; i++){
      stage[id] = i; //Mark the stage this thread is trying to enter
      waiting[i] = id; //Mark the last thread to try and enter this stage

      do{
         spin = false;
         for(j=0; j<n; j++){
            if(j==id) continue;
            if(stage[j] >= i && waiting[i] == id){ //If there is a thread at a
                higher level and I'm the last thread to try to get this level,
                then i'm locked out.
               spin = true;
               break;
            }
         }

      }while(spin);
   }
}
```

```
exit(id){
   stage[id] = 0;
}
```

For example, suppose $n = 3$ threads.

```
stage = [0, 0, 0];
waiting = [0 0, 0];

T0-------------
i=1
stage[0] = 1
waiting[1] = 0

T1-------------
i=1
stage[1] = 1
waiting[1] = 1

T0--------------
i=1
if(stage[1] >= 1 && waiting[1] == 0){
   enter next stage
}

T1--------------
j=0
stage[0] >= 1 && waiting[1] == 1
   spin
```

Proof it works:
REVISIT THIS

Lemma: There are at most $n-i$ threads at stage i. So at stage $n-1$ we at most $n-(n-1)$ threads ie 1-thread.

Induction on i:
Base case: $i = 0$
Suppose we have $\leq n-i+1$ threads at level $i-1$. Assume the contrary, ie: there are $n-i+1$ at level i. Let T_A be the last thread to write waiting ie: waiting[i].
That is, for any other thread at level i, call it T_x, the write by T_x into waiting array happened earlier than the write for T_A. Since stage is written before waiting, stage[T_x] = i is written before waiting[i] = T_X, so then if waiting[i] = T_A is written, then T_A sees stage[T_x] ¿ i but

12 Bakery Algorithm

Kind of like the ticket algorithm.

```
enter(id){
   turn[id] = max(turn) + 1
   for(i=0; i<n; i++){
      if(i == id) continue;
      while(turn[id] != 0 && turn(id) >= turn[i] );
   }
}
```

Here, the max operation must be atomic. We must also worry about integer overflow here.

13 Hardware Locks

Here, we use hardware instructions to build locks. Depending on your CPU, these may or may not exist.

The easiest of these is `test-and-set` (TS). The way it works is by getting a value, checking it, and setting it all in one operation.

```
   TS(x, y){//set x = y and return old value of x.
      temp = x;
      x = y;
      return temp;
   }
```

All of this is atomic, implemented by the hardware.

So building a lock out of this is easy:

```
lock:
   init:
      int lock = 0
   enter:
      while(TS(lock, 1) == 1);
   exit:
      lock = 0;
```

See how simple it is? There are some drawbacks though, TS() is kind of a heavy operation to be using in a loop like that.

`Teat-and-test-and-set` solves this issue. We just read the lock value over and over until the values is 0, then, if it is 0, we do test and set. This saves doing the full operation over and over.

```
enter(id){
   while(lock);
   while(TS(lock, 1) == 1){
      while(lock);
   }
}
```

We could also add exponential backoff. That is, before trying again, we wait longer and longer until we get the lock. This is to release the CPU for a small amount of time so that we don't do tons of unnecessary work.

`Fetch and add` is basically just increments a value, and returns the original.

```
FA(v c)
 temp = v
 v = v + c
 return temp
```

We could use this to make the ticket algorithm

We also have `Compare and Swap`. This sets a variable to a value if it already is a certain value.

```
CAS(x, a, b)
   if (x != a)
      return false;
   else:
      x = b;
      return true;
```

Could also implement a lock with this.

14 Queue Locks

We have two different queue locks.

M.C.S: lock an arbitrary number of threads using an explicit queue.

