COMP 409 Study guide

Francis Piché

April 26, 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]); //If both check at same time both will see false
   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 //If both set at same time will cause livelock
   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.

4.2 Petersons Algorithm

```
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.

4.3 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, (there's 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 threads 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:

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, and 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 then T_A cannot enter stage i, since waiting[i] = T_A and stage[T_X] $> i$.
Thus we have that at each level $i + 1$, there are at most $n - i + 1$ threads.

11.2 FCFS

The above is starvation free but is not FCFS. Which thread is served first depends heavily on the order in which the stages are acquired.

We can make it FCFS by defining what first come first server really is:
A locking protocol can be broken down into a finite "doorway" sequence (finite in length) and an arbitrarily long spin sequence.
For FCFS to be respected: if Doorway(A) is before Doorway(B) then the critical section of A is run before the critical section of B.

11.3 Ticket Algorithm

This is a FCFS algorithm in which each thread "grabs a ticket" then goes to "sit down" until their ticket is called.

```
init(){
   now_serving = 0;
   int turn[n]; //each thread's number
   next = 0;
}

enter(id){
   //Get your ticket
   turn[id] = next; //must be atomic with the increment
   next++;

   //Wait for your ticket to be called
   while(turn[id] != now_serving); //spin
}

exit(id){
   //Tell the nex thread their ticket is being served.
   now_serving++;
}
```

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 could then use that as a lock.

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); //Atomically get the tail, and then update it to be
       me
   if(pred != null) { //if pred is null then we can enter, otherwise we must wait
      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, if yes then we can
          safely exit
         return;
      }
       else{//Otherwise someone else is trying to enter
         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 signaling 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 several queues. One queue for the threads that are trying to acquire the monitor lock, and another for each condition variable keeping track of the threads waiting on each condition variable.

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

exitMonitor():
   wake up first person in monitor queue

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


notify():
   take thread for condvar queue and add to the monitor queue

notifyAll():
   move all threads from condvar queue to the monitor queue
```

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, $\{p_1, \ldots p_5\}$

```
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.

23 Barriers

Suppose you have an array with which you want to populate with data (using many threads). Then, using many threads you want to do some computation on that data. You would need for all the threads to wait for the others to finish populating before you can start computing.

A barrier is a type of lock that does that. One way to implement this for 2 threads would be with semaphores:

```
sem1 = semaphore(0);
sem2 = semaphore(0);

//Thread 1
V(sem2); //Allow other thread to start now that I'm ready
P(sem1); //Block myself if the other has not signalled he's ready.

//Thread 2
V(sem1);
P(sem2);
```

For n-threads:

```
volatile int numThreads = n;
fetch-and-add(numthreads, -1); //atomic decrement
while(numthreads != 0) yield();
//The above solution works only once. What if we want to reuse the barrier?


//WHAT NOT TO DO
class Barrier{
   int numThreads = n;
   synchronized void await(){
      numThreads--;
      if(numThreads == 0){
         numThreads = n;
```

```
            notifyAll();
        }else{
            while(numThreads != 0){
                wait();
            }
        }
    }
}
```

The reusable barrier above doesn't work since we do not wait for all threads to leave the barrier before resetting it. This is a problem since if a second phase uses the same barrier, threads still stuck on the first phase would block threads at the second. Also, if there are some threads waiting inside, and we reset, then they will wakeup and immediately see that the numThreads is n and not 0, and will go back to sleep.

To fix this, we need a sense-reversing barrier.

23.1 Sense-Reversing Barrier

The idea behind this is to toggle between phases.

```
int numThreads = n;
boolean phase = true;
boolean phases[n]; //all false initially

if(fetch-and-add(numThreads, -1) == 0){
    numThreads = n; //reset
    phase = phases[id];
}else{
    while(phase != phases[id]) //spin;
}

phases[id] = (!phases[id]); //toggle the phase for this thread.
```

The above workse by ensuring that sequential phases use opposite flags in their wait condition. This avoids the deadlock issue from before.

24 A Note on Hardware Primitives

So far we have: TS, FA, CAS. Turns out we can implement each one by using the others.

