CSE312 Homework1 Report

Yeşim Yalçın-200104004094

1-) Problem Definition and My Solution Approach

The problem was to add create, exit, yield and join functions to the given OS and then implement multiple threads that communicate with eachother in a producer-consumer way.

I firstly thought about where I wanted to implement which functions. After watching the videos, I decided to implement the thread functions in the kernel.cpp. The reason of this decision was that I needed to reach the created TaskManager to manage the thread functions. However, I also created a header file called mysyscalls.h to be able to use these functions outside of kernel.cpp. I made some changes on multitasking.h and multitasking.cpp to be able to implement these functions.

Afterwards, I decided to implement the threads in the kernel.cpp as well however, later on I changed that decision. I created my producer-consumer threads in a class named ProducerConsumerThread and this class is defined in ProducerConsumer.h then implemented in ProducerConsumer.cpp. The reason of this decision was that I wanted to have their global variables properly organized. If I had implemented them in the kernel.cpp, then the whole kernel.cpp would have access to the thread globals. However, this would have been very troublesome for the threads to function properly. Moreover, giving the threads some abstraction ended up being a better design.

I also decided to create a few test threads named taskA, taskB, taskC and taskD. These are very basic threads that are implemented in kernel.cpp. The only reason I created them was to test thread_exit(), thread_join() also to create a more heavy scheduling. They are implemented in kernel.cpp instead of some separate class because they are very simple threads which do not have any global variables etc. It would not make any difference aside from abstraction to implement them anywhere else.

Eventually, I have edited:
-multitasking.h
-multitasking.cpp
-kernel.cpp
-mysyscalls.h
-ProducerConsumer.h
-ProducerConsumer.cpp
-interrupts.h
-interrupts.cpp

2-) Thread Functions

A-) int thread_create(Task* newTask)

To implement this function I utilized the AddTask function of TaskManager. This function takes Task object and adds it to the TaskManager. After adding a new task to the TaskManager this task is added to the RoundRobin schedule array, gains a thread ID, gains a ready state meaning it is runnable.
This function returns the created task’s threadID in case this id will be used for future operations.

B-) void thread_exit()
To implement this function I added states to Tasks.
The states are:
-1 if terminated
0 if blocked
1 if ready

After thread_exit() functions is called, the function taskManager::terminateRunningThread() function runs. This function sets the state of the thread as terminated and increments the terminated thread's counter.
The threads that have terminated state are ignored while in the scheduler. They are never chosen to run again. Moreover, they are removed from the tasks array of the TaskManager whenever they are encountered in the Scheduler.
I firstly tried to remove the terminated tasks from the tasks array in taskManager::terminateRunningThread() function. However, it did not work. I assume it was related to the cpustate therefore, I decided to remove them in the Scheduler. Because cpustate is properly set in this function, this was the right decision.

In taskManager::terminateRunningThread() function, I also unblock any threads that are waiting for this thread to be terminated.

In the end I call thread_yield() so that the terminated thread stops running and another thread is run instead.

C-) void thread_join(int threadID)
To implement this function I added threadID's to Tasks. The threadID's start from 0 and is incremented by 1 for each thread. It is different from their indexes in the task array. The indexes of the threads might change when a terminated thread is removed from the array. Therefore I added this threadID concept.

After thread_join() function is called TaskManager::joinThread(int threadID) runs. This function firstly checks if the given threadID is a valid ID.
The threadID should be:
-Bigger or equal to 0
-Smaller than totalTaskCount
-Not equal to current thread's id
Also if the threadID belongs to a thread that is unblocked because of the current thread, it is not supported either.
If the threadID is not supported, the function returns with a false meaning the join failed.
If the threadID is supported then the function continues. It sets the state as blocked and saves the threadID that is being waited to terminate as waitingThread.

In the end I call thread_yield() so that the blocked thread stops runing and another thread is run instead.

D-) void thread_yield()
To implement this function I firstly had an idea to initiate a software interrupt in thread_yield() function. By doing that, I would be able to achive changing the current running thread with some other thread and putting it to the back of the RoundRobin queue. The thread

that called thread_yield() would immediately stop running giving it's place to another thread and would not run until all the other threads got one chance to run.

To be able to do that, I made to_bottom function inside interruptstubs.s global and called this function inside my thread_yield() function. By doing this, I would force an interrupt to happen. At first, it worked as I intended but after testing my code more, I realized it caused some unhandled interrupt errors.
I tried to use the static InterruptHandler functions inside the InterruptManager class thinking the interruptnumber in to_bottom function was not set properly when I called it from outside. However, this solution caused the same errors.