```
class Node{
   Node next;
   boolean locked;
```

```
}

global pointer //one per lock
static Node tail; // keep track of the end of the queue
Node me = new Node(); //each thread has its own node, including the current one

enter(){
   me.next = null ;
   Node pred = TS(tail, me); //Replaces the tail with me
   if(pred != null) { //if equal means theres nobody ahead of us!(in the critical
       section)
      me.locked = true;
      pred.next = me;
      while(me.locked); //spin
   }
}

exit(){
   if(me.next == null){ // if theres nobody after me, doesn't necessarily mean
       nobody is going to be there. There might be another thread waiting to get
       in
      if(CAS(tail, me, null)){ //check if I am the tail
        return
      }
       else{
        while(me.next == null); //wait for other person to finish trying to enter
        }
   }
   me.next.locked = false;
   me.next = null;
}
```

This method is first-come-first-served. Considered cache friendly since only 1, potentially 2 threads are spinning at a time. But, it uses both TS and CAS (need hardware that supports). The other drawback is that the exit also has a spin. It would be nice to just be able to leave without waiting.

15 CLH Lock

```
class Node{
   boolean locked;
}//notice, no next field

global ptr
```

```
   static Node tail = new Node();
   //each thread has a node ptr and a predecessor
   Node me = new Node();
   Node myPred = null;

enter(){
   me.locked = true
   myPred = TS(tail, me) // get the tail, swap with ourself
   while(myPred.locked);
}

exit(){
   me.locked = false
   me = myPred
}
```

This is again first come first served, and solves many of the problems MCS introduced. Only needs one hardware instruction, and doesn't spin on exit.

16 Java Lock Design

We want the synchronized operation to be fast. Unfortunately, this is not always possible. We must make tradeoffs. So, the most common case is prioritized at the expense of the uncommon.

Common cases (to uncommon):

- Locking an unlocked object
- Shallow recursive locking
- Deep recursive locking
- Shallow contention. (Already locked, but no other thread waiting).
- Deep contention (Very rare).

So from here, we see that we should try to optimize low-contention cases, since they are much more common. Often, very few threads are stuck on a lock at one time.

We want to keep this low-overhead since implementing a queue lock for each object would be heavy. The solution here is "Thin Locks". This is when we reserve a small amount of memory in each object. 24 bits, to be exact. These 24-bits are divided into 3 pieces. 1 shape bit, 15 ownerId bits, and 8 for a count. Only the owner of a lock can modify this piece of memory. The locking count keeps track of how many times we've locked the object, -1. -1

to have 0 as being locked once, allowing for 1 extra lock. This supports recursive locking up to 256 deep.

The main idea is that a thread locks an unlocked object by doing a CAS(lockword, 0, id<<IDOFFSET) where the IDOFFSET is to place the number in the correct bits in the 24 bit word. On success, we will now own the lock. Our id will be placed in the lockword. On failure, we check the shape bit to verify it is a 0. (If shape is 0 means its a thin lock). We then check to see if the ID is ours. If so, then we are in a recursive locking situation. All we need to do is increment the count. (Note that this count may overflow).
If the shape bit was a 1, then the remaining 23 bits are actually a pointer to a p_thread mutex. (Or rather an index into a table of mutexes)

Transitioning to a fat lock is done by the owner. If the count is overflowed, all we need to do is allocate a new mutex, and write the lock into a fat lock. If we didn't own it and we're about to overflow the count, will spin and wait to acquire the lock before transitioning to a fat lock.

An extension to this is the "Tasuki" lock, which avoids spinning by allowing lock "deflation" by transitioning fat locks back to thin locks by using more space.

17 Blocking

So far, we've mostly dealt with spin-locks. The advantage to those is that the threads are always ready and can continue as soon as the lock is ready. The drawback is that the CPU gets used up a lot from the unnecessary checks. For long-spinning situations, it's better if the thread was never even scheduled.

Blocking solutions can be more efficient in certain scenarios. Here, instead of spinning, the thread goes to sleep. The intention is that whoever leaves the lock will wake up the sleeping threads.

```
enter(){
   if locked then:
      sleep()
}

exit(){
   sleeping.first.wakeup()
}
```

But what if one thread comes and tests the condition, sees that its locked, and just before sleeping another thread comes and releases the lock. Now, the first thread still goes to sleep, but the lock may never be woken up, since the other thread has already called the wakeup

function. This is known as the "lost wakeup" problem.

17.1 Semaphores

A semaphore consists of a value that is an integer (≥ 0), and two atomic operations: up, down.

