CSE312 – HOMEWORK 1

For this homework we need to implement a thread library for our operating system.

In order to do this I used a layout design from Viktor Engellmans’s 15th video of his ‘Write your own operating system’ series on youtube.

What I already had were classes: Task and TaskManager.

I used both of them and made: Thread and ThreadManager.

First aim was to make this multithreading instead of multitasking/multiprocessing. In order to achieve that I placed a threadManager as an inner class of a Task. This allowed me to perform scheduling according to the inner threads of each task/process.

Some initial modifications were:

* Constructor of class Task was changed so that when instantiated it would create a thread called mainThread which would than be added to the queue of threads for that process in ThreadManager.
* In order to achieve multithreading, I also had to modify TaskManager’s Schedule() method which is called on each timer interrupt. Here, whenever there is more than one thread inside a process I make sure to update current thread’s id number to represent the index of the next element in my ‘queue’ array. This way next time this process performs execution the same thread won’t execute this time but only the one that was next in the queue.

After this I need to make sure that my code can perform thread creation, termination, joining and yielding.

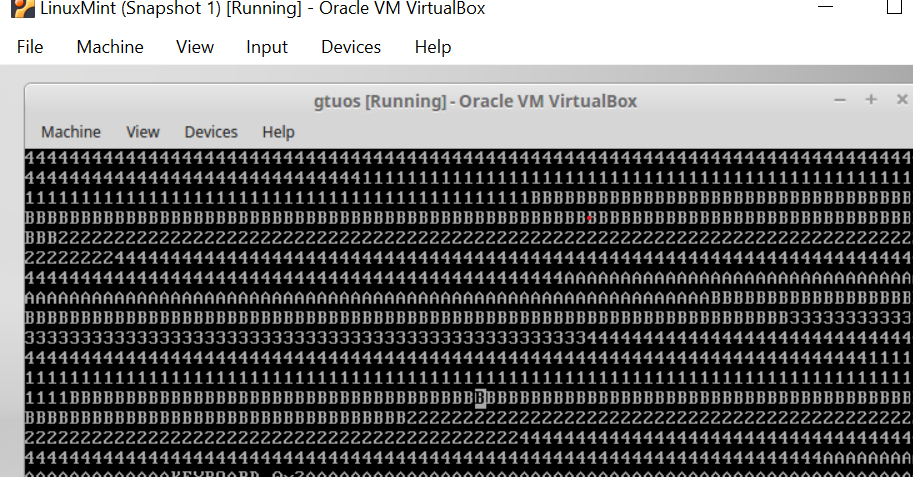
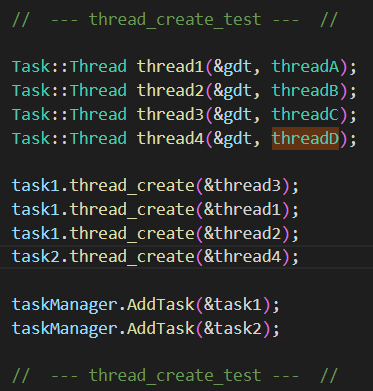
Thread manager and Thread table entry class

In order to do some additional private fields introduced. Mainly to make this possible instead of have a queue of threads in thread manager instance, I decided to make a new class called ThreadTableEntry. This was needed because it allows storing state of each thread. For storing state of threads new type was define called ThreadState and state of each thread could be either: RUNNING, YIELDED, TERMINATED or WAITING at any given point in time. This made scheduling more intuitive and easy. Also ThreadTableEntry contains an array of waiting threads ( threads that have joined a specific thread ) which is necessary so that upon termination of the thread, it would signal/inform threads that have been waiting on it by altering their number of joined threads by one. On the other hand numJoining field is needed in order to track the number of threads certain thread is waiting on. When this number reaches 0, a thread can finally continue running.

Thread creation:

This part was fairly simple and the same idea is used as it was for the tasks. It just adds another thread to the queue of threads in the process this thread is initialized. Also for this to work previously mentioned modifications in Task’s schedule() method are required.

