Synchronization Tools

Andrés Ponce

November 9, 2020

> The process of synchronization involves avoiding race conditions as programs change the data in processes.

A *cooperating process* is one which can change other process's data and which can have its own data changed.

Critical-Section Problem

Processes all have a **critical section**, where they are reading or writing to information shared by other processes. Thus when a program is executing in its critical section, no other process should be executing in its critical section either. The process makes a request to change its critical section in its **entry section**, and after the critical section we can have an **exit section** followed by the remaining code in the **remaining section**.

Thus there are three principles which control the synchronization paradigm:

1. **Mutual Exclusion**: If there is a process P executing the code in its critical section, then no other process can execute the code in its critical section.

2. **Progress**: If there are some processes that wish to enter their critical section, then only those processes which are not in their remainder section can decide which program will enter the critical section.

3. **Bounded Waiting**: If a process makes a request to enter its critical section, then there is a bound on the amount of times other processes can enter their own critical section before the original process can enter the critical section.

Peterson's Solution

This method requires that two processes that are sharing data also share two values : `int turn;` and `boolean flag[2];`.

The turn is set to i when P_i can enter its critical section. The $flag[i]$ value is set to `true` when the process itself is ready to enter the critical section.

We can show that Peterson's solution works, if the `load()` and `store()` operations are not reordered. However, sometimes modern

compilers will reorder the instructions to gain some speed when executing programs. For single-threaded applications this is no biggie, but with multiple cores this reordering could introduce some data races.

Memory Models

On some systems, when memory is changed, it propagates across all processors such that any other processor will be able to see the changes; this is called **strongly ordered** memory. Likewise, there might be memory that does not immediately propagate to other processors or become visible immediately, a.k.a. **weakly ordered** memory.

So another approach is to use atomic instructions, which execute as one uninterrupted block and will execute without being reordered. We could have an atomic function which changes the value of a lock variable, and use that to regulate atmoic sets of instructions.

Mutex Locks

Stands for **mutual exclusion**. Similar to Peterson's solution, the way a process enters its critical section is first by using the `acquire()` method to acquire the lock. If the lock is available, then the process can enter its critical section, otherwise it means the other process is in the critical section. The code looks something like:

```
while (True){
        acquire_lock();
                critical section
        release_lock();
                remainder_section;
}
```

The `acquire()` and `release()` methods are crucial to this operation. In the former function, we have to check if the lock is available, and wait if it is not. At the end we return False. The latter method only sets the **available** variable to True, which will allow other applicant processes to access their critical section. These two must be processed atomically!!

In the function for acquiriing the lock, if the lock is busy, then the program has to loop continuously and wait for the lock to become availbale. This could become a real issue in a multiprocessor system that might require many context switches to fully carry it off. If the lock is only to be held for a *short duration*, then we can use a **spinlock**. [1] The process might wait for the lock to become available on another core [2] and when the lock becomes available, we don't require

[1] Spinlocks are the method of choice for many OSs today.

[2] We say it *spins* on another core while it's waiting

a context switch.

Semaphores

A **semaphore** is an integer variable that can only be modified in two functions: in the `signal()` function and the `wait()` function. Again, these two functions have to be performed atomically, even while we loop in the wait function. [3]

Using the semaphore construct we could also have one section of a program only run after another section of a program already run. If we set a semaphore at the end of P_1, then the beginning of P_2 can have a `wait(synch)` call, where `synch()` is a semaphore.

[3] There are two different types of semaphores: **counting** and **binary**. Binary semaphores only take the value of 0 or 1, and in this way function like mutex locks. Systems that don't offer mutex locks use binary semaphores. Counting semaphores offer a counting variable, and can serve for a resource that has limited instances.

Monitors

There are still some issues with the mutex lock and semaphore approach. For example, omitting the wait and signal function calls could result in some sytem resources being presumed available when in fact they are not, similar to the example in 6.1 about updating a value concurrently and being off by one.

Monitors are a type of abstract data type, where we wrap different user functionality inside a class. This ensures that only one thing within the class is running at the same time, and thus can avoid some of the issues with synchronization.

The monitor can have processes suspend execution based on a condition. We can also use only semaphores to implement the wait and signal procedures. If a condition causes several other processes to become suspended, how do we choose which one to wake up once the execution resumes? Similar to threading, we could have a FCFS approach. Every process that requests a resource could have an associated priority number, such as **x.wait(c)**. Then, when the currently executing process calls the signal function, the number with the highest procedure can execute. This procedure can be an estimate of the time which the program requires to execute.

Liveness

The main idea is how we can ensure that the amount of time a process has to wait to enter its critical section is bounded. If we have a process with an infinite loop, then ggezpz. One such scenario is when two processes are waiting and they can only be caused by the other waiting process executing a signal call, but the other process hasn't entered its critical section. There is also the problem of **priority inversion**. Suppose three processes with priorities L, M, H are exe-

cuting. Process L is currently using a semaphore, so it is executing. Process H has to wait for that semaphore to become available for it to move into execution. However, before that happens M comes in and preempts process L, which means a process with lower priority increased the execuiton time of a process with higher priority.