Because I could not manage to force an interrupt myself and the software interrupts are not implemented in InterruptManager I decided to use another solution.

I added an interruptOccured data member to InterruptManager class and set this as 1 whenever an interrupt occured. In thread_yield() function, I set it as 0 and did a busy waiting until it became 1 again. It meant until an interrupt happened, the thread would have to stay in a loop. So if thread_yield() was called after thread_exit(), it would prevent a terminated function to run beyond it's limits. If the thread_yield() was called after thread_join(), it would prevent a blocked function to run without giving it's place as well. Therefore, eventhough it caused some waiting, it helped thread_yield() do it's job. Because timer interrupts happen very frequently, the waiting is very insignificant.

E-) Possible Deadlocks and Exceptions I Have Handled Related to Thread Functions:

- I do not let any terminated threads to stay in the tasks array for long. If I did, it would led zombie like threads to exist and take unnecessary space.
- I do not let any two threads block eachother by using join with eachother's IDs. If I did, both of them would be blocked and never be unblocked.
- I do not let any thread to block itself by using join with their ID's, a terminated thread's ID or a non existing thread's ID. If I did, it would stay blocked forever.
- Instead of using indexes for thread_join(), I use thread ID's. This way is a lot safer because indexes change, ID's do not. This prevents different behaviour of the same code.
- After thread_exit() and thread_join() I immediately call thread_yield() so that the terminated or blocked threads immediately give their turns to another thread. If I did not do that, the blocked threads might have run for more time than they should have.
- I return threadID's after creating them to achieve an implementation closer to real thread_create() implementations. This gives the opportunity to have easier communication between specific threads.

Notes:

- When a thread terminates, it needs to call thread_exit() function. If it does not call this function while terminating for any reason, then the system does not function properly. Therefore, I have added thread_exit() function anywhere a thread was going to terminate.

F-) Tests

To test all 4 thread functions I created taskA, taskB, taskC, taskD

- taskA runs in an infinite loop printing “A”. After printing once, it uses thread_join() on taskB. Therefore it should wait until taskB terminates.
- taskB runs in an infinite loop printing “B”. However, after it’s 100000’th iteration, it terminates. It also tries to use thread_join() on taskA however because taskA already used thread_join() on taskB, taskB’s should be ignored.
- taskC runs in an infinite loop printing “CCCCCCCC”. After printing 100 times, it uses thread_join() on taskD. Therefore it should wait until taskD terminates.
- taskD runs in an infinite loop printing “DDDDDDDDD”. However, after it’s 100000’th iteration, it terminates.

I ran a test using these threads and according to the test:
-A was printed once then stopped
-CCCCCCCC was printed once then stopped.
-B and DDDDDDDDD was printed 100000 times.
-After B and D terminated, A and C woke up so A and CCCCCCCC was printed infinitely.
The test results were the same as expected.
I cannot provide footage because without providing a video, it is very hard to show it with screenshots.
However, you can simulate the same test if you run the code. (The code will also have other producer-consumer threads running in the version I will send however, these test functions will be included as well. It should still be observable.)

3-) Producer-Consumer Threads

I firstly created a class named ProducerConsumerThread. This class have only static functions and data members. I firstly did not do it like that. However, while creating tasks, I needed to pass function pointers as a parameter. Because member function pointers aren’t the same as regular function pointers I had to make everything static.
Before making everything static, I was planning to reuse the class to create different instances of it and different pairs of producer-consumer while using the same code. However, because I made everything static this was not possible anymore. Therefore, instead of making only 1 producer, consumer functions in the class, I made two so that I could create different pairs using the same class.
To be able to do that I created producerThreadA, consumerThreadA, producerThreadB, consumerThreadB functions and using them, I created threads.
A and B only indicate the pair difference. Aside from that, the code they use are the same. They also have different static private data members for the same purposes like globals.

A-) Thread Scheduling
The producer threads are responsible of incrementing the itemCounter. ItemCounter stores the amount of produced items. Before incrementing the item counter, they also set the element with the current item counter as 1 inside the itemsList array meaning there is a product in that index. The capacity of produced items is set to 100. Because the itemList has size 100, the production of more than 100 items must be prevented.

The consumer threads are responsible of decrementing the itemCounter. After decrementing, they also set the element with the current item counter as 0 inside the itemsList array meaning there is not a product in that index. It should be noted that, the consumer should not try to consume non existing products. Therefore, the itemCount must never be smaller than 0.

