COMP 409 Study guide

Francis Piché

January 24, 2019

Contents

Part I
Preliminaries

1 Disclaimer

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.

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.

4 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.

5 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.

5.1 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.

5.2 Conditions

6 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).

6.1 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.

7 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.

8 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.

9 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.

10 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.