Operating System Project I

Zafer Altay

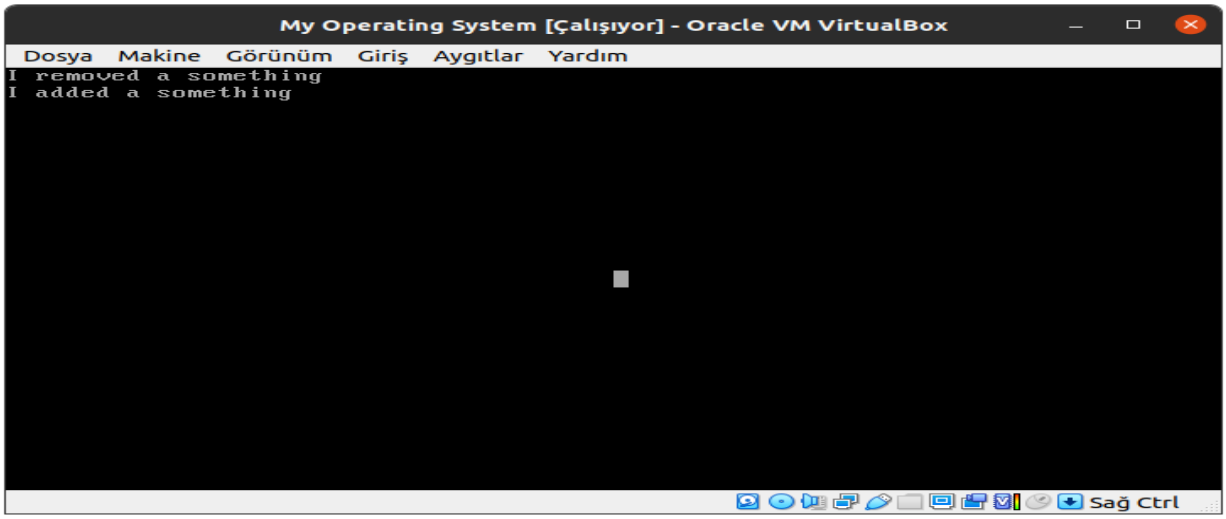
161044063

Design Decision

I am adding or deleting an item from the buffer in my producer-consumer algorithm.

- ### Producer-Consumer with Race Conditions :

  Here, threads can access the common areas without any blocking.The program hangs because it exceeds the bound of the array due to race condition.



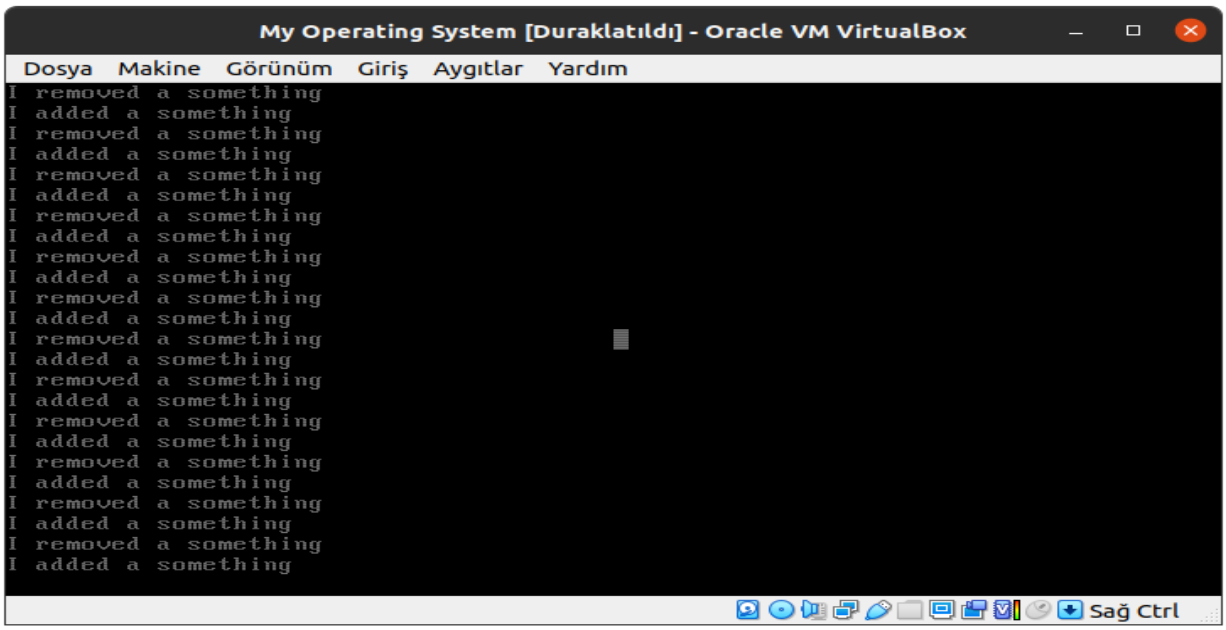

```
void consumer()
{

    int mynum=0;
    while(true){

        //while(empty==10);
        //enter_criticalregion(mynum);


        //while(lock==0) lock++; //----------------> Race Condition and Lock

        full--;
        empty++;
        mybuffer[count22]='e';
        count22--;



        int i=0;
        while(i<100000000){i++;}
        printf("I removed a something\n");

        //leave_criticalregion(mynum);
        //lock--; //--------------------------------->Race Condition and Lock

    }
}
```

