Introduction

So far, we've developed the concept of a lock and seen how one may be correctly made using the right hardware and operating system. Regrettably, locks aren't the only primitives required to construct concurrent applications.

There are a lot of situations where a thread wants to check if a condition is true before continuing its execution. For example, a parent thread may want to wait for a child thread to finish before proceeding (this is commonly referred to as a join()); how should this wait be implemented? Take a look at Figure 30.1 for an example.

The following output is what we'd like to see here:

As seen in Figure 30.2, we may try using a shared variable. Although this technique works in most cases, it is wasteful because the parent spins and wastes CPU time. Instead, we'd like to find a mechanism to put the parent to sleep until the condition we're looking for (e.g., the child has finished executing) is met.

It is common in multi-threaded programs for a thread to wait for a condition to become true before proceeding. The straightforward approach of spinning until the condition becomes true is wasteful, wastes CPU cycles, and can be wrong in some instances. So, what is the best way for a thread to wait for a condition?

Definition and Routines

A thread can use a condition variable to wait for a condition to become true. to A condition variable is an explicit queue that threads can join when some state of execution (i.e., a condition) is not as expected (by waiting on the condition); when that state changes, another thread can wake one (or more) of those waiting threads, allowing them to continue (by signaling on the condition). The concept dates back to Dijkstra's usage of "private semaphores"; in Hoare's work on monitors, a similar concept was dubbed a "condition variable."

To create a condition variable like this, write something like this: The command pthread cond t c; declares c as a condition variable (note: proper initialization is also required). The wait() and signal() actions are coupled with a condition variable (). When a thread wants to sleep, it uses the wait() function; when a thread changes something in the program, it uses the signal() function to wake up a sleeping thread waiting on this condition. The POSIX calls are formatted as follows:

For the sake of clarity, we'll refer to these as wait() and signal(). The wait() function also takes a mutex as an input, and it expects that this mutex is locked when wait() is invoked. Wait() is responsible for releasing the lock and putting the calling thread to sleep (atomically); when the thread wakes up (after being alerted by another thread), it must re-acquire the lock before returning to the caller. This complication originates from the desire to avoid certain race situations while a thread is attempting to go to sleep. To further grasp this, consider the answer to the join problem (Figure 30.3).

There are two scenarios to think about. In the first case, the parent thread starts the child thread but continues to execute (assuming we only have one processor) and hence invokes thr join() to wait for the child thread to finish. In this case, it will get the lock, check if the kid is done (which it isn't), and then call wait() to put itself to sleep (hence releasing the lock). The child thread will eventually run, print the word "child," then call thr exit() to wake up the parent thread; this code simply acquires the lock, sets the state variable done, and signals the parent, waking it up.

Finally, the parent will execute (returning from wait() while holding the lock), unlock the lock, and print the final message "parent: end."
In the second example, the child runs as soon as it is created, sets done to 1, calls signal to wake a sleeping thread (which there isn't, so it merely returns), and is finished. The parent then runs, calls thr join(), checks that done is 1, and returns without waiting.

Last but not least, you'll notice that the parent decides whether to wait on the condition using a while loop rather than merely an if statement. While it may not appear to be strictly necessary in terms of the program's logic, it is always a good idea, as we'll see later.
Let's attempt a couple different implementations of the thr exit() and thr join() code to make sure you realize how important each element is. You might be asking if the state variable is required. What if the code looked something like this?

Unfortunately, this strategy is ineffective. Consider the case where the child runs immediately and calls exit(); in this case, the kid will signal, but no thread will be sleeping on the condition. When the parent runs, it will just call wait and remain stuck; no thread will ever be able to wake it up. The importance of the state variable done can be seen in this example; it records the value that the threads are interested in knowing. It serves as the foundation for sleeping, awakening, and locking.

Another bad implementation can be found here (Figure 30.5). We'll pretend that you don't need to hold a lock to signal and wait in this scenario. What kind of issue could arise here? Consider this!