```
P() {//"down"
   while(s == 0){
      sleep()
   }
   s -= 1
}

V() { //"up"
   s = s + 1
   wakeup()
}
```

Note that there are no guarantees who gets woken up.

17.1.1 Binary Semaphores

This is a semaphore with values of 0 or 1. Is essentially a mutex. Starts at 1, and locking = P() and unlocking = V().

What if it starts at 0? Well this is called a signaling semaphore. If T0 does a down on s, then some other thread is going to do an up on s and wake it up. Not necessarily a lock, but more of a "wait for another thread".

17.1.2 Counting Semaphores

Here, the value can be arbitrary.

We can now view this as a resource counter. By initializing to some value n we can say that n threads can access at one time.

HEre is the classic producer consumer problem

```
buffer[N];
semaphore empty;
semaphore full;

int pIndex = 0;
int pfilled = 0;
```

```
Producer(){
   while(true){
      d = produce()
      P(empty); //Down on the empty spaces
      buffer[pIndex] = d
      pIndex++ % N;
      V(full); //Up on the filled spaces
   }
}

Consumer(){
   while(true){
      P(full);
      d = buffer[cIndex];
      d++ % N;
      V(empty);
      consume(d);
   }

}
```

EXERCISE: Multiple producers, consumers variations.

17.1.3 Semaphore Drawbacks

The issue with semaphores is that it's easy to forget a P() or V(), which can lead to deadlock.

The more major is that they conflate two behaviors. They can be a counter / signal, and they can be used for mutual exclusion.

18 Monitors

The idea is to have an abstraction on mutual exclusion, by packaging data and operations together.

```
Monitor{
   private data....

   f1()...
   f2()...
}
```

All these operations are mutually exclusive and only one thread can execute any of the monitors methods at one time. The data inside should only be accessed by the monitors methods. (Note that Java does not have monitors, but we can make them very easily.)

With monitors, the signalling is done with condition variables. These conditional variables have wait() and notify() operations that allow doing an up/down. The desired behavior here is that we enter the monitor, check some state, and realize we cannot continue. However, rather than leaving the monitor, we simply call condvar.wait(). This will release the lock on the monitor, and put the thread to sleep. Once some other thread enters the monitor, changes some state and wakes up the other. However, once the other is woken up, it must still wait until all other threads have left he monitor in order to acquire the lock. So *Waking up does not mean running.*

In order to avoid a lost wakeup, our lock release and sleep are atomic. So as soon as we release the lock, we go to sleep immediately.

Monitors have two queues. One queue for the threads that are trying to acquire the monitor lock, and another for the threads waiting on a condition variable.

```
enter():
   if we can acquire the lock go in
   otherwise add to monitor queue

wait():
   add myself to condvar queue
   go to sleep and exit the monitor


exit():
   wake up first person in monitor queue
```

SOME STUFF GOES HERE
AAAAAAa
AAAAAAAA

Suppose we have two queue's, one *mq* for all threads trying to aquire a lock, and *cq* all threads waiting for a condvar. Along with these two queues we will have 5 more atomic instructions:

```
enter(T)
   while(anyone else in monitor){
      add T to mq and put T to sleep
   }

exit(T)
   take a thread out of mq and wake it up

wait(T)
   add T to cq
   release monitor and put T to sleep
```

```
notify()
   take a thread from cq and add it to mq

notifyAll():
   move everything from cq to mq
```

EXPLAIN THIS

18.1 Java stuff

In Java concurrency, every synchronized block has a single, unnamed condvar. (Note that we access condvars in Java with Object.wait() and Object.notify())

So, we cannot specify which thread to wait/notify. This is specified by the queues.

18.1.1 Spurious Wakeup

Threads can be woken up without actually being notified! (Why) This is because threads sometimes wakeup, then run some code to check if they were signaled, and go back to sleep. This is why we must use while() when checking a condvar, and not if() because the while() will ensure that the wakeup was not spurious.

Suppose we have a buffer. T0 comes along and wants to remove from the buffer. It finds it empty, and so goes to sleep. T1 comes and adds to the buffer. It then notifies. T0 wakes up, but then before it is able to get the lock, T2 comes and removes the data. T0 then sees the buffer as empty and goes back to sleep. As we can see, T0 woke up without actually doing anything. An if() block would allow T0 to continue in its code, even though the buffer is empty, when in fact it should have slept.

19 Readers and Writers Problem

Suppose we have a large database. Writing to the DB should be an exclusive action, but reading from the DB should not. Makes sense, since reading can only fail if writing is concurrently happening, and writing can fail if another writer is in the same place. So, we treat readers as a group or class, and writers as individuals. (keep track of each writer separately).

So, either 1 writer accesses the DB or N readers are accessing at any time.