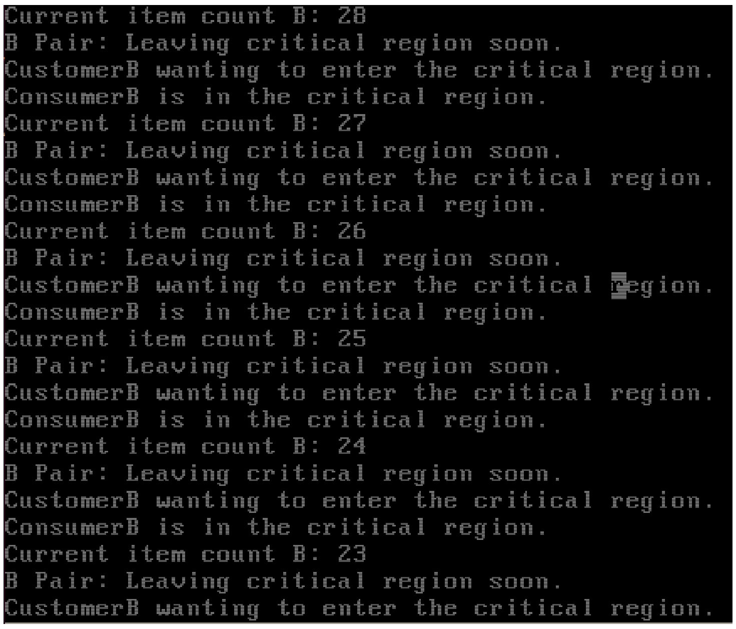
Moreover, while incrementing the itemCount a consumer must not access this itemCount at the same time vice versa. Therefore, I declared this operation in a critical region. Before entering the critical region I called enterRegion() function and while leaving I called leaveRegion() function.

In enterRegion() function, I firstly find out who called the function. If the caller is a producer, I set their index as 0. If the caller is a consumer, I set their index as 1. These indexes are used in interested arrays. Interested arrays are used to find out if a producer or consumer is interested in getting into the critical region or in the region. If the value in the array is 1 it means they are interested. If 0, they are not interested. However, before setting all these values, I check if the current itemCounter is valid to enter the critical region. If not I use busy waiting till it becomes valid. After that, I set the interested array values and also give the turn to the caller. If the turn is in the caller and the other consumer/producer is interested in getting into the critical region (meaning they are actually inside), I do busy waiting till they are not interested anymore. Afterwards, I let the caller enter the critical region.

In leaveRegion() function, I again find out who the caller is and set their interested value as 0 meaning they are no longer in the critical region.

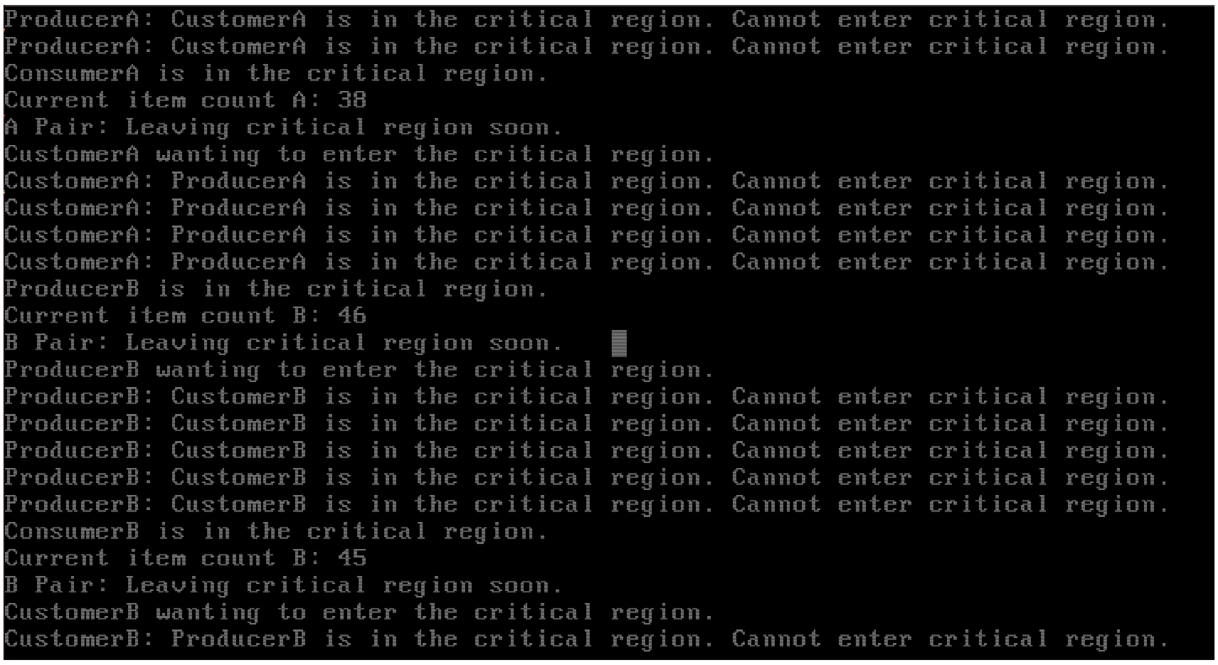
In my test I created 2 pairs using this producer-consumer scheduling. One pair being pairA, the other being pairB. As I mentioned, they are using different functions and different data members as pairs. I prohibited creating multiple pairs using the same functions by having producerSet, consumerSet data members. These are set to 1 if there is already an existing thread of the function and another one being run is not allowed. These are set to 0 if there is not an existing thread of the function. This was needed because having more than one producers or consumers for the same pair would make the functions run wrongly and communication between only two threads is wanted in the homework.

