What we'd like to do is to create a single abstraction that can be used to address all our synchronization needs.

In the early 1960's, the Dutch computer scientist Edsger Dijkstra proposed a new abstract data type called the semaphore, which has an integer value greater than or equal to 0.

A programmer can declare a semaphore as shown here, specifying its initial value.

The semaphore lives in a memory location shared by all the processes that need to synchronize their operation.

The semaphore is accessed with two operations: WAIT and SIGNAL.

The WAIT operation will wait until the specified semaphore has a value greater than 0, then it will decrement the semaphore value and return to the calling program.

If the semaphore value is 0 when WAIT is called, conceptually execution is suspended until the semaphore value is non-zero.

In a simple (inefficient) implementation, the WAIT routine loops, periodically testing the value of the semaphore, proceeding when its value is non-zero.

The SIGNAL operation increments the value of the specified semaphore.

If there any processes WAITing on that semaphore, exactly one of them may now proceed.

We'll have to be careful with the implementation of SIGNAL and WAIT to ensure that the "exactly one" constraint is satisfied, i.e., that two processes both WAITing on the same semaphore won't both think they can decrement it and proceed after a SIGNAL.

A semaphore initialized with the value K guarantees that the i_th call to SIGNAL will precede (i+K)_th call to WAIT.

In a moment, we'll see some concrete examples that will make this clear.

Note that in 6.004, we're ruling out semaphores with negative values.

In the literature, you may see P(s) used in place of WAIT(s) and V(s) used in place of SIGNAL(s).

These operation names are derived from the Dutch words for "test" and "increase".

Let's see how to use semaphores to implement precedence constraints.

Here are two processes, each running a program with 5 statements.

Execution proceeds sequentially within each process, so A1 executes before A2, and so on.

But there are no constraints on the order of execution between the processes, so statement B1 in Process B might be executed before or after any of the statements in Process A.

Even if A and B are running in a timeshared environment on a single physical processor, execution may switch at any time between processes A and B.

Suppose we wish to impose the constraint that the execution of statement A2 completes before execution of statement B4 begins.

The red arrow shows the constraint we want.

Here's the recipe for implementing this sort of simple precedence constraint using semaphores.

First, declare a semaphore (called "s" in this example) and initialize its value to 0.

Place a call to signal(s) at the start of the arrow.

In this example, signal(s) is placed after the statement A2 in process A.

Then place a call to wait(s) at the end of the arrow.

In this example, wait(s) is placed before the statement B4 in process B.

With these modifications, process A executes as before, with the signal to semaphore s happening after statement A2 is executed.

Statements B1 through B3 also execute as before, but when the wait(s) is executed, execution of process B is suspended until the signal(s) statement has finished execution.

This guarantees that execution of B4 will start only after execution of A2 has completed.

By initializing the semaphore s to 0, we enforced the constraint that the first call to signal(s) had to complete before the first call to wait(s) would succeed.

Another way to think about semaphores is as a management tool for a shared pool of K resources, where K is the initial value of the semaphore.

You use the SIGNAL operation to add or return resources to the shared pool.

And you use the WAIT operation to allocate a resource for your exclusive use.

At any given time, the value of the semaphore gives the number of unallocated resources still available in the shared pool.

Note that the WAIT and SIGNAL operations can be in the same process, or they may be in different processes, depending on when the resource is allocated and returned.

We can use semaphores to manage our N-character FIFO buffer.

Here we've defined a semaphore CHARS and initialized it to 0.

The value of CHARS will tell us how many characters are in the buffer.

So SEND does a signal(CHARS) after it has added a character to the buffer, indicating the buffer now contains an additional character.

And RCV does a wait(CHARS) to ensure the buffer has at least one character before reading from the buffer.

Since CHARS was initialized to 0, we've enforced the constraint that the i_th call to signal(CHARS) precedes the completion of the i_th call to wait(CHARS).

In other words, RCV can't consume a character until it has been placed in the buffer by SEND.

Does this mean our producer and consumer are now properly synchronized?

Using the CHARS semaphore, we implemented *one* of the two precedence constraints we identified as being necessary for correct operation.

Next we'll see how to implement the other precedence constraint.

What keeps the producer from putting more than N characters into the N-character buffer?

Nothing.

Oops, the producer can start to overwrite characters placed in the buffer earlier even though they haven't yet been read by the consumer.

This is called buffer overflow and the sequence of characters transmitted from producer to consumer becomes hopelessly corrupted.

What we've guaranteed so far is that the consumer can read a character only after the producer has placed it in the buffer, i.e., the consumer can't read from an empty buffer.

What we still need to guarantee is that the producer can't get too far ahead of the consumer.

Since the buffer holds at most N characters, the producer can't send the (i+N)th character until the consumer has read the i_th character.

Here we've added a second semaphore, SPACES, to manage the number of spaces in the buffer.

Initially the buffer is empty, so it has N spaces.

The producer must WAIT for a space to be available.

When SPACES in non-zero, the WAIT succeeds, decrementing the number of available spaces by one and then the producer fills that space with the next character.

The consumer signals the availability of another space after it reads a character from the buffer.

There's a nice symmetry here.

The producer consumes spaces and produces characters.

The consumer consumes characters and produces spaces.

Semaphores are used to track the availability of both resources (i.e., characters and spaces), synchronizing the execution of the producer and consumer.

This works great when there is a single producer process and a single consumer process.

Next we'll think about what will happen if we have multiple producers and multiple consumers.