In this exercise, we will learn how semaphores can be used to ensure that different precedence constraints in our programs can be satisfied.

Before diving into the use of semaphores to enforce our precedence requirements, let's review what tools we have available to us.

You can think of a semaphore as a shared resource that is limited in quantity.

If we have a semaphore S that is initialized to 0 then it represents the fact that currently resource S is not available.

If S equals 1, then that means that exactly one S resource is available for use.

If S equals 2, then there are 2 S resources available, and so on.

In order to make use of the shared resource, a process must first grab that resource.

This is achieved by adding a wait(S) call before the code that requires the resource.

As long as the value of S equals 0, the code that is waiting for this resource is stalled meaning that it can't get past the wait(S) command.

To get past the wait(S) call, the value of semaphore S must be greater than 0, indicating that the resource is available.

Grabbing the resource is achieved by decrementing the value of the semaphore by 1.

Analogous to the wait(S) command, we have a signal(S) command.

A signal of semaphore S indicates that one additional S resource has become available.

The signal(S) command increments the value of S by 1.

The result of this is that a process that is waiting on S will now be able to grab it and proceed with the next line of code.

Now, lets consider two processes, P1 and P2, that run concurrently.

P1 has two sections of code where section A is followed by section B.

Similarly, P2 has two sections which are C followed by D.

Within each process execution proceeds sequentially, so we are guaranteed that A will always precede B, and C will always precede D. Let's also assume that there is no looping and that each process runs exactly once.

We want to consider how we can make use of different semaphores to ensure any necessary precedence constraints in the code.

Suppose that the constraint that we need to satisfy is that the section B code completes before the section C code begins execution.

We can achieve this using semaphore S by first initializing the semaphore to 0 in shared memory.

Next, in order to ensure that section C code does not begin running too early, we add a wait(S) call before the section C code.

As long as S = 0, the code in process P2 will not get to run.

P1 on the other hand, will not be constrained in this way, so section A code can begin running right away.

Since section B follows section A, it will be executed after section A.

Once B completes, process P1 needs to signal our semaphore to indicate that it is now okay for process P2 to begin its execution.

The signal(S) call will set S = 1, which will allow P2 to finally move beyond the wait(S) command.

Next, lets consider a slightly more complicated constraint where section D precedes section A, OR section B precedes section C. In other words, P1 and P2 cannot overlap.

One has to run followed by the other but either of them can be the one to run first.

To achieve this we want to use our S semaphore as a mutual exclusion semaphore.

A mutual exclusion semaphore, or mutex, ensures that only one complete block of code can run at a time without getting interrupted.

This can be achieved by initializing our semaphore S to 1 and having a wait(S) statement at the top of both processes.

Since S is initialized to 1, only one of the two processes will be able to grab the S semaphore.

Whichever process happens to grab it first is the one that will run first.

There is one last piece of code we need to add to complete our requirements which is that at the end of both processes' code, we must signal(S).

If we do not signal(S) then only the process that happened to grab the S semaphore first will get to run while the other is stuck waiting for the S semaphore.

If at the end of the process, we signal S, then S gets incremented back to 1 thus allowing the next process to execute.

Note that because this code does not loop, there is no concern about the first process grabbing the S semaphore again.

Finally, lets consider one last set of constraints.

In this case, we want to ensure that the first section of both processes P1 and P2 run before the second section of processes P1 and P2.

In other words A must precede B and D, and C must precede B and D.

The constraint that A must precede B, and C must precede D is satisfied by default because the code is always executed in order.

This means that our constraint reduces to A preceding D, and C preceding B.

To achieve this, we need to use two semaphores, say S and T and initialize them to 0.

After the first section of a process completes, it should signal to the other process that it may begin its second section of code provided that it has already completed its first section of code.

To ensure that it has already completed its first section of code, we place the signal calls between the two sections of code in each process.

In addition to signaling the other process that it may proceed, each of the processes needs to wait for the semaphore that the other process is signaling.

This combination of signal and wait ensures that sections A and C of the code will run before sections B and D. Since the semaphores are initialized to 0, the wait(S) will not complete until P1 calls signal(S) at which point it has already completed section A. Similarly, the wait(T) will not complete until P2 calls signal(T) at which point it has already completed section C.

So once the processes can get past their wait commands, we are guaranteed that both first sections of code have already run.

We have also not forced any additional constraints by requiring A to run before C or C to run before A, and so on.

Of course we could have swapped our use of the S and T semaphores and ended up with exactly the same behavior.

Note, however, that we cannot swap the signal and wait commands around.

If we tried to call wait before signal, then both processes would get deadlocked waiting for a semaphore that never gets signaled.

This highlights the fact that when using semaphores you must always be very careful to not only worry about satisfying the desired requirements, but also ensuring that there is no possibility of ending up in a deadlock situation where one or more processes can never run to completion.