Operating Systems
COMS W4118
Lecture 15

Alexander Roth

2015 – 03 – 12

1 Spin Locks

- Checks a signal variable over and over.
- Wastes CPU cycles.

1.1 Using `yield()` with Sleep Locks

- We can replace the empty `while` loop with a call to the function `yield`.
- `Yield` is not defined.
- There must be some notion of a variable and we must let the operating system put other processes to sleep.
- The operating system must have some way of knowing who is waiting on which variable.
- We need some data structure that keeps track of which processes are waiting on the lock.
- If the operating system were to wake up all the processes at once, there would be a huge overhead from all the context switches and only one processes would get the lock.
- There is no knowing when we will get a turn.
- Thus, **starvation** is a real possibility.
- Bounded waiting is not satisfied.
- There must be some notion of a `wake` function and some idea of a queue that contains the processes.

- Each lock will have a queue associated with it that holds all the waiting processes.
- Thus, there are processes waiting in a first come, first serve fashion.
- Having a `flag` variable has the notion of a lost wakeup and a starved process.
- Starvation is the hardest condition to satisfy with locking.
- Oftentimes, the condition will be sacrificed in order to get the system working.

1.1.1 Fixing Sleeping Locks

- `mutex` structure that has a flag and a queue.
- `Lock`: function acquires a lock by setting the flag, otherwise it puts itself into the queue.
- `Unlock`: if queue is empty, set the flag ot zero or dequeue and wakeup the process we took out of the queue.
- Flag does not get reset when we take a process off of the queue.
- To avoid losing wake ups, we implement a spin lock with a guard variable.
- Inside the `yield` function, we need to check that we are dequeued already.
- You need to recognize when a race condition is possible.
- You need to make the area that holds the spin lock as small as possible.
- Spin lock must happen in the kernel, and should only happen in the cases where you are doing something very short.

2 Readers-Writers Problem

- Old, but common paradigm. You have to read a lot, but sometimes you have to write.
- We have a single lock and we lock and unlock in different modes.
- Lock and unlock in either read or write mode.

2.1 Implementing Reader-Writer Lock

- For read lock, we have a counter that notes how many readers lock at a time.
- When one reader has a lock, all the other readers will keep going.
- When you increment the number of readers, we need to lock and unlock while we increment to maintain atomicity.
- Synchronization is not easy.
- The issue with this implementation is that the writer may starve if there is a constant stream of readers.
- Writer can only activate until all the readers have moved through.
- We introduce another lock called `writer`, that locks around a `write_lock` call in order to acquire the original lock.
- Using one lock for two purposes is typically a bad idea.

3 Semaphore Motivation

- There are some things a simple `mutex` cannot do.
- Sometimes you want two threads, one to prepare a data structure and one to wait until the first is done.
- You want to impose order.
- Mutexes do not provide order; semaphores do.
- Semaphores can handle the producer/consumer problem.
- A semaphore is some synchronization variable that contains an integer value.
- The integer value cannot be accessed directly.
- Binary semaphore is similar to a mutex, while a counting semaphore has an integer value more than 1.
- You can post over the initial value of the semaphore.
- The initial value is not the maximum value; you can extend past it.

3.1 Producer-Consumer Problem

- We have a fixed-bound buffer that contain a mix of data and nothing.
- Producer has a pointer to the next empty slot.
- Consumer has a pointer to the first filled slot.
- Multiple consumers will consume an item in each slot and move the pointer to the next slot.
- If consumer reaches the end of the buffer, it wraps around to the beginning of the buffer, so the buffer is a circular buffer.
- The buffer is filled when Producer is right behind the Consumer pointer.
- The buffer is empty when the Consumer pointer is right behind the Consumer counter.

3.1.1 Producer-Consumer Implementation

- We create two semaphores: `full` and `empty`.
- Producer waits on the empty semaphore until empty is non-zero. If there is no empty spots, then we cannot produce.
- Once empty becomes a positive number, then there is a slot open in the array.
- Producer will then decrement empty, and fill the slot, finally increment the number of the full semaphore to say there is a newly filled position.
- Consumer will do the opposite of the producer function.
- If full is zero, then there is no item in the buffer; otherwise, empty the slot and increment the empty semaphore.
- Need to synchronize the movement between the empty and full semaphores.
- Thus, we create a new guard semaphore during the fill slot and empty slot phases.
- Needs to be guarded to avoid multiple consumers and producers creating a race condition.
- The guard provides the mutual exclusion property.

4 Monitors and Condition Variables

4.1 Montiors

- Monitor is the fusion of synchronization and object-oriented programming.
- Every java object is a monitor.
- It is an object-oriented class where every method is a synchronized method.
- Every method you call has to grab the lock and release said lock at the end of the method.
- Every object has hidden variables, which is a mutex and N-many conditional variables.