Tests:

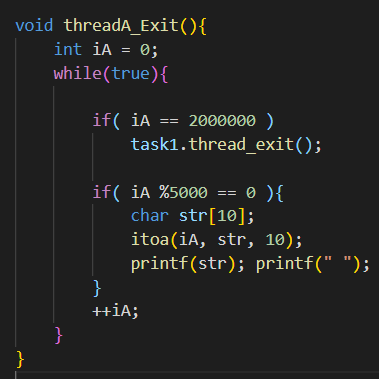


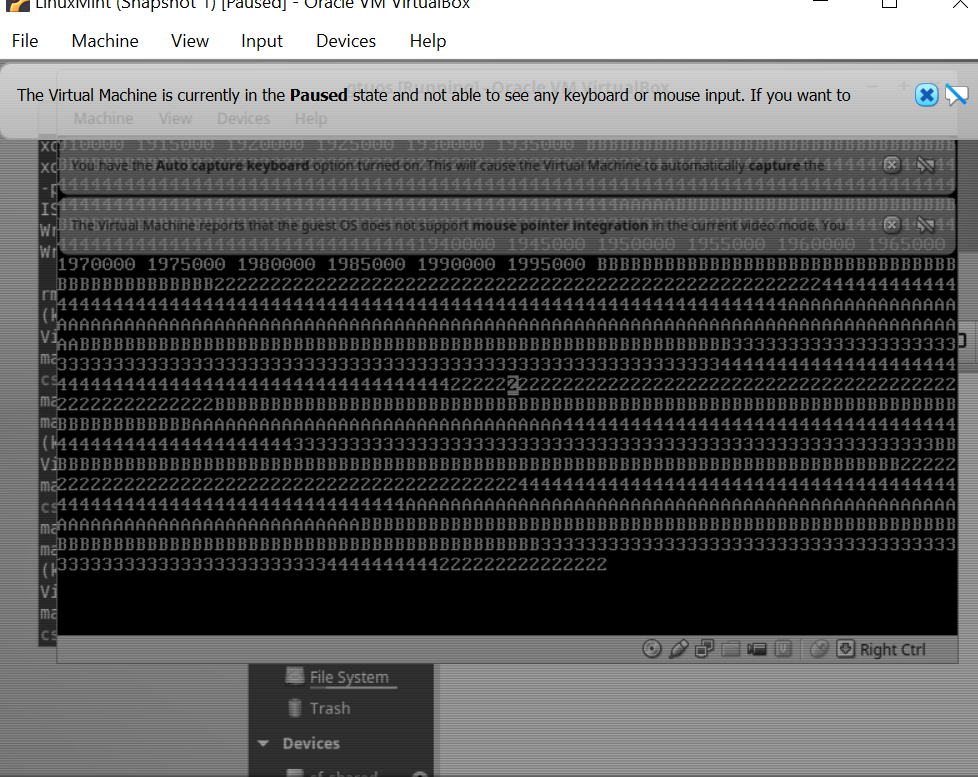
*We can see all the initiated and added thrads are running at the same time*

Thread termination:

In order to perform thread termination thread's state needs to be changed. At the same time, thread to be terminated should inform all other thread's that have 'joined' it and have now state of WAITING that it finished its execution and that they can now continue. This method will remove the given thread from thread manager's queue making sure that when a time interrupt occurs cpustate will be given to thread that is still running.

Tests:



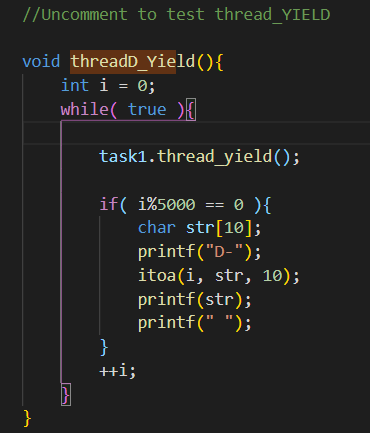


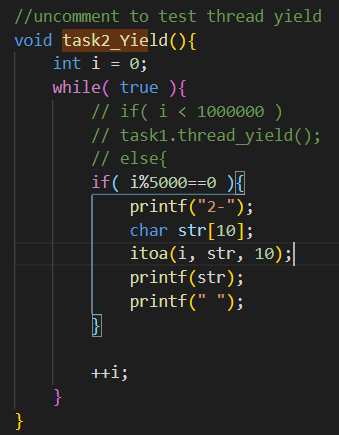
*Only after thread one ( the one outputing numbers until 2 000 000 ) reaches iteration of 2 000 000 does it call exit and after that we can’t see its output anymore*