B-) Testing



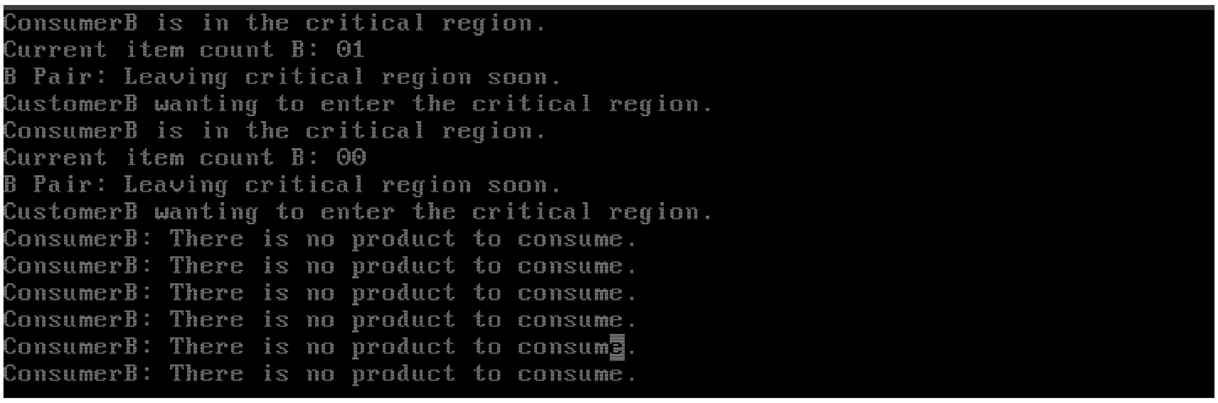
```
Current item count B: 28
B Pair: Leaving critical region soon.
CustomerB wanting to enter the critical region.
ConsumerB is in the critical region.
Current item count B: 27
B Pair: Leaving critical region soon.
CustomerB wanting to enter the critical region.
ConsumerB is in the critical region.
Current item count B: 26
B Pair: Leaving critical region soon.
CustomerB wanting to enter the critical region.
ConsumerB is in the critical region.
Current item count B: 25
B Pair: Leaving critical region soon.
CustomerB wanting to enter the critical region.
ConsumerB is in the critical region.
Current item count B: 24
B Pair: Leaving critical region soon.
CustomerB wanting to enter the critical region.
ConsumerB is in the critical region.
Current item count B: 23
B Pair: Leaving critical region soon.
CustomerB wanting to enter the critical region.
```

The behaviour of a Customer running and consuming items were normal. The order of getting into critical regions, consuming, getting out of regions were the same as expected.