```
int readers = 0
Mutex r = 1, rw = 1 //r is for modifying readers
```

```
//rw is to ensure mutual exclusion

writer():
   while(true){
      down(rw);
      write_something();
      up(rw);
   }

reader():
   while(true){
      down(r);
      readers++;
      if(readers == 1){ //first reader is coming in, lock the writers for all
          other readers
         down(rw);
      }
      up(r)

      read_something()

      down(r);
      readers--;
      if(readers==0){//last reader to leave, releases writer lock
         up(rw);
      }
      up(r);
   }
```

Consider this variation:

```
int nr, nw, wr, ww
//nr and nw are for tracking number of readers and writers.
//wr and ww are for tracking the number of waiting readers and writers
Mutex e

CV okread, okwrite

Reader(){
   lock(e);
   if(nw > 0 || ww >0){ //If there are any working or waiting writers, we wait.
      wr++;
      wait(e, okread);
   }
   else{
      nr++; //otherwise we work on the database.
   }
   unlock(e);
```

```
   read_something()

   lock(e)
   nr--; //remove ourselves from the current readers
   if(wr == 0){ //if we are the last reader
      if(ww > 0){ //allow waiting writers to write.
         ww--;
         nw=1;
         notify(okwrite);
      }
   }
   unlock(e);
}


Writer(){
   lock(e)
   if(nr > 0 || nw > 0 || wr > 0){ //anyone working or readers waiting?
      ww++;
      wait(e, okwrite);
   }else{
      nw++ //add self to currently working list
   }

   unlock(e)
   //writes to DB

   lock(e);
   nw--;
   if(wr != 0){
      nr = wr; //send all the waiting readers to be allowed to read.
      norifyAll(okread);
   }else if(ww >0){//no waiting writers, so wake up a writer
      ww--;
      nw=1;
      notify(okwrite);
   }
   unlock(e);
}
```

WHAT ARE THE BENEFITS OF THIS APPROACH OVER THE OTHER? Here, we give preference to readers over writers, since only the last reader checks to see if there are writers, and the writers add readers first.

The problem with the above code is that we use an if() statement we are susceptible to

spuring wakeups as before. There also exists a solution with only 1 condvar which you can try as an exercise.

20 Deadlock

To get a look at deadlock, we first look at a classic problem.

20.1 Dining Philosophers

There are 5 people sitting at a table. There are 5 chopsticks placed between them, $\{p_1, \dots p_5\}$

```
while(True){
   think();
   get hungry;
   get 1st chopstick;
   get 2nd chopstick;
   eat;
   release chopsticks;
}
```

How can we ensure that all philosophers can eat without getting "stuck"?(They need two sticks to eat)

There are many solutions, here's one.

```
think()

P(mutex)
eats
V(global)
```

21 Dealing with Deadlock

There are a few approaches to dealing with deadlocks. In the Dining philosophers problem, we can use a global lock, but this reduces concurrency since then only one philosopher can eat at a time. We fcan have one lock per chopstick, but this leads to deadlock. Another is to use a global and a per-resource lock, however this has a lot of unnecessary locking. In general, just throwing more locks at the problem is not a good idea.

Consider this apporach:

```
P(global)
P(f_i)
P(f_{i+1 mod n} )
```

```
V(global)

Eat

V(f_{i+1 mod n})
V(f_i)
```

This doesn't lead to the best use of concurrency, but fixes the deadlock issue. Suppose P_0 grabs the global lock, then grabs fork 0. Then grabs fork 1. Releases global. Then P_1 grabs the global lock. Then tries to grab fork 1, but ends up stuck, since P_0 still has the fork. Eventually P_0 will finish and all will be fine, but any other philosopher who tries to grab the global lock will get stuck since P_1 holds the lock. So it's possible to have one philosopher blocking all the rest which is not ideal.

We could randomize lock acquisition, by randomly deciding whether to pick up the right or left fork first. But relying on a random choice involves some luck. Deadlock is less likely, but not eliminated.

We could allow only 4 philosophers to sit at the table at the same time, (even if there are 5 plates), and make the chairs the resource.

