There are three examples of race condition problems: Producer/Consumer, Dining Philosophers, Readers/Writers

**Producers and Consumers**

*I realize this is not the example from class but another I studied to try and understand the concept.*

In this example,

a producer and a consumer both work on the same set of data, such as a buffer.

If the producer produces data faster than the consumer can retrieve the data from the buffer this causes the producer to not be able to add to the buffer because it is full.

If the consumer retrieves data faster than the producer can write data to the buffer, the consumer will run out of data to retrieve because the buffer is empty.

An example would be a buffer where the producer thread inserts a value to the buffer to represent the temperature in Fahrenheit and a consumer thread takes each value collected and converts it into Celsius. Both are working on the same set of data, in this case a buffer, which makes the buffer itself the critical region.

Mutex semaphores only….

With just mutex semaphores to resolve the race condition, we need 1 mutex semaphore to represent when the thread enters the critical region and 2 mutex semaphores to represent when the two undesirable states of empty and full occur in the buffer.

The producer checks to see if there are any empty slots (in other words, whether the buffer is full). The producer goes to sleep if there are no empty slots. If there *are* empty slots, the mutex locks the critical region so the producer can produce and insert an item into the buffer and then increments the count.

Then the mutex unlocks the critical region and the thread exits. It then checks to see if there is one full slot (N-1) in the buffer and if so, sends a signal to wake up the consumer.

Conversely, the consumer checks to see if there are any full slots (whether the buffer is empty). The consumer goes to sleep if there are no full slots. If there *are* full slots, the mutex locks the critical region so the consumer can remove an item from the buffer and then decrements the count.

The mutex unlocks the critical region and the thread exits. It then checks to see if there is at least one empty slot and if so, wakes up the producer.

Lost Wakeup signal

In the code where an if condition is used, the test condition is a two-step operation, non-atomic. First it loads the value to a register, then it tests the value that was loaded into the register against the condition. Because the test condition (checking to see if the buffer is full or empty) is not atomic it is possible, for example, for a consumer process to start first with a value of 0 (empty buffer) loaded to the register and before it can test the condition a context switch occurs causing the producer to enter the critical region, locking out the consumer, it produces an item, inserts it into the buffer and increments the count to 1, all while the consumer is blocked. When the producer exits the critical region, it sends a signal to wake up the consumer, but the signal is lost because when the consumer picks up where it left off when it was blocked, it still sees the 0 that was previously loaded to the register. The producer continues to produce items until the buffer is full and then goes to sleep awaiting the consumer’s signal that is never to come because the consumer is already asleep. This causes both the producer and consumer to wait infinitely.

This is the example we reviewed in class….

Using just synchronization semaphores….

Using synchronization semaphores of down() and up(),

for a producer thread, using a while loop, that states while TRUE,

produce an item,

down(empty), meaning decrement the number of empty slots,

insert the item produced,

up(full), meaning increment the number of full slots.

In the consumer thread, again using a while loop, that states while TRUE,

down(full), meaning decrement the number of full slots,

remove an item,

up(empty), meaning increment the number of empty slots.

Not having a mutex semaphore above may fix the empty/full issue but it still hasn’t corrected the race condition because the threads can still access the critical region at the same time.

When a mutex is added within the while loop this is used as a locking mechanism to allow each thread to enter the critical region and do their work without the interference of other threads accessing the critical region at the same time.

The new process for these threads would be….

Using synchronization semaphores of down() and up() and mutex semaphores of down(mutex) to lock and up(mutex) to unlock,

for a producer thread, using a while loop, that states while TRUE,

produce an item,

down(empty), meaning decrement the number of empty slots,

then a down(mutex) that locks other threads out of the critical region,

insert the item produced,

up(mutex) to unlock the critical region,

then up(full), meaning increment the number of full slots.

In the consumer thread, again using a while loop, that states while TRUE,

down(full), meaning decrement the number of full slots,

then a down(mutex) that locks other threads out of the critical region,

remove an item,

up(mutex) to unlock the critical region,

then up(empty), meaning increment the number of empty slots.

The synchronization and mutex semaphores used together resolve the race condition.

However, if the mutex and the synchronization semaphores are not in the correct order you put a mechanism that is not required within the critical region and is lost to the other thread.

If you enter the critical region and change the count of the full or empty slots within the critical region, if it happens that the producer realizes within the critical region that there are no empty slots it immediately goes to sleep IN the critical region and the consumer is never able to enter the critical region thus creating a deadlock between the threads.

Conversely, if the consumer follows this same process, it will also go to sleep IN the critical region and the producer is never able to enter the critical region causing deadlock as well.

Dining Philosophers