```
CAS(x, a, b){
   bool rc; //Value to be returned on success.
   while(TS(castlock, 1) == 1); //spin
   if(x != a){
      rc = false;
   }else{
      rc = false;
      x = b;
      castlock = 0;
      return rc;
   }
}
```

Note that this can take an arbitrarily long amount of time and is not fault tolerant. If any of the other threads die we can get problems. Also note that this uses spinning. CAS should have a finite cost.

24.1 Consensus

If we have n threads, each start with a different value, and we want all of them to agree on the value, we need:

- Consistency: all agree in the end
- Valid: the agreed value is one of the starting values
- Wait-free: finite time and fault-tolerant;

So here each thread would call:

```
int decide(int input){

}
```

and we expect a consensus to emerge.

We can sometimes solve this using our primitives. Only sometimes since it depends on the number of threads involved. We call the number of threads we can reach a consensus with the **consensus-number**.

With basic R/W atomicity, the consensus number would be 1. Long proof in textbook.

Suppose we have a boolean problem of 0 or 1, and only 2 threads. The concensus value in the end will be 0 or 1. We can't solve 2 consensus with only atomic read/writes.

With TS (or FA) we can do:

```
int decide(int v){
   x = TS(decider, v); //decider is a global value initialized to false
   if(x==1) return v;
   else return x;
}
```

So TS and FA have a consensus number of 2. Turns out CAS has a consensus number of ∞.

```
int decide(int v){
   CAS(decider, FALSE, v);
   return decider;
}
```

25 Scheduling

While scheduling is mostly up to the OS, there is still the idea of scheduling and fairness within our applications. Consider this program:

```
volatile boolean go=false;
-------------
Thread 0

while(!go){
   KILL_TIME
}

somethingImportant();

--------------
Thread 1

go=true;
```

On a single core machine with time-slicing this will work. But without time-slicing we could schedule T_0 since, from the CPU point of view, it's doing useful work, and never schedule T_1. So we are left up to the OS to decide whether our application works. No bueno.

We need some fairness guarantees.

25.1 Fairness

Unconditional Fairness is when every unconditional atomic statement that is eligible to execute eventually does. (A conditional atomic statement is of the form: `await(b) c` meaning we execute `b` only if `c` is true.)
For example:

```
-------T0-------   ------T1------
lock();              lock();
while(!go){          go = true;
   unlock();        unlock();
   killtime();
   lock();
}
unlock();
something();
```

Weak Fairness is when a scheduling system is unconditionally fair and every conditional atomic action *eventually* executes, assuming its condition becomes true and remains true.
For example, suppose in the above program we change T1 to be:

```
while(true){
    lock();
    go = true;
    unlock();
      .
      .
      .
    lock();
    go = false;
    unlock();

}
```

Then T0 may never run if the timing is "unlucky" since it might always see go as false. This is not weak fairness, since the condition does not stay true.

String Fairness is when a scheduling policy is unconditionally true, and any conditionally atomic action is eventually executed assuming its condition is true *infinitely often*. (Often enough that you end up seeing it eventually).

In practice, we generally see weak fairness, and rarely strong fairness.

26 Linearization

A concurrent program is *correct* if it is equivalent to some sequential execution. We must be able to know that it is possible for a program to be correct.

However, operations may interleave in a way that is not always equivalent to a sequential execution, which seems to break correctness.

If we assume each operation takes place at a single moment in time, for example:

```
push(){
 .
 . //random stuff
    happens that
    doesn't affect the
    data structure
 .
 . <- moment when the
    push actualy takes
    effect
}
```

then this moment is called the **linearization point**. Then non-atomic things can be thought of as atomic. It's then easier to show (at a higher level) the interleavings of operations. We could prove without linearization points but it becomes much more difficult since we would have to consider every minute detail such as function calls, return statements etc.

What if we can't think of execution as interleaving? Modern hardware has tricks to improve performance that can go outside the sequential interleaving paradigm.

26.1 Write Buffers

One such hardware trick is to not always write values to main memory immediately. We instead store a buffer of writes, since one write may overwrite another. For example if we write `x=5` and then `x=3` there's no point writing `x=5` into main memory. Then, at some point in the future we flush the buffer to main memory. When reading values, we then first check the write buffer before reading from main memory.

Each processor would have its own write buffer. This is a problem for our interleaving idea since we can have two processors with different ideas of what the current values of things are. For example:

```
-=P0-- --P1--
x=1    y=2
b=y    a=x
```

Then its possible for (a, b) = (0, 0), if each processor reads into a, b before the write buffers are flushed. This result is not possible in sequential execution since for a=0, then y=2 must have already been committed, and so its impossible for b=y to see 0.

27 Memory Models

27.1 Memory Consistency

These are paradigms which define which writes are visible between which reads.

Strict Consistency is when the order is the order we wrote with. Ie: atomic interleavings.

Sequential Consistency (most popular) defines a total order of operations. ie:

```
--T0-- --T1--
A();   E();
B();   F();
C();
D();
```

Then A must be before B before C before D and E must be before F.

Coherence is when we have SC per variable. Each variable has its own sequential timeline.