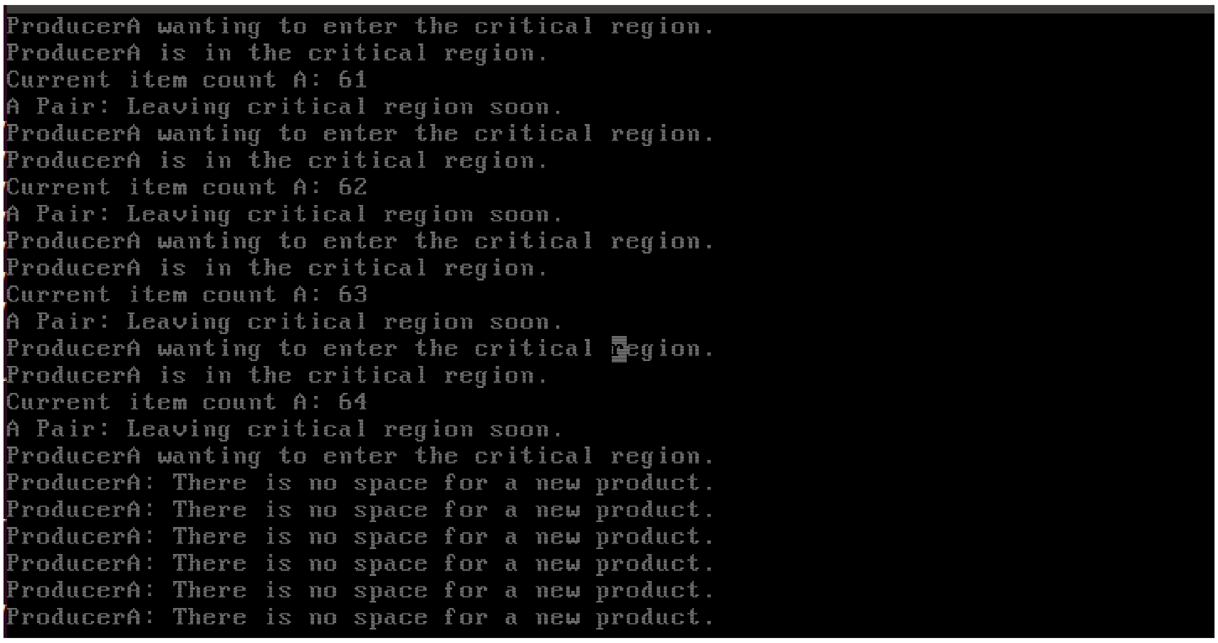
```
ProducerA: CustomerA is in the critical region. Cannot enter critical region.
ProducerA: CustomerA is in the critical region. Cannot enter critical region.
ConsumerA is in the critical region.
Current item count A: 38
A Pair: Leaving critical region soon.
CustomerA wanting to enter the critical region.
CustomerA: ProducerA is in the critical region. Cannot enter critical region.
CustomerA: ProducerA is in the critical region. Cannot enter critical region.
CustomerA: ProducerA is in the critical region. Cannot enter critical region.
CustomerA: ProducerA is in the critical region. Cannot enter critical region.
ProducerB is in the critical region.
Current item count B: 46
B Pair: Leaving critical region soon.
ProducerB wanting to enter the critical region.
ProducerB: CustomerB is in the critical region. Cannot enter critical region.
ProducerB: CustomerB is in the critical region. Cannot enter critical region.
ProducerB: CustomerB is in the critical region. Cannot enter critical region.
ProducerB: CustomerB is in the critical region. Cannot enter critical region.
ProducerB: CustomerB is in the critical region. Cannot enter critical region.
ConsumerB is in the critical region.
Current item count B: 45
B Pair: Leaving critical region soon.
CustomerB wanting to enter the critical region.
CustomerB: ProducerB is in the critical region. Cannot enter critical region.
```

When there were switches between consumers and producers, the Peterson's Algorithm worked as intended. It only let one thread into the critical region. You can see here:
-Somewhere above CustomerB wanted to enter to the critical region therefore it's interested value was set to 1. Here you can understand it from the lines:
ProducerB: CustomerB is in the critical region. Cannot enter the critical region.
Afterwards, it repeats for a while because of the busy waiting.
Then the thread changes to CustomerB and we see the line:
ConsumerB is in the critical region.
We see that our assumption above about ConsumerB was right. It's interested value was set to 1 and it finally entered the region. Afterwards, it leaves the region and wants to enter the region again. However, above ProducerB's interested value was set to 1 already and it was busy waiting. Therefore, ConsumerB cannot enter the critical region. When the scheduler switches to ProducerB, the turnB value will not belong to them anymore therefore it's busy waiting will stop and it will enter the region first.
This test shows that there is no deadlock situations between thread switches.



```
ConsumerB is in the critical region.
Current item count B: 01
B Pair: Leaving critical region soon.
CustomerB wanting to enter the critical region.
ConsumerB is in the critical region.
Current item count B: 00
B Pair: Leaving critical region soon.
CustomerB wanting to enter the critical region.
ConsumerB: There is no product to consume.
ConsumerB: There is no product to consume.
ConsumerB: There is no product to consume.
ConsumerB: There is no product to consume.
ConsumerB: There is no product to consume.
ConsumerB: There is no product to consume.
```

