****

**Department of Electrical Engineering and Computer Science**

**ECCE- Operating Systems**

**Process Synchronization**

By:

Rashed Aljaberi

ID: 100053179

**Overview:**

In this project, we will explore several ways of insuring that two processes don’t access a memory location at the same time

**Introduction:**

Imagine seeing a tweet that you like, and this tweet has 0 likes so you decide to give it a like. At the same moment you go to click the like button, someone else does the same thing. Logically, after you two give it a like, there are two new likesso the tweet should update and show the value 2; but for some reason it shows 1. Let’s visualize why this happens. After a tweet is liked, a chain reaction occurs to increment the number of likes in the database. This is how it looks like:

**You: Second person**

Liked tweet

Liked tweet

**Server**

Get current number of likes

Likes=0

Get current number of likes

Likes+1=1

Increment likes

Increment likes

Likes+1=1

**Server**

Likes=1

Update new value to 1

Update new value to 1

each ‘like’ creates a new process that increment the number of likes. The process has 3 steps:

1. Get current value
2. Compute the new value
3. Update the value in server to the new computed value

So, both processes got the current value and it was 0. Then they individually incremented their values and it became 1. Finally, they both set the new value of ‘likes’ to 1. This error is known as a race condition. It occurred because both processes accessed the memory at the same time. So reasonably, to prevent the error we need to make sure they don’t access the memory at the same time. In other words, when one process is accessing the memory and is still not done with it (this is known as being in the critical section) the other process should wait. So it should look something like this:

**You: Second person**

Liked tweet

Liked tweet

**Server**

Likes=0

Get current number of likes

Increment likes

Likes+1=1

Get current number of likes

**Server**

Likes=1

Update new value to 1

Likes+1=2

Increment likes

**Server**

Likes=2

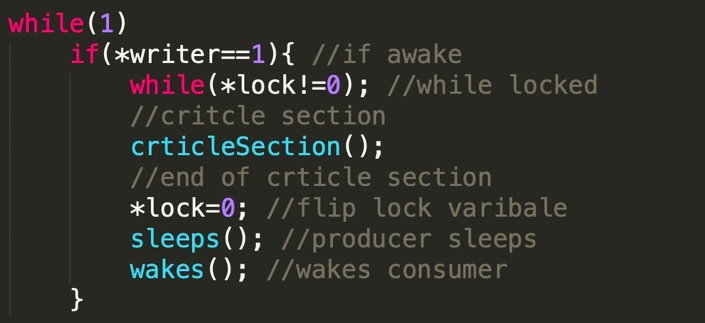
Update new value to 1

This way, we have achieved mutual exclusion and prevented a race condition.

As easy as it sounds, it is not that straightforward to ensure that 2 processes don’t enter a critical section at the same time. So in this report, I will discuss several solutions in which mutual exclusion is achieved.