```
--T0-- --T1--
x=1    a=y
x=2    b=x
y=3    c=x
```

Here, it's possible to get (a, b,c) = (3, 1, 2) or (3, 1, 1) since they respect the per-variable timelines but not the overall. This is not possible in SC since if a=3, then x=2 must have been executed after x=1, and so its impossible that b=1 AND c=2. It would have to be that b=c.

PRAM (processor consistency) is when each processor has its own timeline.

```
--T0-- --T1--
A()    X()
B()    Y()
C()    Z()
```

A before B before C, and X before Y before Z is defined, but nothing can be said about A before X, X before B or any cross-processor combinations.

27.2 Relationships Between Models

$$SC \Rightarrow PRAM$$

$$SC \Rightarrow Coherence$$

$$SC \Rightarrow x86 - TSO$$

$$Coherence \nRightarrow SC$$

$$Coherence \nRightarrow PRAM$$

$$Coherence \nRightarrow x86 - TSO$$

$$PRAM \nRightarrow Coherence$$

$$PRAM \nRightarrow SC$$

$$PRAM \nRightarrow x86 - TSO$$

$$x86 - TSO \Rightarrow Coherence$$

$$x86 - TSO \Rightarrow PRAM$$

$$x86 - TSO \nRightarrow SC$$

28 Data Race Detection

Data races / race conditions **occur at runtime**. This means that we cannot always find them through static analysis of the code. Even if the code shows that there could be one, it's not always the case.

```
T1                  T2
do{                 do{
  r1=x                r2=y
}while(r1==0)       }while(r2==0);
y=42                x=42
```

Does this have a data race? Even though one thread is reading while the other writes (which usually causes a data race), we see that there isn't actually one. Each thread mutually waits for the other to finish its loop. The write never gets executed **at the same time** as the read.

29 ABA

The ABA problem occurs when:

- T1 read value A.
- T2 modifies A to value B, then back to A again
- T1 reads value A, and assumes nothing has changed.

But something DID change! The ABA problem appears often when trying to implement lock-free data structures:

30 Lock Free Concurrency

A lock free program:

- Cannot cause suspension in another thread (non-blocking)
- Guarantees system-wide progress

Note that non-blocking program is also wait-free if each thread is also guaranteed progress.

The idea behind this is that blocking is expensive.

Lock free concurrency is achieved using lock-free data structures

30.1 Lock-Free Stack and ABA

A lock-free stack is a concurrent data structure that allows multiple threads to modify it.

```
Lock-Free Stack:

void push(Node n){
   do{
      Node t = top; //top of stack
      n.next = t; //Next of the new node points to the top

   }while(!CAS(top, t, n)); //repeatedly try to swap top with the new node.
}

Node pop(){
   Node t, n;
   do{
      t= top;
      if (t==null){
         throw new EmptyException(); //stack is emptry
      }
      n= t.next; //New top is the next of the old top
   }while(!CAS(top,t,n))
   return t;
}
```

This is lock-free because if a thread fails to push or pop only if there was an infinite number of calls. (Which can't happen, eventually progress will be made).

This implementation suffers from the ABA problem.

```
Initially, stack contains x->y->z.
T0              T1
pop(){
t=x
```

```
t.next = y
//preempt t0 for t1
                    pop()
                    t=x
                    t.next=y
                    CAS(top, x,y);

                    pop()
                    t=y
                    t.next=z
                    CAS(top, y, z);

                    free(y); //free thee memory for y

                    push(x);
                    t = z
                    x.next = z;
                    CAS(top, z, x);
                  //preempt t1 for t0

//T0 resumes where it left off:
CAS(top, x, y);//Succeeds because t1 put x back to the top
//Now the stack looks like: y->x->z
```

But now, if we try to access the top of the stack, we will crash because that location (y) has been freed. All of this is because t1 took away x, then y, then put back x, tricking t0 into thinking that nothing had changed.

Note that if we use garbage collection, we don't have this problem, since T0 still has reference to y. Once T1 finishes its pop, y can be garbage collected as it "catches up" to the changes of T2.

30.2 Avoiding ABA

30.2.1 LL and SC

Load-linked, stored conditional. These are two operations to allow us to construct tests to ensure a memory address has not been written, rather than relying on value-based tests.

In Power-PC assembly, these tests are given by:

```
lwarx(x){
   temp = x;
   //x is reserved by the processor
   return temp;
```

```
}

stwcx(x, y){
//Check if someone else has reserved the variable, and set the new value.
   if(x is still reserved for us){
      x=y;
      remove reservation;
      return true;
   }
   return false;
}

redo.t(){
   lwarx r1,0,x
   addi r1,r1,1
   stwx r1,0,x
   bne redo.t
   //redo if reservation failed
}
```

Reservations are lost if someone else does LL/SC, or if x is written. If we used this instead of CAS we can avoid ABA problem.

Java doesn't give LL/SC, we instead use AtomicStampedReference which provides a "wide" compare and swap which increments a stamp every time the value is changed.

30.3 Elimination

LockFree is limited by sequential CAS calls. We can attempt to speed this up by using elimination.

If a push() is followed by a pop(), the effects cancel out. If we could find concurrent pairs, they could use eachothers values without touching the stack.

For this we use an elimination array or elimination backoff array, Elimination array is when push and pop meet in the array at the same time, don't go to the stack. Backoff array is when if a CAS failed, go back to the array.

30.4 Lock Free Exchanger Stack

Lockfree exchanger stack has 3 states: EMPTY, BUSY, WAITING.

- First thread comes, sets the state to WAITING.

- CAS to set the state
- CAS to set data

If either cas fails, we give up or try to exchange. If the second thread sees waiting, grab the data: `CAS(<x, WAITING>, <item, BUSY>)`. If the thread sees busy, if we are the first thread, grab the data and change to EMPTY. If exchange fails, go do a regular push/pop.

30.5 Lock Free Linked List

Keep reference to head and tail. Assume a singly linked list.

```
tryAdd(Node n, Node prev){
   n.next = prev.next;
   return CAS(prev.next, n.next, n); //try to insert n after prev
}

tryRemove(Node n, Node prev){
   return CAS(prev.next, n, n.next);
}
```

The above doesn't quite work. Consider:

```
H-x-y-z-T
T1                          T2
CAS(Head.next, x, y);    w.next = y
                          CAS(x.next, y, w);


//Now the list looks like:
x-w
   \
  H-y-z-T
```

So T1 successfully deletes x, but we end up with a fail for the w insert.

Another problem:

```
T0                  T1
CAS(H.next, x, y)   CAS(x.next,y,z)
//T0 delete X       T1 delete y
//Now the list looks like:
  x
   \
H-y-z-T
```

And we see that y was not deleted.

30.5.1 Solutions to Linked List Problems

The solution proposed by Valois in 1995, is to have **auxiliary nodes** between the data nodes:

```
H-o-x-o-y-o-z-o-T
```

Alternatively, Tim Harris proposed in 2001 to **mark nodes as deleted** rather than to delete immediately. Then after some time we would come through and do a "clean-up". This will give operations that look like:

```
tryAdd(Node n, Node prev){
   n.next = prev.next;
   return CAS(<prev.next, mark>, <n.next, false>, <n, false>);
}

tryRemove(Node n, Node prev){
   Node succ = n.next;
   //Mark successor to prevent insertion/deletion
   if(CAS(<n.next, mark>, <succ, false>, <succ, true>)){
      //Delete, but if it fails we don't care, we'll clean up later
      CAS(<prev.next, mark>, <n, false>, <succ, false>);
      return true;
   }
   return false;
}
```

Now lets try adding w after x and removing x again:

```
T0                          T1

//T0 remove x               //T1 add w
CAS(<x.next, mark>,         CAS(<x.next, mark>,
    <y, false>,                 <y, false>,
    <y, true>);                 <w, false>); //Fails if T0 marked y
//Fails if T0 changed x.next
//Now delete
CAS(<H.next, mark>,
    <x, false>,
    <y, false>);
```

And if we do conflicting deletes: (delete x and y)

```
T0                          T1
```

```
//Mark y                    //Mark z
CAS(<x.next, mark>          CAS(<y.next, mark>,
   <y, false>                  <z, false>,
    <y, true>                   <z, true>);
)
//Delete x                  //Delete y
CAS(<H.next, mark>,         CAS(<x.next, mark>,
    <x, false>,                 <y, false>,
    <y, false>)                 <z, false>)
```

Above, only one of the deletes will be successful. That is, if T0 marks y first, then T1 cannot delete y, leaving it behind in the list. (x is still successfully deleted).

We now need a way to cleanup these marked nodes. Might as well do it during a find operation.