Here we see that when there is no products to consume, the thread does busy waiting as intended.



```
ProducerA wanting to enter the critical region.
ProducerA is in the critical region.
Current item count A: 61
A Pair: Leaving critical region soon.
ProducerA wanting to enter the critical region.
ProducerA is in the critical region.
Current item count A: 62
A Pair: Leaving critical region soon.
ProducerA wanting to enter the critical region.
ProducerA is in the critical region.
Current item count A: 63
A Pair: Leaving critical region soon.
ProducerA wanting to enter the critical region.
ProducerA is in the critical region.
Current item count A: 64
A Pair: Leaving critical region soon.
ProducerA wanting to enter the critical region.
ProducerA: There is no space for a new product.
ProducerA: There is no space for a new product.
ProducerA: There is no space for a new product.
ProducerA: There is no space for a new product.
ProducerA: There is no space for a new product.
ProducerA: There is no space for a new product.
```

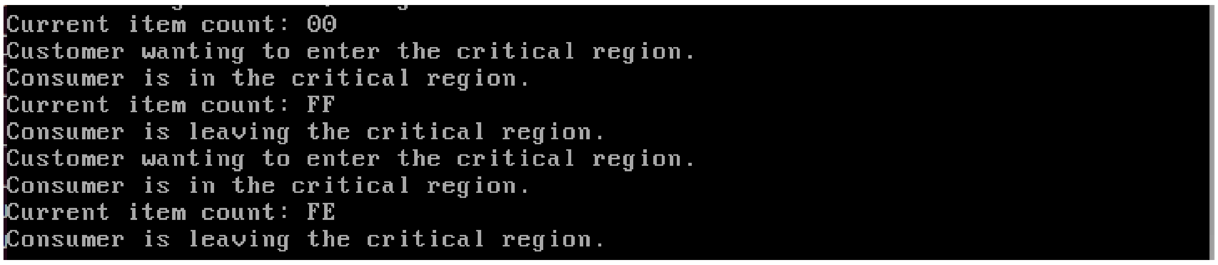
Here we see that when there is no space for a new product, busy waiting is done as intended.

C-) Race Conditions

Race conditions occur when two different threads try to access to the same resource at the same time. If this resource should not be accessed at the same time, the race condition occurs. In my scenario the race condition might happen if both the consumer and the producer tries to enter the critical region at the same time. This might lead to consumption of non existing items or production of more than max amount of items.

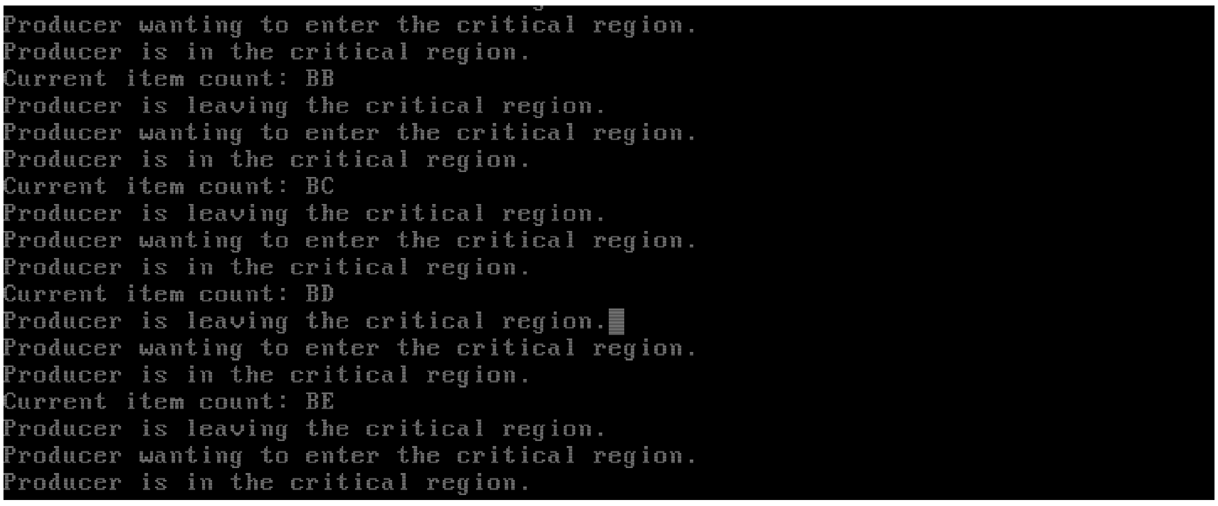
To show race conditions I created another implementation of producerThreadA and consumerThreadA functions. In the version I sent, I commented them out because there cannot be two implementation of the same functions. To test them you can comment out the actual implementations of them which are implemented above and when you want to test the actual versions, you can comment out these secondary implementations which are implemented below. It is mentioned which ones are the real, which ones are the secondary ones in the code as comments.

