

Section: Thread Cooperation and Synchronization

1 Announcements

1. Design Docs due next Monday
2. Design Review Schedule posted soon, check back about when you can sign up
3. **Any questions**

Disclaimer: These notes aren't substitutes for attending lecture or reading the book. Those have been created with careful revision and review. These notes are mainly intended to be a more thorough index of the lectures and reading so that you can go off and read the full treatment on the topic and be able to ask the right questions while reading those two sources. If you have comments or suggestions please send me email and I'll fix them up.

2 Recap

Any Questions?

First and foremost I want to spend some time doing a quick recap on all the stuff before and hopefully use it to motivate this weeks material. Instead of a boring and verbose description I think it'll be more beneficial to go through an exercise that covers a lot of material.

Exercise: *We have on many occasions and in section and lecture referred to the fact that while one thread is doing I/O and asleep another thread can be running. How does this happen? Let me formalize a few things first. Lets say we have three threads: Thread A is I/O dominant however it has to do minimal computation to figure out what I/O to issue next (think database). Threads B and C are long running scientific code with no I/O and is compute bound (think Matrix-Matrix Multiplication). In addition lets also say for the sake of argument, the threads have already been initialized and we are interested in some time frame in which no startup/finish housekeeping need be done. Lets also assume that in order to issue an I/O operation the thread must make an explicit call into the OS while an interrupt signals the completion of that I/O event. We also have a timer interrupt system to help with preemption of threads. If Thread A is initially running, work out the various transitions of Threads A, B, and C through the life cycle diagram. In addition also work out what the stack looks like for each thread at each stage as it transitions through the cycle and what specific signals trigger the transition.*

3 Intro

I'm going to reiterate that Chapter 6 does a great job of explaining this stuff so I'll say go read it! It has a lot of interesting details. So this section my primary aim is to go over some of the concepts that are very difficult and actually go through them a lot slower and encourage you to read the book for the details. The best way to simulate the code in your head is ask yourself "if I put a yield call after every line of assembly code will my multi-threaded code still work?"

4 Thread Cooperation

So far we have gone into decent detail about how threads are created and spawned as well as the lifecycle of a thread. We have often hinted that the advantage of threads over processes is the ability to share memory now that they share the same memory space. But let us talk a little bit more about this. There are many pitfalls that can happen here so I'll just go through one example from the book and then lets think motivate why we need thread synchronization. Take a look at Algorithm 1 and Algorithm 2.

Algorithm 1 Algorithm for producer

```
1: while true do
2:   while counter == BUFFER_SIZE do
3:     do nothing
4:   end while
5:   buffer[in] = nextProduced
6:   in = (in + 1)%BUFFER_SIZE
7:   counter ++
8: end while
```

Algorithm 2 Algorithm for Consumer

```
1: while true do
2:   while counter == 0 do
3:     do nothing
4:   end while
5:   nextConsumed = buffer[out]
6:   out = (out + 1)%BUFFER_SIZE
7:   counter --
8: end while
```

Notice in lines 7 of both Algorithm 1 and Algorithm 2 we are incrementing and decrementing the *counter* variable.

Because of **concurrent** execution the instructions of Algorithm 3 and Algorithm 4 can be interleaved in any order.

Question: If the value of *counter* is initialized to 5 before any thread starts, what are the possible values of *counter* with an arbitrary interleaving of Algo-

Algorithm 3 sample assembly translation of counter++

1: $register_1 = counter$
2: $register_1 = register_1 + 1$
3: $counter = register_1$

Algorithm 4 sample assembly translation of counter --

1: $register_1 = counter$
2: $register_1 = register_1 - 1$
3: $counter = register_1$

Algorithm 3 and Algorithm 4? Do not assume anything about the order in which the instructions can be interleaved.

Preventing this type of interaction and protecting ourself from the arbitrary interleaving is what *Thread Synchronization* is all about. This is one of those things that requires hands on practice to truly understand the nuances. The best way to learn is to practice so I encourage you to do some of the exercises in the back of the chapter.

5 Critical Sections

I won't go too much into the definition because the book does a good job explaining it and I encourage you to read it. There is one thing I will stress though is the three requirements for the critical sections.

1. Mutual Exclusion: Only one process can be running its critical section at any given time. Notice that when we say critical section we don't prevent other threads from running code and context switching in the middle of this. All our design needs guarantee is that the other threads running will not be affecting the shared state that the critical section is protecting. There is a subtle difference here, but it's important.