```
find(x){
   restart: while(true){
      pred = H;
      current = pred.next;
      while(current != T){
         successor = current.next;
         while(current.marked){//If I am marked, delete my predecessor as long as
             it is not marked.
            if(!CAS(<pred.next, marked>, <current, false>, <successor, false>)){
                //If pred is marked, need to restart
               goto restart;
            }else{
                //Iterate
               current = successor;
               successor = current.next;
            }
         }
         if (current.data == x.data) return current;

         pred = current;
         current = successor;
      }
   }
   return null; //not found
}
```

30.6 Lock Free in General

There exists a **universal construction** that we can apply to many data structures.

Assume your data structure has a **single entry point:**

```
public interface SeqObject{
   public Response apply(Invocation i);
}
```

This apply method is the entry point. For example, a Stack. Invocation would be push, pop functions. Response would be the thing returned by pop.

An invocation modifies the data structure. We could record all of these as a "log" of how the data has changed over time. We could then use this to recreate the state of the object. Works in almost all cases, but very slow. Would have to decide which ops run in which order for concurrent invokes. (**Consensus**). All threads need to agree on which operation to do next.

```
Thread t:
  i = new Invoc(...);
  do{
    j = consensus(i) //magically reach concensus
    while(i!=j){ //Wait for my operation to be chosen
     s=tail of history
     r = s.state //Get the result of the tail

     do{//set the whole state from the beginning
       r = r.apply(s); //apply the result
       s = s.next; //iterate
     }while(s != i);
    }
  }
```

Doing this would require reusable consensus operation, which is doable but awkward. A better way would be to contain the consensus "object" inside the invoke object. Then each invoke has its own version of how to reach consensus. (And thus determine whether it can run or not).

31 Open MP

Open MP is an **overlay language**, on top of C,Cpp and Fortran. It uses `#pragma` directives to control parallelization.

We use it to define parallel fragments:

```
#pragma omp parallel
executeInParallel() // The next statement gets executed by a separate thread.
```

```
printf("%d", omp_get_num_threads()) //Is 1 since there is a join after parallel
    segments

#pragma omp parallel
{
...
}//This whole block runs in parallel


#pragma omp parallel
{
   #pragma omp parallel
   ...
   //Nested parallelism
}

#pragma omp parallel for
for(int i=0; i<100; i++){
   A[i] = i;
}//Nicely formatted for-loops can have each iteration done in a different thread

#pragma omp single
//or
#pragma omp master
//To use only one thread for the next line (can be used in a nested section)

#pragma omp parallel sections
#pragma omp section //this will run on one thread
...
#pragma omp section //this will run on another at the same time
...
```

31.1 Data in OMP

All static data is shared by default. We can indicate shared variables or private variables, where each thread gets its own version of the variables. *These private variables are un-initialized an non-persistent. When the thread dies, they die.*

FirstPrivate means initialize the variable to whatever value is outside this scope.
LastPrivate means *publish* the private value to the outer scope when done.
reduction(ops) means to perform all operation in the list "ops" on each of the values of variables in all threads.

31.2 Synchronization

OpenMP gives the following synchronization techniques:

- Single, master, critical (which thread to run on)
- Barriers
- Atomic operations
- Flushing

The **memory model** for OpenMP is a weak model. Each thread has it's own cache (not a full write buffer).
The **flush** operative lets us flush these thread cache for certain variables. `flush(x,y,z)` will send x,y,z back to main memory and clear from working memory.

If two flush operations have intersecting variables, we need a consensus on who goes first, so that each thread will see the writes to main memory in the same order. If there is no intersection, there are no guarantees about who goes first. Within a single thread, however, if we flush, the value is seen immediately.

Atomic operations have an immediate flush of the variable involved.