The problem here is a modest racial imbalance. If the parent calls thr join() and then checks the value of done, it will notice that it is 0, and will attempt to sleep. But, right before the phone rings, wait to go to sleep; otherwise, the parent will be interrupted, and the child will flee. The kid sets the state variable done to 1 and signals, but there is no thread ready to be woken up. When the parent runs again, it sleeps indefinitely, which is a terrible state of affairs.

Hopefully, you can understand some of the essential criteria for using condition variables correctly from this simple join example. To make sure you understand, we'll go over a more sophisticated example: the producer/consumer or bounded-buffer problem.
Another bad implementation can be found here (Figure 30.5). We'll pretend that you don't need to hold a lock to signal and wait in this scenario. What kind of issue could arise here? Consider this:

The Producer/Consumer (Bounded Buffer) Problem

The producer/consumer problem, also known as the bounded buffer problem, is the next synchronization problem we'll look at in this chapter. Indeed, it was to solve this producer/consumer dilemma that Dijkstra and his colleagues invented the generalized semaphore (which may be used as a lock or a condition variable); we'll go over semaphores in more detail later.

Consider a scenario in which there are one or more producer threads and one or more consumer threads. Producers create data items and store them in a buffer; consumers take those items out of the buffer and use them in some fashion.
This layout can be seen in a variety of real-world systems. For example, in a multi-threaded web servee, a producer queues HTTP requests in a work queue (i.e., the bounded buffer); consumer threads execute these requests.

When you pipe the output of one program into another, such as grep foo file.txt | wc -l, a bounded buffer is employed. This example runs two processes at the same time: grep writes lines from file.txt containing the string foo to standard output, and the UNIX shell redirects the output to a UNIX pipe (created by the pipe system call). The process wc's standard input is connected to the other end of this pipe, which simply counts the number of lines in the input stream and writes out the result. Thus, the grep process is the producer, and the wc process is the consumer; between them is an in-kernel constrained buffer, and you are simply the happy user in this example.

We must, of course, require synchronized access to the bounded buffer because it is a shared resource, lest a race condition emerge. Let's look at some actual code to get a better understanding of the situation.

The first thing we'll need is a shared buffer, which a producer can put data into and a consumer can take data from. For simplicity, we'll only use a single integer and the two inner functions to put a value into the shared buffer and retrieve a value out of the buffer (you could absolutely use a pointer to a data structure instead). For more information, see Figure 30.6.

Isn't it simple? The put() procedure thinks the buffer is empty (and verifies this with an assertion), then simply writes a value to it and sets count to 1 to indicate that it is full. The get() procedure accomplishes the inverse, returning the value after emptying the buffer (i.e., setting count to 0). Don't worry about the one entry in this shared buffer; we’ll extend it later to a queue that can contain several entries, which will be even more entertaining than it sounds.

Now we need to develop some procedures to determine when it is safe to use the buffer, either to put data into it or to retrieve data from it. The prerequisites should be self-evident: When count is 0 (i.e., when the buffer is empty), only put data into the buffer; when count is one, only get data from the buffer (i.e., when the buffer is full). We have done something incorrect if we implement the synchronization code so that a producer sends data into a full buffer and a consumer obtains data from an empty one (and in this code, an assertion will fire).

This job will be carried out by two sorts of threads, one known as the production threads and the other as the consumer threads. For example, figure 30.7 shows the code for a producer that cycles an integer through the shared buffer several times and a consumer who obtains data from the shared buffer (forever), printing out the data item it retrieved from the shared buffer each time.

A Broken Solution

Consider the case where there is only one producer and one consumer. Because put() modifies the buffer and get() reads from it, the put() and get() routines both have crucial sections. Putting a lock around the code, on the other hand, isn't enough; we need something more.

That something more, predictably, is a set of condition variables. We have a single condition variable cond and associated lock mutex in this (broken) initial try (Figure 30.8).

Let's take a look at how manufacturers and customers communicate. A producer waits for the buffer to be empty (p1–p3) before filling it. The consumer follows the same logic as the producer but is looking for a different condition: fullness (c1–c3).

The code in Figure 30.8 works with just a single producer and a single consumer. However, if there are more than one of these threads (for example, two consumers), the solution has two major flaws. What exactly are they?