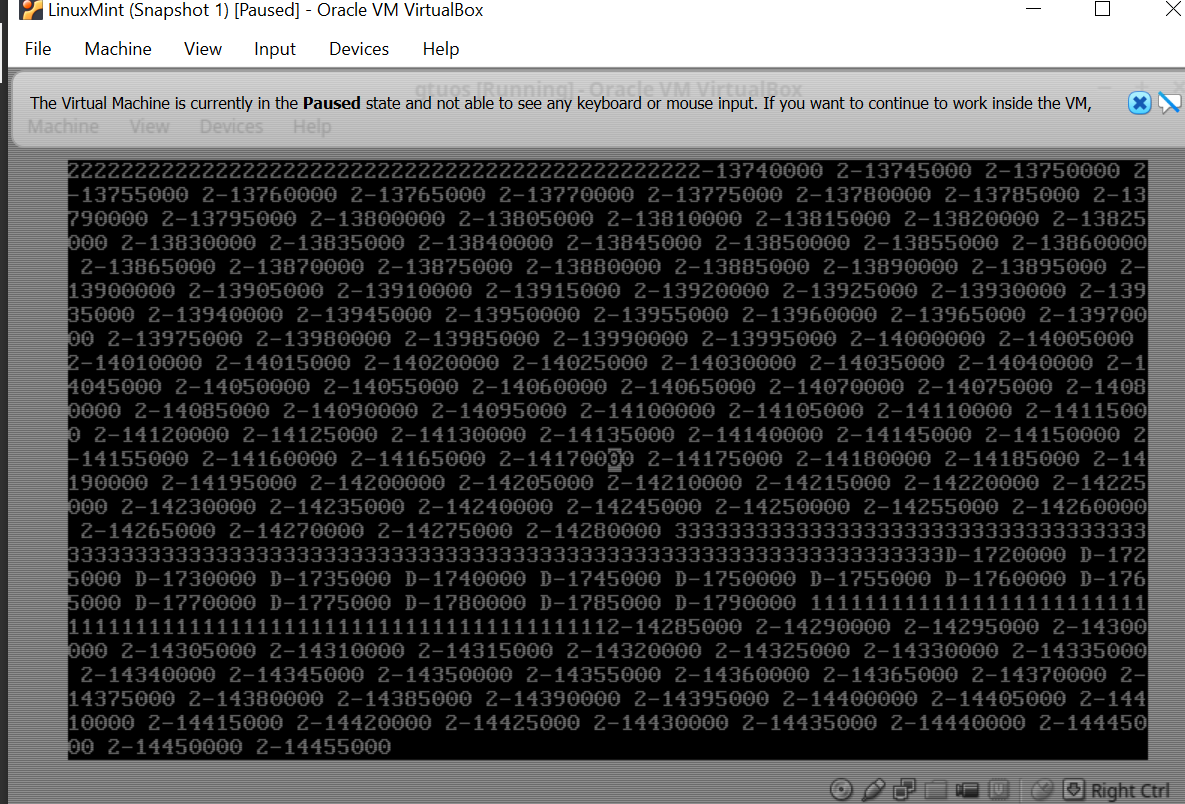
Thread yielding:

When yield occurs given thread immediately stops its execution and skips execution the next time its turn comes Important to note is that my system doesn't allowe yield to be called if another running thread doesn't exist. This is to avoid deadlock and system being overly 'kind and polite'. Otherwise case might happen that all threads are being 'yielded' and that system stops because they are all giving resources to one another.