Question: This is obviously a performance improvement, what performance reasons motivate us to allow threads to run code unrelated to this critical section while we are in our critical section? Think of examples

2. Progress: Here I think the book makes an interesting point about the remainder section but I think it is explained a bit poorly. What we want the guarantee is that if there are threads that are waiting to enter a critical section, then those threads will decide amongst themselves who gets to run next. And we also want to make sure that every thread gets to run at some point. This doesn't mean that different threads can't run more frequently than others, we just want to make sure our algorithm for picking can pick a thread. The idea of the remainder section is a thread who has finished the critical section but is not waiting on a lock. So we merely say that thread can't be involved in the process of selecting a next thread. Only threads that are ready to enter critical sections and have no work to do before entering the critical section may enter.

3. Bounded Waiting; Again related to progress and we touched on it. Bounded waiting merely says that the any thread waiting for a critical section must get a chance to run eventually in a bounded time.

6 Synchronization Mechanisms: Hardware

The book then talks about Peterson's Solution. I think this actually a very interesting section because it'll test your knowledge until that point. If you can understand what that code is doing and how it controls a critical section you're in great shape.

Exercise: *Go through Peterson's Solution in the book and how it works and then try to work through about what changes (if any) you'd need to make for three threads.*

6.1 Test and Set

This is one of the cornerstones of creating hardware that can handle multi-threaded code. The test-and-set is a new instruction that will be added to the MIPS Instruction Set Architecture (ISA). Test and set is given a memory location to check. It *atomically* returns the old value of that location and sets it to true. This might seem counter intuitive but think of it working with an other thread that atomically sets that memory location to false. Figure 6.5 in seventh edition of the book shows a code sample that works with this. Notice that we can create protect a critical section with this through spinning on a while loop.

Question: **Does the implementation in Figure 6.5 satisfy all the correctness properties listed above? If so explain why, if not explain which one it fails and how you'd fix it**

6.2 Swap

Swap is similar to test and set but atomically swaps two locations in memory. See Figure 6.6 in the 7th edition for the actual code. Again the same spin lock in Figure 6.5 can be implemented using the swap function.

Exercise: *Given one atomic primitive you can provide correct implementations for the other primitives. (a) Give a correct implementation of test-and-set assuming that we only have swap as an atomic. (b) Give a correct implementation of swap assuming that we only have test-and-set as an atomic*

6.3 Others

There are many other hardware primitives that aren't mentioned in the book that are used on modern machines which include

1. fetch-and-add: like test-and-set but instead of setting the value of the memory location to 1 we increment the value and return the old value
2. compare-and-swap: this an instruction that atomically compares the contents of a memory location to a given value and, if they are the same, modifies the contents of that memory location to a given new value.

3. load-link/conditional-store: load a value from memory and note what the value is. When it comes time to write the value back we atomically do the following: if the value in memory has changed from our value that we saw in the load-link do nothing, else set the new value in memory to the value we are writing.

Don't worry too much about these, I just threw these in so you'd get an idea of some of the other primitives available in hardware.

Exercise: *If you want to truly get a grasp of how these are all related, try implementing each of these with one of the others (i.e. implement fetch-and-add with swap or test-and-set, etc). What is the best hardware primitive? BTW this is much more of a CS152, CS252, and CS258 type of question*

7 Synchronization Mechanisms: Software

If nothing else, the material from the previous section should have shown that it is a pain to try to work at the level of hardware primitives and debugging a bad implementation is equally painful. Like the age old tradition of Computer Science, we want to abstract all this away into cleaner packages. I'll only mention the higher level semantics of the these packages. (I'll leave it as an exercise to figure out how to implement them with the hardware mechanisms described above.) Lets take a look at some of the most commonly used packages.

7.1 Locks

Locks are simple wrappers around hardware abstractions. Normally locks provide two functions.

- acquire: When we call acquire we are basically in a race with the other threads for the ability to enter a critical section. Once one thread successfully grabs this lock we are guaranteed that the thread that "won" the race will be the only one running that critical section until it releases the lock. Note that another thread can switch in, but if we have properly protected our data, then the other threads that are affecting shared state related to this critical section will be held at the acquire state.
- release: Once we are done with a critical section we release the lock which basically throws the lock up for grab for any thread in the acquire. Then another thread grabs the locks and gets to run the critical section.

7.2 Semaphores

These are my personal favorite since I've used them on many occasions in parallel code. The advantage of semaphores is that they are pretty simple to understand. They are primarily used in applications that require a producer / consumer relationship between multiple threads. There are two operations that can be performed on a semaphore:

- wait (aka. decrement or $P()$ ¹): We wait on a semaphore to be greater than 0. Once it is we decrement it. Note that the process of discovering the semaphore to be greater than 0 and then decrementing it must be done atomically.

Question: Why must the discovery of the semaphore being greater than 0 and the decrement be done atomically?

- signal (aka. increment or $V()$ ²): This atomically increments the integer bound to the semaphore.

