GEBZE TECHNICAL UNIVERSITY
COMPUTER ENGINEERING

OPERATING SYSTEM LECTURE

HOMEWORK 1

REPORT 1

AHMET FURKAN KURBAN

1801042674

APRIL 8

Design Decisions

First of all, I asked teacher yusuf what the task in the multitasking part given to us was, and I got the answer, process. That's why I created a thread class. There are func , id , yield and join variables in the privacy of my thread class. I created Yield and join to control the thread schduler as well. I kept a thread array in the thread manager class to manage my threads easily. I have functions and thread schedule function (I'll talk about it later). I created a thread with the create_thread function. Thread initialize function runs when I call create_thread and assigns the function given as a parameter to cpustate's eip variable. I put the threads I created into the array that I created with the thread_add function of the thread manager class. Then I sent these thread managers to the processes. I kept a thread manager object inside the task class. In order to provide easy access to my threads via Task. Then I added the created tasks to the task manager and the program started to run. The architecture is as follows.

Thread->ThreadManager->Task->TaskManager

Thread Structure

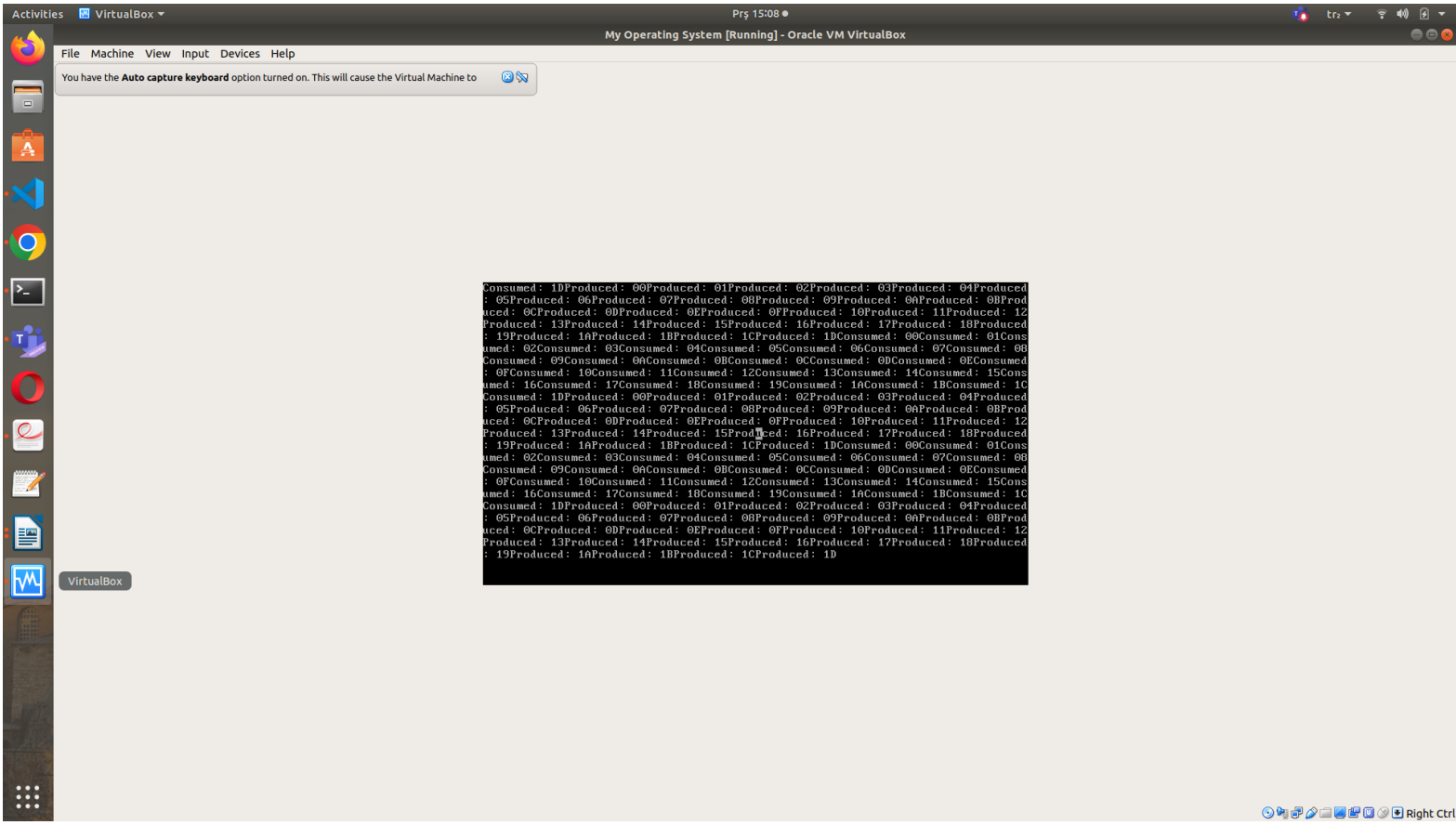
I set the thread manager class as friend. I kept the stack , cpustate , id , yield, func and join values in it as in the task. I wrote their get set functions. Unlike these, I have a Thread_initialize function, which assigns the given function to the eip value of cpustate as I mentioned in the introduction. I also have a thread manager class, which includes functions that perform thread array, create , terminate ,yield and join operations. I control the transition between threads with the Thread schdedule function.