Tests:





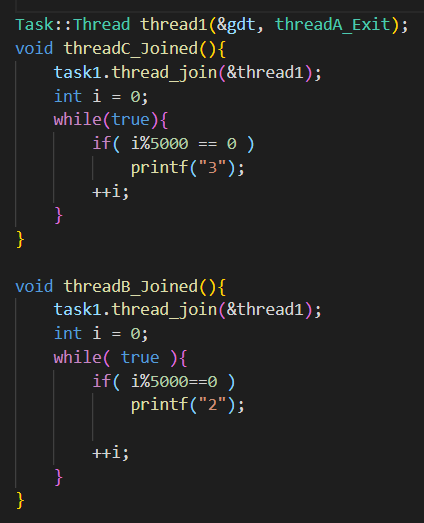


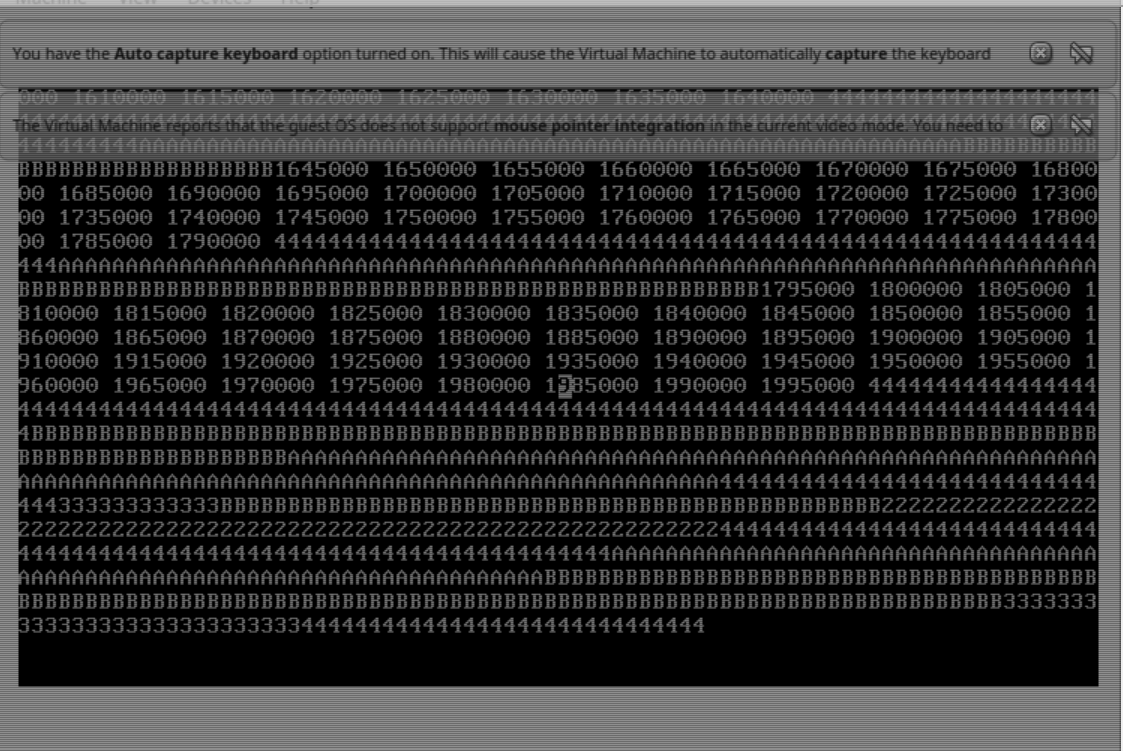
*We can see that even though both task 2’s main thread and task 2’s thread ‘D’ are performing the same thing. Value of main Thread’s counter is 10 times higher. This happened due to accumulated yielding of the thread ‘D’ allowing main thread to have more incrementations.*

Thread joining:

In order to perform joining ThreadTableEntry is needed. When join is called thread immediately stops and starts waiting for termination of desired thread. Number of joined threads in its thread table entry increases and callee thread doesn't continue until this number reaches zero value.