Let's look at the first issue, which involves the if statement before the wait. Assume there are two producers (Tc1 and Tc2) and two customers (Tc1 and Tc2) (Tp). A consumer (Tc1) starts first; it gets the lock (c1), examines whether any buffers are ready for consumption (c2) and if none are, waits (c3) (which releases the lock).

The producer (Tp) then takes off. It gets the lock (p1), checks if all buffers are full (p2), and if they aren't, it goes ahead and fills the buffer (p4). After that, the producer indicates that a buffer has been filled (p5). This is important because it shifts the first consumer (Tc1) from a condition variable to the ready queue, allowing Tc1 to run (but not yet running). After that, the producer keeps going until it realizes the buffer is full, at which time it sleeps (p6, p1–p3).

Here's where the issue arises: another consumer (Tc2) slips in and eats the buffer's single value (c1, c2, c4, c5, c6, skip- ping the wait at c3 because the buffer is full). Assume Tc1 is running, and it re-acquires the lock right before returning from the wait. It then calls get() (c4), but no buffers are available to be consumed! An assertion is triggered, indicating that the code has not performed as expected. Tc2 crept in and drank the one value in the buffer that had been produced, indicating that we should have somehow blocked Tc1 from attempting to consume. Figure 30.9 depicts the action taken by each thread over time, as well as the state of its scheduler (Ready, Running, or Sleeping).

This occurs because the constrained buffer changed once Tc1 was woken (by Tc2) but before Tc1 started (by Tc2). Signaling a thread only wakes it up; it is thus a clue that the world's state has changed (in this case, that a value has been placed in the buffer), but there is no guarantee that the state will remain as desired when the woken thread runs. Mesa semantics is the name given to this interpretation of what a signal means, after the first study that built a condition variable in this way; Hoare semantics, on the other hand, is more difficult to construct but provides a stronger guarantee that the woken thread will run immediately after waking up. Mesa semantics are used in almost every system ever designed.

A little better: While instead of If

Fortunately, changing the if to a while is a simple remedy (Figure 30.10). Consider why this works: now, consumer Tc1 wakes up and (while the lock is held) promptly re-checks the shared variable's state (c2). The consumer just goes back to sleep if the buffer is empty at that point (c3). In the producer, the result if is also modified to a while (p2).
An easy tip to remember with condition variables is to always use while loops. You don't have to re-check the condition all of the time, but it's always a good idea to do so; just do it and be happy.
However, this code still has a fault, which is the second of the two issues listed above. Is it visible to you? The fact that there is only one condition variable has anything to do with it. Before you continue reading, try to figure out what the issue is.

When two consumers (Tc1 and Tc2) run initially, and both go to sleep, the problem develops (c3). The producer then dashes off, writes a value to the buffer, and wakes up one of the consumers (say Tc1). The producer then cycles back (releasing and reacquiring the lock along the way) and tries to add more data to the buffer; but, because the buffer is full, the producer instead waits for the condition to occur (thus sleeping). Finally, two threads are sleeping on a condition, and one consumer is ready to run (Tc1) (Tc2 and Tp).

Tc1 then wakes up by returning from wait() (c3), re-checking the condition (c2), and consuming the value when the buffer is full (c4). This consumer then sends a signal to the condition (c5), awakening only one sleeping thread. Which thread, though, should it awaken?

Because the consumer has depleted the buffer, it is evident that the producer should be notified. However, we have a problem if it wakes the consumer Tc2 (which is probably conceivable, depending on how the wait queue is managed). Tc2 will wake up, see that the buffer (c2) is empty, and go back to sleep (c3). Tp, the producer, is now awake and reading this section of the book. When two consumers (Tc1 and Tc2) run initially, and both go to sleep, the problem develops (c3). The producer then dashes off, writes a value to the buffer, and wakes up one of the consumers (say Tc1). The producer then cycles back (releasing and reacquiring the lock along the way) and tries to add more data to the buffer; but, because the buffer is full, the producer instead waits for the condition to occur (thus sleeping). So, two threads are sleeping on a condition, and one consumer is ready to run (Tc1) (Tc2 and Tp). Things are going to get exciting, and we're ready to cause a problem!