```
//Petersons algorithm with OMP

T1:                                T2:
  (A)|atomic(flag[0]=1)            (A)|atomic(flag[1]=1)
     |flush(flag[0])                  |flush(flag[1])

  (B)|flush(flag[1])               (B)|flush(flag[0])
     |atomic(tmp=flag[1])             |atomic(tmp=flag[0])
     |if(tmp==0) enter();             |if(tmp==0) enter();
```

Notice above, blocks (A) and (B) are independent. There is no guarantees about the ordering between them. We could see the results of (B) before the results of (A), which would break the algorithm and mutual exclusion. We could fix this by merging the flushes: `flush(flag[0], flag[1])`. This would force an ordering between them since now thee intersection of the flush is non-zero (both threads are flushing the same values).

32 March 21

Java doesn't have an OpenMP overlay. That doesn't mean people haven't tried. There's some complexities that show up when we try to mesh Java's inner workings with OpenMP.

The first of these issues is Private variables. How would we handle objects? In C++ we have copy constructors that would work, but in Java, do we do a shallow copy? Deep copy?

Java does not have flushing, and has a lot of built-in synchronization to account for.

33 PGAS: Partitioned Global Address Space

Locality is important, since its quite expensive to leave the CPU. It would be better if we could pool the data operated on by one thread for efficiency.

The idea is to divide memory into chunks. Global blocks (which try to not touch others stuff), and also private data that nobody can touch.

33.1 X10

X10 is a PGAS language. It gives some basic concurrency mechanisms

```
async {} //Run the following block asynchronously (may or may not use another
    thread)
//Can have nested async's
x=0
async{
   x=1;
   async{
      x=2;
   }
   x=3;
   async{
      x=4
   }
   x=5
}
x=6

//Execution structure looks like this:

   x=0
  /   \
 x=6   x=1
      /  \
    x=3   x=2
   /  \
 x=5  x=4

//Example
for(i in 0...100){
   async{a[i] = a[i]+1}
```

```
}
//At this point any number of the iterations could have occured (no join happens)
```

The **finish** operative specifies that we must wait for all other asyncs in this block to finish before proceeding.

```
finish{
   for(i in 0...100){
   async{a[i] = a[i]+1}
}
\\At this point the array is completely done being modified.
```

This relates to PGAS because *each async block runs in its own address space.*

The **at(p)** operator allows specifying which space to run the statement. This pauses execution in the main thread. The idea is to send the operation **to** the data. Still need to copy data over.

34 Tasks

Suppose we have a collection of tasks (pool of tasks), instead of having one thread per task, we could have a few threads which will go and collect the tasks and execute them. Java gives several types of ThreadPool constructs.

- Executors: potentially launch a new thread to execute a task)
- DirectExecutor: Single threaded execution of tasks
- AsynchExecutor: Create a new thread to execute a runnable
- ExecutorService: Similar to executors but can return data.

With an executor service, can pass in a callable and it will execute it. A `ThreadPoolExecutor` is used by specifying a fixed number of threads, and submitting a bunch of tasks.

A `future` is task that may or may not execute in parallel, but that we can decide to get the value from in the future. If we call `future.get()` and the task is done, we get the value. Otherwise we block until the result is ready. So you should put your futures far enough in your program that you expect the task to be done.

```
public class Future{

   Callable c;
   public Future(Callable c){
      this.c = c;
   }

   public Object get(){
```

```
      return c.call();
   }
}
//This version is non-parallel. Could implement parallel version using threads
    and join.
```

34.1 Dependencies

If tasks have dependencies between them, we can express this as a graph. This will be a directed acyclic graph (DAG). The order of the choices of which tasks to execute first affects performance if we have limited threads.