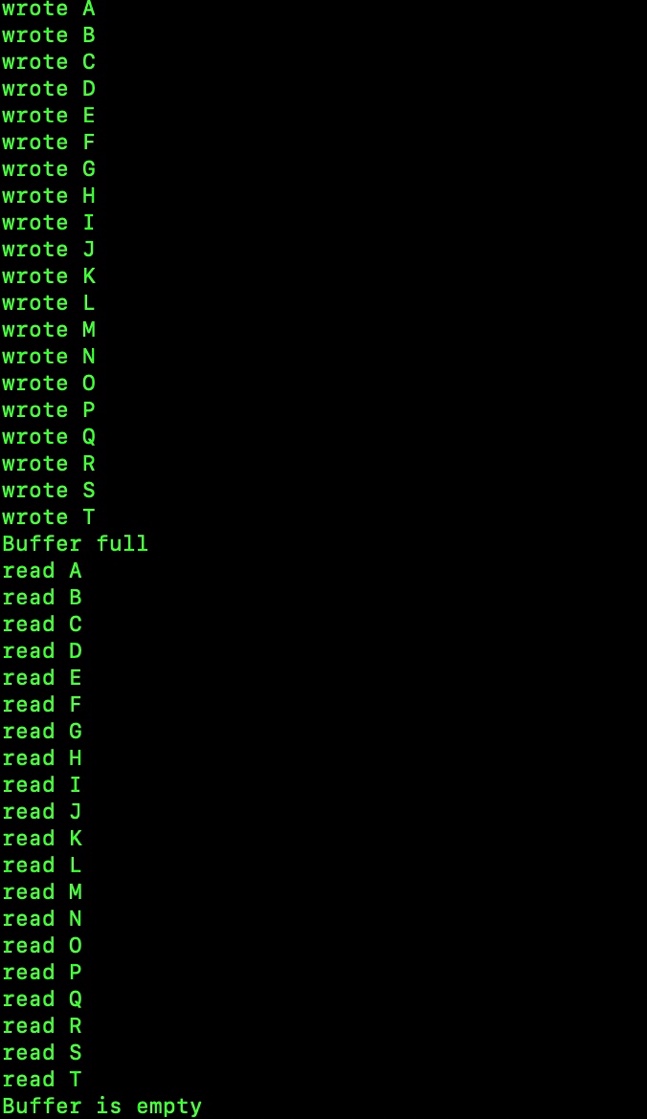
**Problem:**

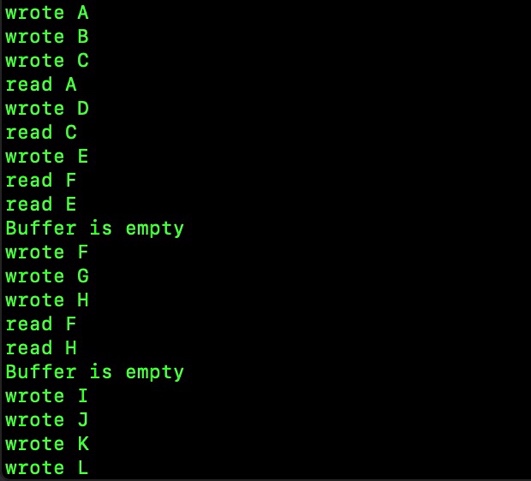
There are 2 processes: A and B. A is a producer writing into a fixed buffer, while B is a consumer reading the elements written by A. The 2 processes should achieve mutual exclusion by exchanging sleep/wake calls and using process synchronizing techniques. 5 techniques will be used and they are: Lock Variables, Strict Alternation, Peterson’s Solution, Binary Semaphores, and Counting Semaphores. For Counting Semaphores, an additional consumer process C will exist.

**Lock Variables**

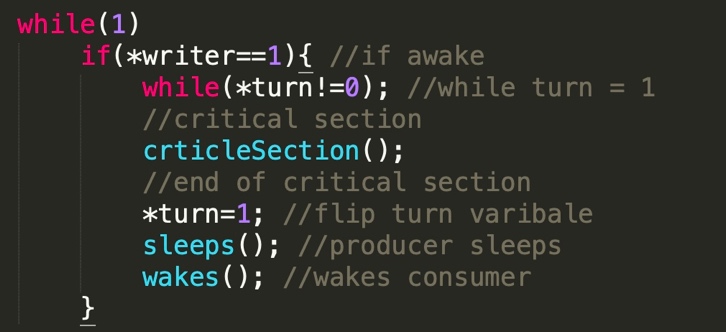
For Lock Variables, the 2 processes have a shared variable ‘lock’. This variable is initialized to 0. When the lock is 0, it means that the no one is in the critical section. Then a process would flip the lock variable to 1, indicating it is going into the critical section. When it is set at 1, the other process won’t be able to enter. When the process exits the critical section, it will reset the lock variable to 0 so that the other process could enter. A standard output looks like Figure 1 below.

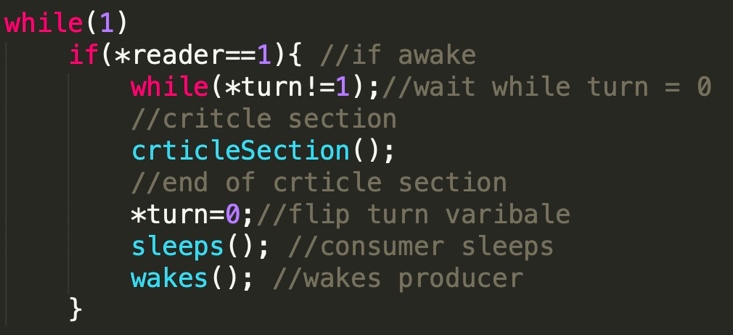
A disadvantage to lock variable is that there is a scenario where both processes check the lock variable at the same time, they both see that it is zero, so both enter the critical section at the same time. This is a race condition and this scenario is demonstrated in figure 2. But this only happens when we are not using sleep/wake calls, because when we use the calls, the producer will sleep after exiting the critical section and will not attempt to change the lock variable so the consumer is safe to flip the lock and enter the critical section. Another disadvantage is that a process could forget to flip the lock (or exit before flipping) and keep it locked forever.