Note that a naive implementation of the semaphores will lead to busy waiting so we must think of a way to make sure that we don't busy wait.

Question: How would you design a semaphore not to busy wait using `sleep()` and wait queues?

(Hint:) Look in `Semaphore.java` in the Nachos source code ;-). The reason that I like this construct a lot is that in the general case when there are no synchronization problems the code is fast. The synchronization code dynamically kicks in once a race is about to be detected. Isn't that cool?! However, because there is no strict enforcement of the ordering of increments and decrements in semaphores, you can get some really nasty bugs. Thus people like Dijkstra and Hoare thought that semaphores were "obsolete."

7.2.1 Protecting critical regions with Semaphores

Here let's go through a quick example of how we can use a semaphore to protect a critical section of code. The classic problem with semaphores is the bounded buffer. The problem description is as follows:

- suppose that two threads are in a producer/consumer relationship. Thread P (the producer) creates an object and puts it in a buffer and then tells the consumer that an object is ready for consumption. Thread C (the consumer) then reads that object off the buffer and then processes it. The question is, what happens when we have a fixed buffer size and we can't afford to lose any objects from being consumed. We need to make sure that we can add to the buffer only when there is at least one empty slot and only take stuff off the buffer when there are a nonzero number of objects on the buffer.

Instead of trying to redo the algorithm here I refer you to figures 6.10 and 6.11 in the seventh edition of the book. In that example there are 3 different semaphores used: one to signal the "full" event, one to signal the "empty" event, and one to assure that only one thread can access the buffer at any point in time and avoid corruption. I'd strongly recommend going through this example in the book and understanding it.

Question: Can we implement this algorithm with 2 semaphores instead of 3? If not, why not? If so, how?

¹ $P()$ comes from the dutch *proberen* which means "to test." The names are in Dutch since this was another great and simple concept invented by Dijkstra

² $V()$ comes from the dutch *verhogen* which means "to increment"

Question: Notice that the mutex signals and waits are nested within the other semaphores. What happens if they are not strictly nested? What does the nesting avoid?

(Hint:) There is a word in computer science devoted to this ugly thing.

7.3 Monitors

Understanding the previous discussion section will lead us to the motivation for thinking of an even more abstract and simple model of Monitors. Monitors are nothing more than an abstraction from doing stupid things with semaphores and locks. Remember that in parallel code the probability that we'll do stupid things and have them crop up at the wrong time is very very high. ³ The more that we can protect ourselves the better off we are. According to the book "a monitor presents a set of programmer-defined operations that are provide mutual exclusion within the monitor." A monitor is a program abstraction that we get to create with underlying synchronization mechanisms, such as locks and semaphores. Once we have these mechanisms and the abstraction we can just use the abstraction and not have to deal with the mess of getting parallel code right. If these seems abstract, its meant to be there isn't anything concrete to say here. Take a look at the book for examples on how to construct different types of monitors.

7.4 Condition Variables

Condition variables are part of the Nachos project so I want to go over them. Condition variables take in a Lock object and provide three fundamental operations:

- sleep: This means we have the lock and we want to give it to someone else. We thus atomically release the lock and go to sleep. (this is where disabling and re-enabling interrupts become important)

Question: Why must the release and sleep be atomic? How does Nachos guarantee this atomicity?

- wake: we are asleep and another thread has woken us up. Once we wake up we know that the lock is ours
- wakeAll: same as wake but there is a free-for-all here. We just put all the sleeping threads on the ready queue and let them fight for the lock. The thread that actually wins this fight will have the lock and the rest will be put back to sleep.

Question: What happens when multiple condition variables share the same lock?

The advantage of condition variables is that it allows us to put threads on the wait queue in a dormant state rather than having them spin wait in the ready/running states.

³I had parallel code break on me in Nachos only in the auto-grader. I was never managed to reproduce it on my machine.

Condition variables allow a clean mechanism for us to realize when we can wake different threads up and thus use the wait state to write code that is not only correct but leads to good performance.

8 Priority Inversion and Donation

We'll go into this in much more detail later but its important for the project. I encourage you to read Emil Ong's description of what priority inversion is. It'll provide an important backdrop into the project. The PDF is available on my website and I encourage you to download and read it. It is a really simple example that illustrates the problem very well.

9 Conclusion

There were a lot of things in this section but the fundamental high level idea is that when we writing multi-threaded programs we want multi-threaded programs to effectively share data and run in parallel. However to make sure that we don't run into race conditions we need to protect critical sections of the code and serialize those sections. There are many hardware and software mechanisms that provide tools on which we can build to assuage the difficulty but its still a very hard problem. The indeterminism of the entire thing is what really lets you shoot yourself in the foot. So careful design is a must here. I'll enforce this idea of careful design in your design docs. ASK QUESTIONS!