```
A   B   C
| \ |   |
D   E-->F //A before D and E, B before E, C and E before F.
| / |
G   H //D and E before G, E before H, H before I
    |
    I
```

The goal is then to minimize the number of steps it takes to complete all tasks (assume all tasks take unit time to complete).

If we have inifite threads, we can't do better than the longest path. This is known as the **critical path**. We need to make a scheduler.

34.2 Greedy Task Scheduler

The greedy task scheduler randomly picks tasks without much thought. This is the simplest kind of scheduler. It's a fair scheduler.

Performance analysis:

- Assume total time is time to execute all nodes sequentially: T_1
- T_∞ is the time taken for unlimited threads
- Assuming all tasks take unit time, length of longest path is a lower bound on T_∞.

Any particular step, we either make a **complete step** (more tasks than threads), or an **incomplete step** (fewer tasks than threads).

34.2.1 Graham-Brent Theorem

Let T_p be the time taken with p threads. Then:

$$T_p \leq \frac{T_1}{p} + T_\infty$$

The T_∞ term implies we can't have more than the length of the critical path number of incomplete steps, and the fraction means that's how many complete steps we can have.

The greedy scheduler is always within twice the optimal schedule.

Proof. Suppose we know the optimal time T_p^*. By the above inequality, we know:

$$T_p^* \geq max(\frac{T_1}{p}, T_\infty)$$

and

$$\frac{T_1}{p} \leq max(\frac{T_1}{p}, T_\infty)$$

Similarly:

$$T_\infty \leq max(\frac{T_1}{p}, T_\infty)$$

and we know:

$$T_p \leq \frac{T_1}{p} + T_\infty$$

so substituting:

$$\leq 2 * max(\frac{T_1}{p}, T_\infty)$$
$$\leq 2 * T_p^*$$

□

35 Transactional Memory

Locking is hard. If the main goal is simply to maintain mutual exclusion, there are other ways we can do it. There are two ways to make sections atomic.

- Optimistic
- Pessimistic

35.1 Pessimistic

The idea here is to turn all atomic blocks into a `synchronized(global_lock){}` block on a global lock. The problem is that we are unaware of outside modifications, so we would need to put all datarace places inside one of these blocks. This would lead to very low level of concurrency.

A more fine-grain approach would be to find all the data in these atomic blocks and have a lock on each variable. Independent atomic sections would now be able to be done in parallel. The issue with this is we would need to analyze the code at runtime to find all the data. (SLOW!).

```
atomic{
   while(p!=null){
      p.data = 1;
      p = p.next
   }
}
```

Would somehow need to lock all nodes of this linked list, but then if there is 1000 nodes, would have 1000 locks. This is very slow. We would need to run the loop ahead of time to find out which places need the locks (again slow). We would also need to avoid deadlock in these situations, so we would need either a trylock or impose an order on the acquisition of locks, which further slows things down.

35.2 Optimistic

Most of the time contention doesn't happen. So we could try to avoid locking all together. To enforce correct execution we would:

- Execute optimistically with no locks
- Detect if data was modified while executing transaction
- If all data untouched then we're ok!
- If any reads/write occurred need a strategy to deal with it.

For the last point, we could **keep a log** of all transactions, and then *revert back in the case of a concurrent modification.*

A better way would be to *not write to main memory until we know the transaction succeeded.* We keep both a read and a write buffer for each atomic block. Then we use the buffers to detect conflicts, and only write to main memory if there was no conflicts.

We run into some language issues however. How would we handle nested transactions? How do we get the wait/notify behavior? For the latter question, we could try to expose failures allowing you to retry. Not quite the same as wait-notify but similar. Nesting would be useful for this, as we could do:

```
atomic{
   atomic{
      if(...){
         retry; //keep waiting until this specific condition passes
      }
   }
}
```

35.3 Hardware Support

35.3.1 HLE: Hardware Lock Elision

We get the `XACQUIRE` prefix for other commands. This action is a locked action itself. It notes the memory address used, and executes subsequent code transactionally. If nobody conflicts with the transaction code, success. `XRELEASE` marks the end of the transaction section. If fail, **retry but now use the lock**

```
XACQUIRE:LOCK:...
a=y
b=x
c=a
XRELEASE
```

35.3.2 RTM: Restricted Transactional Memory

Here we get:

```
XBEGIN<offset> //offet is the code line of what happens on a failure

XABORT<reason> //Jump to failure handler, giving a reason for the failure

XEND

XTEST //Am I in a transaction?
```

Transactions can fail when:

- We have conflicts (this is done per cache-line, so could have unecessary failures if non-conflicting data is on the same cache line)
- XABORT
- buffer overflow (transaction too big)
- Special instructions (pause)
- Interrupts
- Task swapping

36 Thread Level Speculation

Thread level speculation is an attempt at automated parallelization. What if we could auto-magically turn a sequential program into a parallel one?

```
 ___
| A | //TLS Chooses a fork and join point
 ___  <-Fork point
| B |
 ___  <-Join point
| C |
 ___
```

The join point is where we merge with the speculative thread. The speculative thread executes in the future. It may not execute correctly, so we need to hide it's data until we can verify that the state at the end is correct. This has a lot of overhead.

C is executed as a transaction. If we eventually see that C depends on B, we must discard it (since its state is incorrect).

37 Message Passing