Chapter 30: Condition Variables

There are many cases where a thread wishes to check whether a **condition** is true before continuing its execution. For example, a parent thread might wish to check whether a child thread has completed before continuing. How can it do that?ergerbert

Text, letter

Description automatically generated

We can use a shared variable in figure 30.2, but it is inefficient because the parent spins and wastes CPU time. We would like a way to put the parent to sleep until the condition we are waiting comes true.

**30.1 Definition and Routines**

To wait a condition to become true, a thread can make use of what is known as a **condition variable**. A **condition variable** is an explicit queue that threads can put themselves on when some state of execution is not as desired (by **waiting** the condition). Some other thread, when it changes said state, can then wake one of those waiting threads and thus allow them to continue by signaling on the condition.

To declare such a condition variable, one simply writes something like this: pthread\_cond\_t c.

A condition variable has two operations associated with it: wait() and signal(). The wait() call is executed when a thread wishes to put itself to sleep; the signal() call is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition. On POSIX, the calls look like this:



The wait call takes a mutex as parameter. It assumes that this mutex is locked when wait is called. The responsibility of wait() is to release the lock and put the calling thread to sleep. When the thread wakes up (after some other thread has signaled it), it must re-acquire the lock before returning to the caller.

There are two cases to consider. The first is normal where the parent creates the child and continues running itself. The child runs and then wake up the parent. The second case is that the child runs immediately upon creation, sets *done* to 1 and wake sleeping thread. The parent then runs, does not wait and returns.

Text

Description automatically generated

Poor implementation will lead to race condition or parent will sleep forever.

**30.2 The Producer/Consumer (Bounded Buffer) Problem**

Imagine one or more producer threads and one or more consumer threads. Producers generate data items and place them in a buffer. Consumers grab said items from the buffer and consume them in some way. Because the bounded buffer is a shared resource, we must of course require synchronized access to it and prevent race condition. To begin to understand this problem better, let us examine some actual code.

Text, letter

Description automatically generated

The idea is that the put routine assumes the buffer is empty and simply put a value to the shared buffer and marks it full by setting count to 1. The get routine does the opposite.

Now we need to write some routines that know when it is OK to access the buffer to either put data into it or get data out of it. The conditions for this should be obvious: only put data into the buffer when count is zero, and only get data from the buffer when count is one.

This work is going to be done by two types of threads, one set of which we’ll call the producer threads, and the other set which we’ll call consumer threads.

**A Broken Solution**

Text

Description automatically generated with medium confidence

With single producer and consumer, this works, but it will not work for more threads. This has to deal with the problem of fairness as some other consumers might get in the way to get the resource while a waiting consumer is sleeping!

There is no guarantee that when the woken thread runs, the state will still be as desired. This interpretation of what a signal means is often referred to as **Mesa semantics**. The contrast, referred to as **Hoare semantics**, is harder to build but provides a stronger guarantee that the woken thread will run immediately upon being woken. Virtually every system ever built employs Mesa semantics.

**Better, But Still Broken: While, Not If**

The simplest way to fix this is changing the if into while. Thanks to Mesa semantics, a simple rule to remember with condition variables is to **always use while loops**.

The problem occurs when a consumer grabs a resource, but which thread should it wake up? If it wakes another consumer up, and since there is no resources, it will go back to sleep. Now, every thread sleeps!

**The Single Buffer Producer/Consumer Solution**

The solution is to use two condition variables, instead of one, in order to properly signal which type of thread should wake up when the state of the system changes.

In the code, producer threads wait on the condition empty, and signals fill. Conversely, consumer threads wait on fill and signal empty. By doing so, we avoid the previous problem.

Text

Description automatically generated with medium confidence

**The Correct Producer/Consumer Solution**

To improve the solution, we add more buffer slots, so that multiple values can be produced before sleeping, and similarly multiple values can be consumed before sleeping.

The first change for this correct solution is within the buffer structure itself and the corresponding put() and get(). We also slightly change the conditions that producers and consumers check in order to determine whether to sleep or not. We also show the correct waiting and signaling logic. A producer only sleeps if all buffers are currently filled (p2); similarly, a consumer only sleeps if all buffers are currently empty (c2). And thus we solve the producer/consumer problem; time to sit back and drink a cold one.

**30.3 Covering Conditions**

When a thread calls into the memory allocation code, it might have to wait in order for more memory to become free. Conversely, when a thread frees memory, it signals that more memory is free. However, our code above has a problem: which waiting thread (there can be more than one) should be woken up?

A simple solution is to wake up all waiting threads. The downside of this is negative performance impact. We call such condition **covering condition** as it covers all the cases where a thread needs to wake up.