Tests:





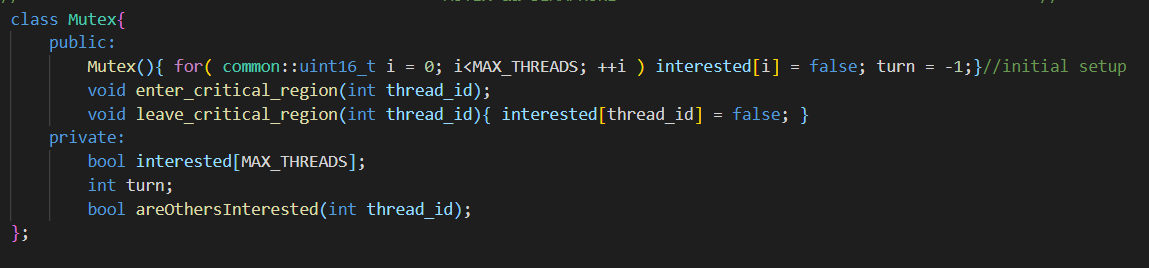
*Only after thread one ( the one outputing numbers until 2 000 000 ) is finished do thread 2 and thread 3 start outputing their values.*

Mutex, Semaphore and Producer consumer:

In order to implement a mutex Peterson's generalized form of the algorithm is used. Class mutex holds an array of interested threads and using it together with 'turn' variable is able to perform mutual exclusion.

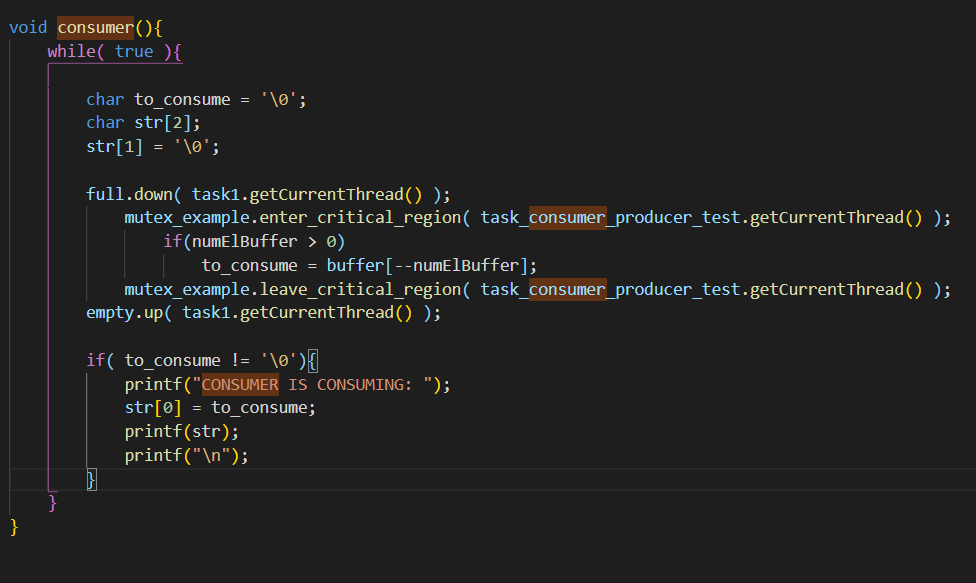
For semaphore implementation I used a mutex alongside with a count. Process waits if the count is 0. Semaphore can't be incremented if it is full and just continues without blocking. Its 2 main methods are up and down.

