Synchronzation Examples

Andrés Ponce

November 30, 2020

> There are some other problems with synchronization, sucha as the critical section problem already discussed.

Remember that we had discussed **semaphores**, which were either binary or integer variables used to indicate the availability of a critical section and whether a process could access such critical section. In reality, those could all act as a mutex lock rather than a strict binary semaphore.

Bounded Buffers Problem

Remember this problem refers to the change of some variable by two different processes.

The essence of this problem is that we have a **consumer** and a **producer**. The former program is in charge of producing content to go into one of n buffers.

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
```

The two processes share the above data structures. The problem lies in how to coordinate them so that no data is lost or copied. We should thus avoid having the producer write to the buffer when it is full, and we should also avoid having the consumer consume from an empty buffer.

Readers-Writers Problem

This problem can be illustrated with a database. If multiple processes want to read from the database, then there is no issue, these are the **reader** programs. **Writer** programs might take a look at the database and try to change some of the values in it. As many reader programs can read the database without any issue, but when there is a writer program involved then we need to be careful. When a program writes to the database, we need it to have exclusive access to the database.

There are two variations of this problem: one where we allow readers to read concurrently, and exclude one of the writers programs. Another solution is to give priority to the writer programs so that they write as soon as possible. Both of these solutions result in some

amount of starvation, in the first case the writer is left without access and in the second one the reader is left without access.

A solution might be to keep a `rw_mutex` semaphore, which both readers and writers have to acquire to change the contents of the database. Multiple readers can access this lock, but only one writer program at a time may access it. This approach might be useful in situations where there are more readers than writers, since it would minimize the amount of writers starved and maximize the concurrent readers accessing the data.

Dining Philosophers Problem

Multiple philosophers gathered around a dinner table want to access the bowl of rice. When a philosopher wants to eat rice, she grabs the chopstick and starts eating. A philospher has to grab one chopstick to each of her sides, so she has to wait until the philosopher near to her have put down the chopstick before eating.

The problem here is how do we allocate finite resources among various competing processes without deadlock.

The **semaphore** solution involves representing each chopsitck (the contested resource) , with a semaphore. Thus, when a philosopher wants to pick up her chopstick, it waits for the other one to become available. However, this could result in all chopsticks being picked up at the same time and thus being unavailable. If all philosophers grabbed their left chopstick at once, then all would starve, since they would all infiintely wait for the other chopstick to become available. We could limit the number of philosophers that are available to eat, or ensure that a philosopher only picks up chopsticks when they're both available.

The **monitor** solution presents a deadlock-free solution but only if we pick up the chopsticks if both are available.

Synchronization within the Kernel

Synching in Windows

Windows provdes a spinlock solution when there is some thread waiting to access a global resource. THis is becuase spinlocks might be more efficient for smaller code segments.[1]

In Linux, we can use synchronization by performing some atomic operations. This means that during these sorts of operations the processes will not be pre-empted and interruped by other processes. However, this is mostly seen in counters. More sophisticated locking mechanisms might be required for other scenarios.j

[1] Remember in a spinlock, the waiting program loops until it can access the critical section.

POSIX Synchronization

The POSIX standard uses mutex locks, semaphores, and events to protect ciritical sections of code. Semaphores are just some binary variables that indicate the tatus of some resource.

Pthreads are the main way for the POSIX standard by which we use semaphores. Since we use mutex locks in C to acquire the right to modify the data in some programs, we usually associate some conditions with it. These conditions are called condition variables. If the condition is not true, we can place the variable in a loop and check it periodically.

Alternative Approaches

Recently there has been an increasing need for mutli-threaded applications but designing such applications that are entirely free from race conditions remains a tad hard.

Transactional Memory

Coming straight from database design, there is an approach to memory called **transactional memory**. A transaction is a sequence of atomic operations that are made and then committed once they are completed. Since operations on memory are made atomically, then there cannot be any deadlock.

In this approach, the TM system will automatically take care of seeing which oeprations can be handled concurrently, such as multiple read operations. It is then on the hardware implementation to find how to properly schedule threads so that there is no collision.

OpenMP

Recall that OpenMP is a library where we can schedule things to run concurrently. We can specify some compiler directives a piece of code that is to be executed by a number of parallel threads. The `#pragma omp critical` directive indicates that the following piece of code is to be only executed by a single thread. This is to inform the rest of the system to only allow one thread to execute at the same time. This directive then functions more or less like a mutex lock, so it would still be up to the programmer to handle possible race conditions.

Functional Programming Languages

Imperative languages such as C, Java, and Python are so becuase they maintain a state of all variables and so during sequential execution of instructions, the order of the instructions is crucial.