**Figure 1 Figure 2**



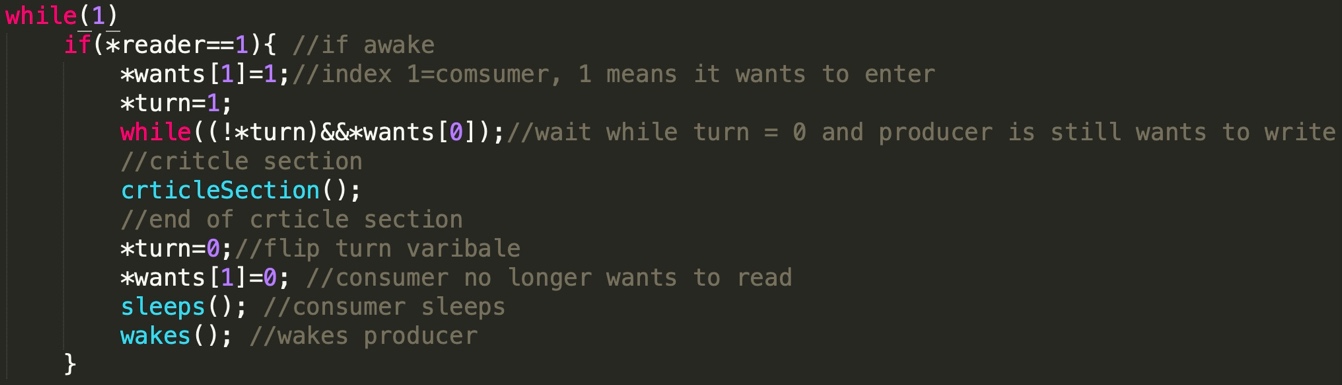
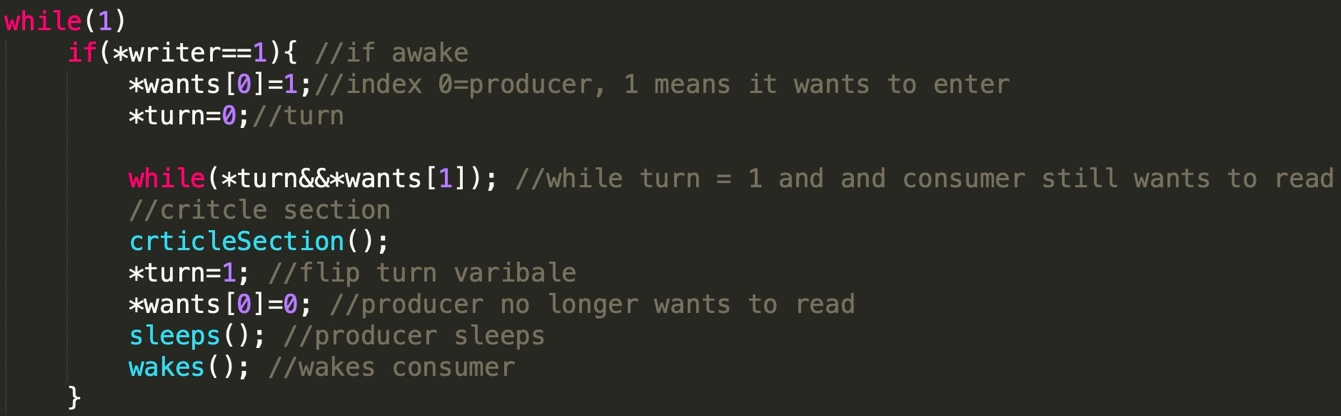
**Strict Alternation**