Thread Scheduling

I put the CPU stat I got from the thread scheduler into the process schedule function. Thus, I was able to change my cpustat between threads and between processes. In the thread schedule, I took some precautions for join and yield, unlike the process schedule. If it has yielded, I increase the current thread and move on to the other thread. In the yield condition, if the function I want to run is waiting for someone and it hasn't finished yet, I'm looking at the next one.

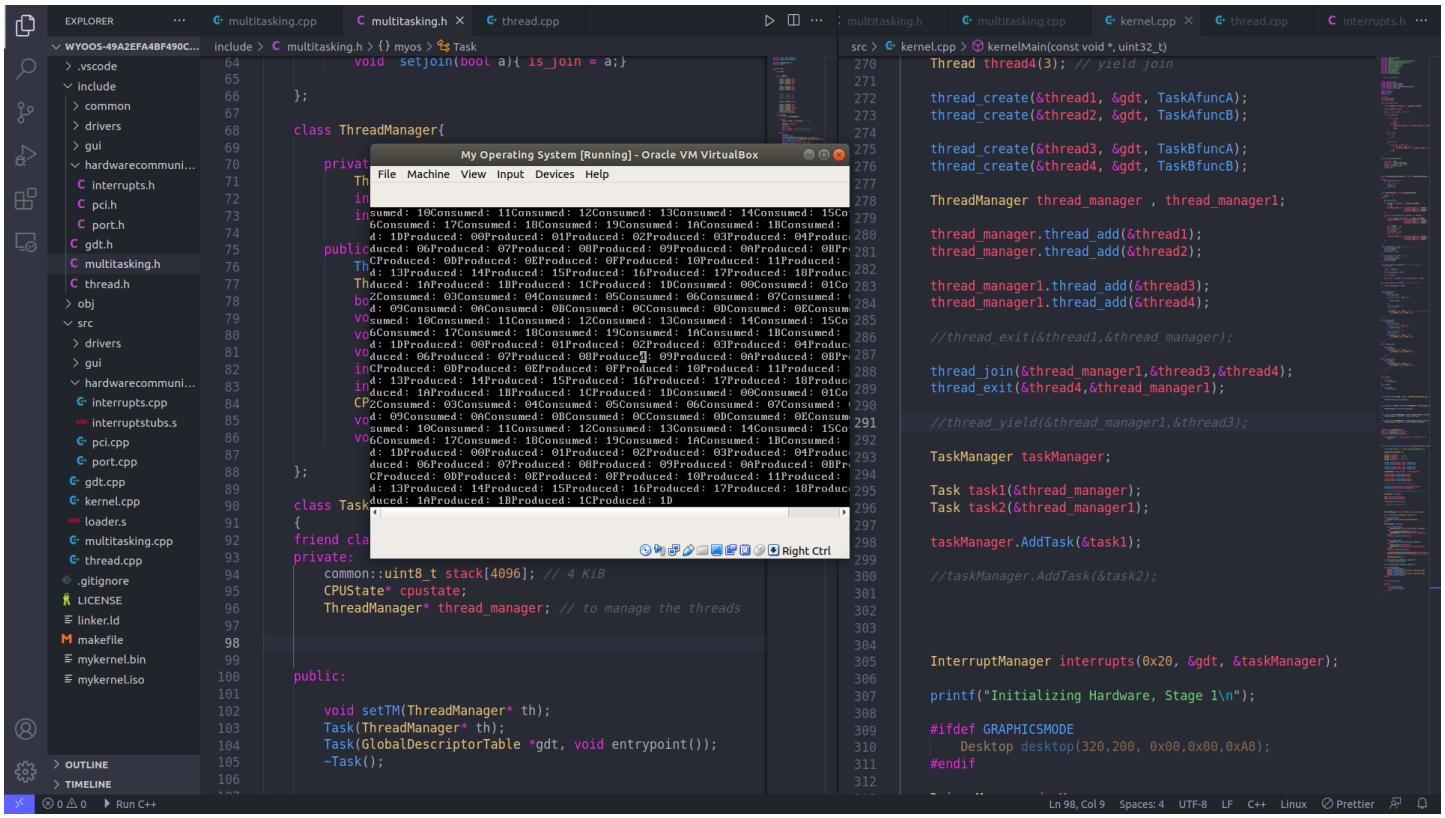
Critical Region

Without Peterson Algorithm:



I changed the same global value in producer consumer functions. Race conditions occur when 2 threads try to change the global value at the same time.

With Peterson Algorithm:



I applied the peterson algorithm to prevent this. I call the enter_region function before entering the critical section. Then leave_region function when leaving the critical section, so that 2 threads cannot reach the global value at the same time. So there is no race condition anymore.