```
P(seats)
P(fi)
p(f_{i+1 mod n})
eat
V(f_{i+1 mod n})
V(fi)
V(seats)
```

Of course the issue here is that it puts a hard cap on the amount of concurrency.

Another solution is to acquire and release. ie: if we get stuck, we give up the resource. This requires that we can detect lock failure.

The most useful one to know about is based on ordering resources. This is a general strategy for eliminating deadlock. Everyone acquires the locks in the same order. (A partial order). So if you want 3 and 5, you need 3 before 5, etc.
This also breaks the dependency cycle and avoids deadlock.

21.1 Conditions for Deadlock

Aka Coffmans Conditions. If any one of these conditions is broken we can avoid deadlock.

- Serially reusable resources

- Incremental acquisition (processes acquire multiple resources one by one)
- No preemption. (once a resource is acquired, it doesn't let go involuntarily)
- Dependency cycle

21.2 Livelock

Imagine two trains coming to a crossing of tracks. Each driver sees the other and waits for the other to proceed. They keep saying "you go first" forever and nobody moves. Since the two are still doing something it's not deadlock, but not making progress so livelock.

22 Misc Things To Know About Threading

22.1 Thread Termination

How do we actually stop threads? Asynchronously stopping a thread is considered dangerous because we don't really know what the thread is currently doing. If it's in the middle of a critical operation, we could really mess something up. Especially if the thread is in the middle of a system call or locking mechanism.

In Posix threads, we have a cooperative mechanism for stopping threads. (We ask a thread to stop and the thread must "agree"). This is known as thread cancellation. We simply set a flag in the thread, and it will check the flag once in a while (at safe points) and exit if needed. We don't check too often since that's inefficient, but we should set "cancellation points". In general, system calls that wait tend to be cancellation points (Any system call that returns EINTR (interuptable)). Note that mutex_lock is NOT a cancellation point. You can add your own cancellation points by using `pthread_test_cancel()`.

Java has thread interrupts, which are essentially the same. We can call `thread.interrupt()`. This is checked in any place that can throw an InterruptedException (ie wait, sleep, join. and NOT synchronized.) We can use thread.isInterrupted() to check ourselves.

22.2 Priorities

Recall that Java has nominally priority preemptive scheduling. Java has 10 priority levels, Pthreads have some options such as `SCHED_RR` and `SCHED_FIFO` each with 32 (separate) levels, and sometimes more. But at the OS level, say WindowsNT has 7 levels. How do the priorities get arranged? The final priority can be hard to figure out. The advice is generally to not use priorities, since it's not very reliable.

Suppose we have 3 threads of high, medium and low priority, and one core. Suppose the low priority thread acquires a lock. Then, a high priority thread starts running, (kicking out

the low priority) and tries to acquire the lock. Someone else is already holding it so high prioirty goes to sleep. Then suppose the medium priority starts running. Then if it runs for a long time, it blocks the low priority from releasing the lock, and therefore preventing the high priority from executing.

There are two solutions to this.

Priority **inheritance** is that a thread holding a lock temporarily acquires the priority of the highest thread waiting for that lock.

The other is priority **ceilings** in which locks have an associated priority, and the holder acquires the priority of the lock. Similar to the previous solution with some subtlety.

22.3 Thread Specific Data

We know that threads have local data. Registers, and stack data is all private.

We also have access to static variables and errno. Errno stores the result of system calls. In a multithreaded context, this is a problem, since they'll both be modifying and accessing errno.
Thread local storage, aka Thread specific data allows having the same variable to hold different values for different threads. For example:

```
class Foo {
   static ThreadLocal x = new ThreadLocal();
}
```

There, two threads can set x to different values and each have their own version of what x is. Note that these are slow, and so should be used sparingly.