****

****

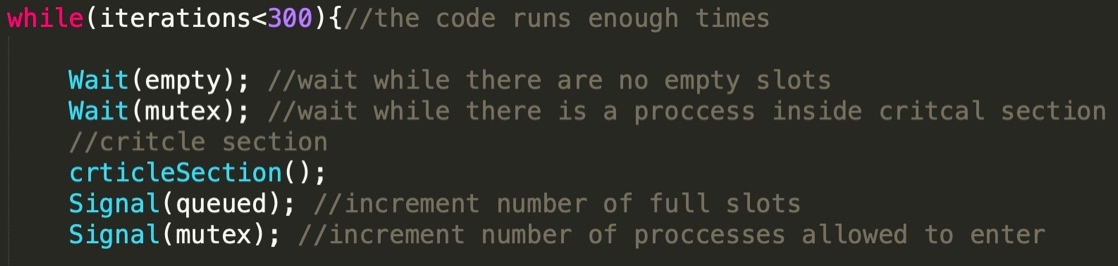
For Strict Alternation, the 2 processes have a shared variable ‘turn’. This variable is initially 0 to indicate that it is the producer’s turn to enter the critical section. When the producer finishes, it will give the turn to the consumer by flipping the variable to 1. The standard output is also demonstrated in Figure 1.

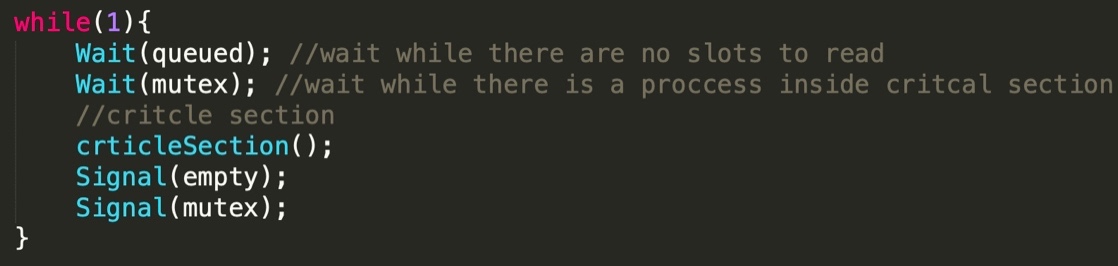
This technique does not have the same race condition issue of both processes changing the variable at the same time (like in Lock Variables), but it has an issue where a process is given a turn but that process does not want to enter critical section so the turn gets stuck forever.

******Peterson’s solution**

This technique builds on the Strict Alternation solution and adds a fix to the issue mentioned above. It does that by introducing an array of flags, each representing the need of a process to enter the critical section. If a process does not want to enter the critical section, its flag would show that and when it is its turn, it won’t get stuck and the other process will be able to enter the critical section. This also has an output demonstrated by figure 1