Because race conditions happen if two threads are in the critical region at the same time, I removed the control mechanism of entering the critical regions which are enterRegion and leaveRegion functions. When these functions are removed, we cannot have any scheduling. Moreover, the amount of the items are not controlled either.



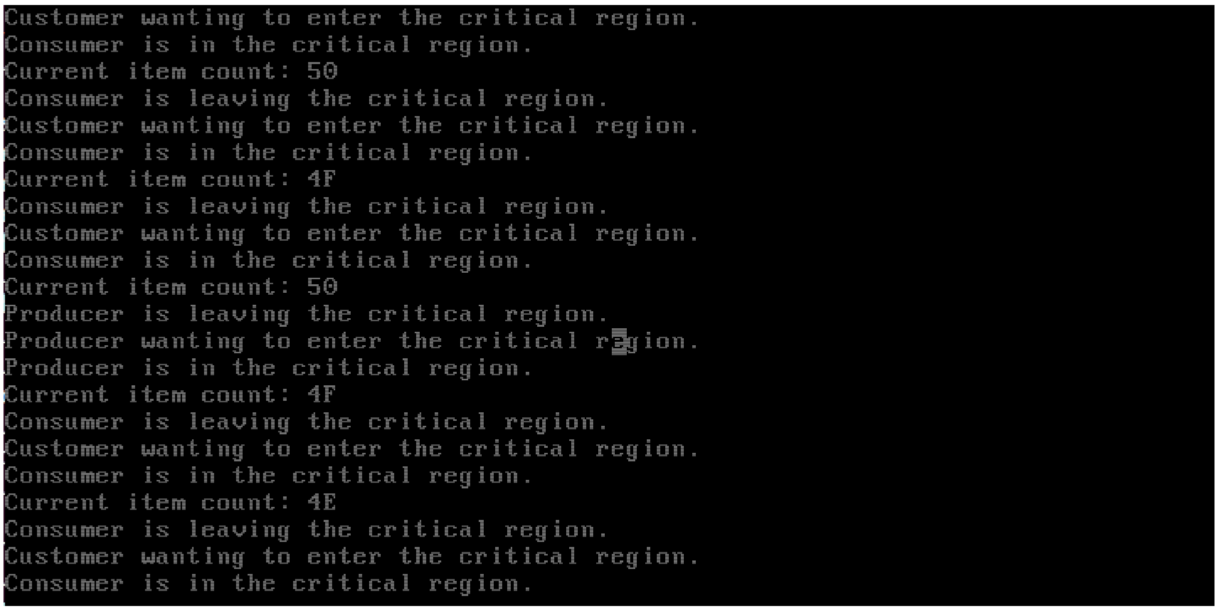
```
Current item count: 00
Customer wanting to enter the critical region.
Consumer is in the critical region.
Current item count: FF
Consumer is leaving the critical region.
Customer wanting to enter the critical region.
Consumer is in the critical region.
Current item count: FE
Consumer is leaving the critical region.
```

Here we see that the item count was 0 but the consumer still entered the critical region and consumed non existing items. The item count became something like FF after 0 which is not meaningful.