To put into the buffer, he's left to sleep. Tc1, the other consumer thread, goes back to sleep as well. The glaring problem is that all three threads are left sleeping; see Figure 30.11 for the painful step-by-step of this horrible disaster.

Signaling is unquestionably necessary, but it must be more targeted. Consumers should awaken only producers, and producers should not awaken consumers.

Producer/Consumer Single Buffer Solution

The solution is simple: instead of using one condition variable, use two to properly communicate which sort of thread should wake up when the system state changes. Figure 30.12 shows the code resulting from this approach.

Producer threads wait for the condition empty in the code while signals fill. Consumer threads, on the other hand, wait for fill and then signal empty. By doing so, the second difficulty is avoided by design: a consumer can never wake a consumer by accident, and a producer can never wake a producer by accident.

When checking for a condition in a multi-threaded application, a while loop is always correct; but, depending on the semantics of signaling, an if statement may be. As a result, always use while to ensure that your code behaves as expected.

While loops are used around conditional checks to manage the case of spurious wake-ups, due to implementation quirks in some thread packages, two threads may be woken up when only a single signal is sent. Consequently, wake-ups that aren't expected are another cause to double-check the condition a thread is waiting for.

The Right Producer/Consumer Solution

We now have a workable producer/consumer solution. However, it isn't completely generic. We make the final adjustment to increase concurrency and efficiency by adding extra buffer slots, allowing several values to be produced and consumed before sleeping. This strategy is more efficient with only a single producer and consumer because it avoids context shifts; with numerous producers or consumers (or both), it even permits concurrent producing or consuming, boosting concurrency. Fortunately, it is only a minor modification to our current solution.

The first adjustment required for this right solution is within the buffer structure and the put() and get() functions (Figure 30.13). We also tweak the conditions that producers and consumers evaluate while deciding whether or not to sleep. Finally, we also demonstrate good waiting and signaling logic (Figure 30.14). A producer sleeps only if all buffers are currently filled (p2), and a consumer sleeps only if all buffers are currently empty (p1) (c2). And with that, we've solved the producer/consumer conundrum; now it's time to relax and enjoy a cold beverage.

Covering Conditions

We'll take a look at another example of condition variables in action.

The issue they encountered is best demonstrated with a simple example, in this instance, a multi-threaded memory allocation library. A code snippet demonstrating the problem is shown in Figure 30.15.

When a thread calls into the memory allocation procedure, as seen in the code, it may have to wait for more memory to become available. When a thread frees memory, it indicates that additional memory is available. On the other hand, our code has a problem: which of the waiting threads (there can be multiple) should be woken up?

Consider the case below. Assume there are zero bytes available; thread Ta calls allocate(100), followed by thread Tb, which calls allocate(50) (10). Because there aren't enough free bytes to satisfy either of these requests, Ta and Tb wait for the condition and go to sleep.

Assume that a third thread, Tc, calls free at that point (50). Unfortunately, when it calls signal to wake a waiting thread, it may not wake the right waiting thread, Tb, which is just waiting for 10 bytes to be freed; Ta should stay waiting because there isn't enough memory available yet. As a result, the code in the figure fails because the thread that wakes up other threads has no idea which thread (or threads) to wake up.

Lampson and Redell's solution is simple: in the code above, replace the pthread cond signal() call with a call to pthread cond broadcast(), which wakes up all waiting threads. We promise that any threads that need to be woken will be woken this way. The consequence, of course, is that we may unnecessarily wake up many other waiting threads that shouldn't (yet) be awake, resulting in a performance hit. Those threads will simply wake up, re-evaluate the situation, and then return to sleep.
Such a condition is known as a covering condition by Lampson and Redell since it covers all the circumstances when a thread needs to wake up (conservatively); the consequence, as we've seen, is that too many threads may be roused.

The astute reader may have also recognized that we could have used this strategy earlier (see the producer/consumer problem with only one condition variable). In that situation, though, a superior solution was available, and we chose to employ it. In general, if your application only works when you switch your signals to broadcasts (even though you don't think it should), you've got a flaw; fix it! However, broadcast may be the most straightforward option in other circumstances, such as the memory allocator described above.

Summary

Sample content New Page