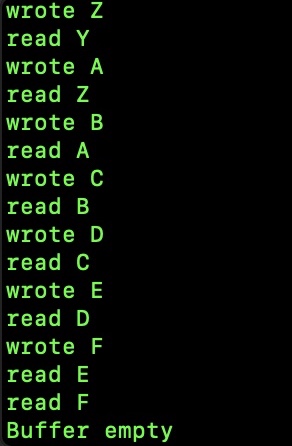
**Binary Semaphores**

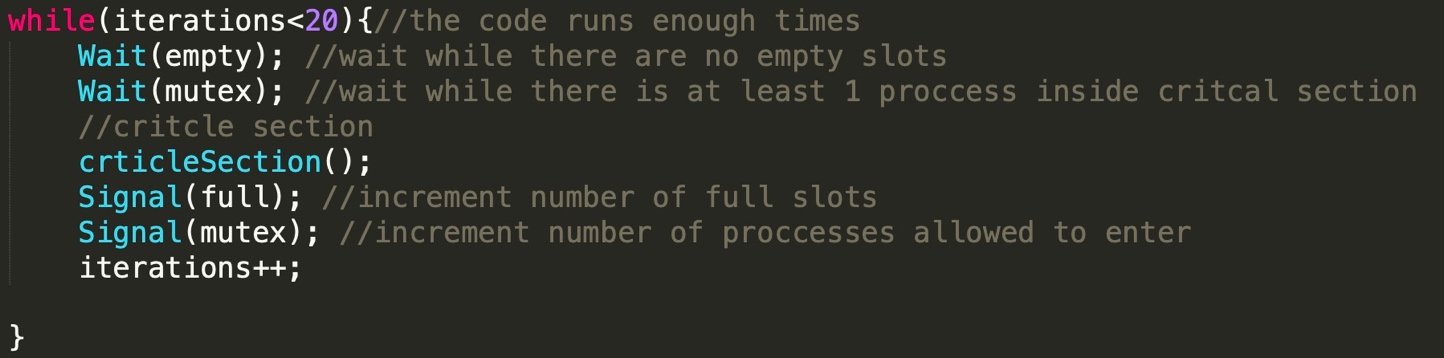
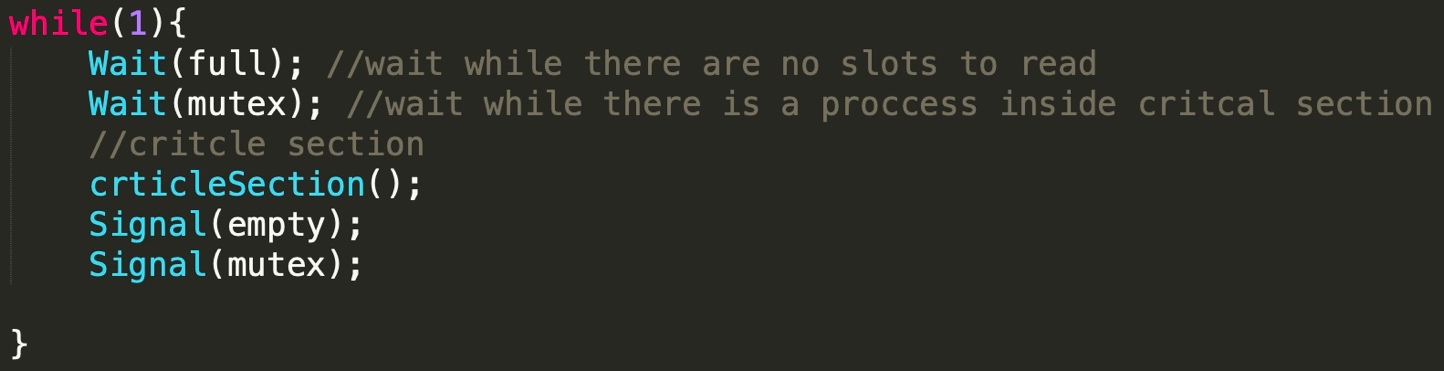


****

In this technique, we use a ‘mutex’ variable that can be seen just like a lock variable. When it is 1, a process can enter and when it is 0 a process can’t. You could also imagine that this value to represent the number of values allowed to enter, when the value is 1, 1 process is allowed to enter and when it is 0, no process is allowed to enter. Also, the algorithm checks (from the producer’s side) if the buffer is full before attempting to enter critical section (or if buffer is empty before entering critical section from consumer’s side). Due to the nature of this algorithm, the producer will write only 1 element then leave the critical section and the consumer will read only 1 element. There is no sequential turn and whoever tries to enter the critical section first gets to enter first. So a consumer might enter a critical section 2 times consecutively before the producer enters again. A standard output is shown in figure 3.

**Figure 3**

****

**Counting Semaphores**

As the name might imply, the Counting Semaphores is like the Binary Semaphores but it works on more than 2 processes. Instead of initializing mutex to 1 like in Binary Semaphores, we start with the max number of processes we want accessing the critical section at the same time. So in this case, it is 2. Since we want both consumers displaying what they read neatly, consumer 1 will print the entire buffer then consumer 2 will print the entire buffer. Figure 4 shows how it would look like

**Figure 4**

****

**Conclusion:**

In this report, several process synchronization solutions were discussed. In my opinion, the most important and practical one to remember is Semaphore since it can incorporate more than 2 processes. But, I will also advice using Peterson’s solution when there are only 2 processes because it is simpler than Semaphore, and it doesn’t have the possibility of getting stuck forever if the other process doesn’t adjust the turn/lock variable like in Lock Variables and Strict Alternation

**Assumptions:**

1. for Counting Semaphores, both processes must read the same exact elements. For this, we need to implement the method so that the producer fills the entire buffer at once then the consumers empty it all at once. Otherwise, processes will enter randomly and once a process read something, the other might not be able to because the buffer might overwrite it.
2. task description says, “**After both processes finish, sleep and wake up calls are exchanged”** so the assumptions is all solutions use sleep/wake on top of the busy waiting.
3. Binary Semaphore and Counting Semaphore don’t need sleep and wake since there is already wait and signal for a mutex and a full/empty variable. Adding a sleep/wake function will be an overkill