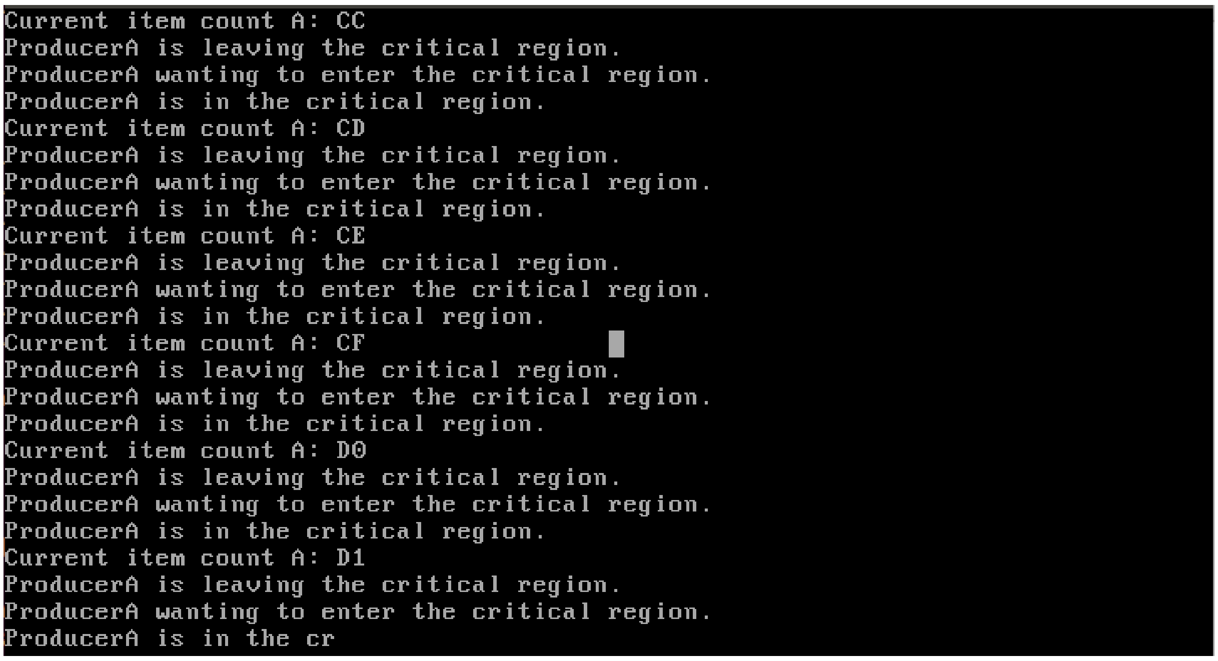
```
Producer wanting to enter the critical region.
Producer is in the critical region.
Current item count: BB
Producer is leaving the critical region.
Producer wanting to enter the critical region.
Producer is in the critical region.
Current item count: BC
Producer is leaving the critical region.
Producer wanting to enter the critical region.
Producer is in the critical region.
Current item count: BD
Producer is leaving the critical region.
Producer wanting to enter the critical region.
Producer is in the critical region.
Current item count: BE
Producer is leaving the critical region.
Producer wanting to enter the critical region.
Producer is in the critical region.
```

Here we see that producer keeps producing items however, the max amount should be 100. BB in decimal is 187.



```
Customer wanting to enter the critical region.
Consumer is in the critical region.
Current item count: 50
Consumer is leaving the critical region.
Customer wanting to enter the critical region.
Consumer is in the critical region.
Current item count: 4F
Consumer is leaving the critical region.
Customer wanting to enter the critical region.
Consumer is in the critical region.
Current item count: 50
Producer is leaving the critical region.
Producer wanting to enter the critical region.
Producer is in the critical region.
Current item count: 4F
Consumer is leaving the critical region.
Customer wanting to enter the critical region.
Consumer is in the critical region.
Current item count: 4E
Consumer is leaving the critical region.
Customer wanting to enter the critical region.
Consumer is in the critical region.
```

Here we see that:
-Customer is in the critical region.
-Current item count: 50
-Producer is leaving the critical region.
Meaning while customer was in the critical region and consuming items, the producer was also in the critical region. If it wasn't the case, there should have been a "Producer wanting to enter the critical region" text before "Producer is leaving the critical region".



```
Current item count A: CC
ProducerA is leaving the critical region.
ProducerA wanting to enter the critical region.
ProducerA is in the critical region.
Current item count A: CD
ProducerA is leaving the critical region.
ProducerA wanting to enter the critical region.
ProducerA is in the critical region.
Current item count A: CE
ProducerA is leaving the critical region.
ProducerA wanting to enter the critical region.
ProducerA is in the critical region.
Current item count A: CF
ProducerA is leaving the critical region.
ProducerA wanting to enter the critical region.
ProducerA is in the critical region.
Current item count A: D0
ProducerA is leaving the critical region.
ProducerA wanting to enter the critical region.
ProducerA is in the critical region.
Current item count A: D1
ProducerA is leaving the critical region.
ProducerA wanting to enter the critical region.
ProducerA is in the cr
```

Also here you can see that the Virtual Box froze probably because it tried to reach non allocated memory of the itemsListA array. Meaning handling the itemsCount is crucial. If two threads are in the critical region at the same time, there is no way to control the itemsCount.