Tests:

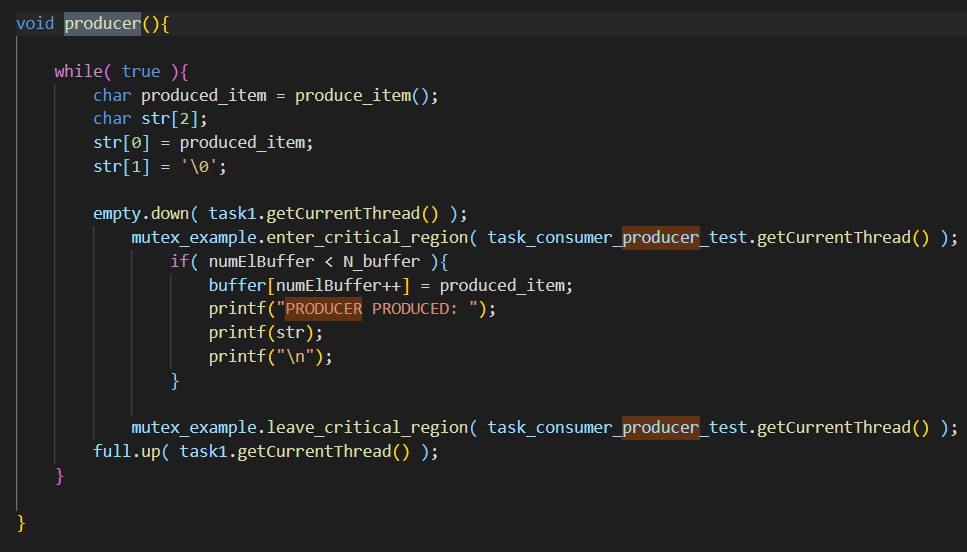


*Here we can see that consumer is performing consumption and producer production,*

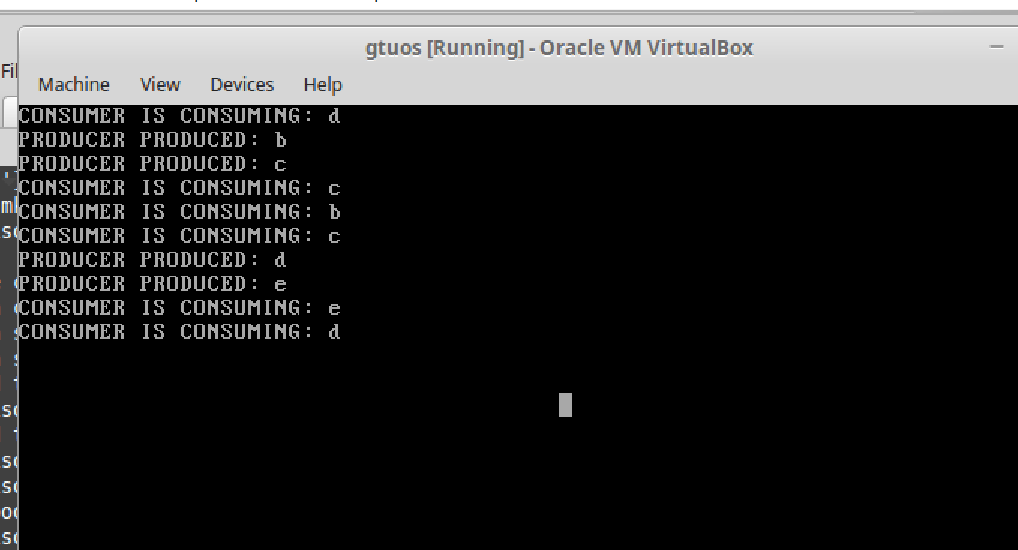
*Mutex is used in order to ensure mutual exclusion for the critical region they are accessing. Here 3 consumers and 2 producers are working in the same time. In this example I didn’t use semaphores and just wanted to see how it worked with mutexes only*



Consumer



Producer



*Here we have 3 consumers and 2 producers. Full potential can’t be seen because*

*producing 1 character is pretty was and takes much less than a time queantum.*

EXTRAS:

An additional problem I stumbled upon was that UNHANDLED interrupt would occur if a task/thread exited a function. In order to solve it additional scheduling method was used. First, manually the interrupt code was checked in ‘interrupts.cpp’. Than, afterwards manual scheduling was done. If number of threads of a task interrupt happened in was one or less, than task is removed from task manager’s queue making sure its cpustate is discarded. In another case when there are multiple threads, thread that caused the interrupt is removed from the queue and its cpustate is discarded.