As seen in the photo above, there is no restriction to access the buffer and global values.

(Race Condition Lock): This is one of the solutions developed by people, but it is not a solution, it continues to create race condition.

- ## Producer-Consumer with Peterson :

  In Peterson's solution, it works without any problem because I locked it at the entrances and exits to the critical area.



```
void consumer()
{

    int mynum=0;
    while(true){

        while(empty==10);
        enter_criticalregion(mynum);


        //while(lock==0) lock++; //----------------> Race Condition and Lock
        int temp=0;
        full--;
        empty++;
        mybuffer[count22]='e';
        count22--;



        int i=0;
        while(i<100000000){i++;}
        printf("I removed a something\n");

        leave_criticalregion(mynum);
        //lock--; //------------------------------->Race Condition and Lock

    }
}
```

Here is my critical region between the enter region and leave region that I lock.Since there can only be 1 thread in these regions at the same time, there is no race condition.These are peterson's solution functions:

```
#define FALSE 0
#define TRUE 1
#define N 2


int turn=0;
int interested[2]={0};

void enter_criticalregion(int process){
    int other;
    other=1-process;
    interested[process]=TRUE;
    turn=process;
    while(turn==process && interested[other]==TRUE);

}

void leave_criticalregion(int process){
    interested[process]=FALSE;
}
```

- Main Thread:

```
extern "C" void kernelMain(const void* multiboot_structure, uint32_t /*multiboot_magic*/)
{



    GlobalDescriptorTable gdt;

/*  THREAD FONKSİYONLARI DENEME
    Task task1(&gdt, taskA,1);
    Task task2(&gdt, taskB,2);
    Task task5(&gdt, taskE,5);
    Task task3(&gdt, taskC,3);
    Task task4(&gdt, taskD,4);
    Task task6(&gdt, taskF,6);

    taskManager.pthread_create(&task1);
    taskManager.pthread_create(&task5);
    taskManager.pthread_create(&task3);
    taskManager.pthread_create(&task2);
    taskManager.pthread_create(&task4);
    taskManager.pthread_create(&task6);
 */

    Task task1(&gdt, producer,1);
    Task task2(&gdt, consumer,2);

    taskManager.pthread_create(&task1);
    taskManager.pthread_create(&task2);
```

This is my main thread. I keep the threadid for the thread, so I can know which thread it is, I gave the ids here manually.We will use this for other functions.Main thread starts a tasks here.

Threads in comments are created to try other thread functions.

- Thread Join :

```
void TaskManager::pthread_join(int num){

    int threadflag=0;
    while(threadflag==0){
        threadflag=1;
       for (int i = 0; i < numTasks; ++i)
       {
            if (num==tasks[i]->taskNum)
            {
                threadflag=0;
                break;
            }
       }
    }


}
```

Here, the join function is constantly checking whether the thread id is in the task array, if the task isnt in array, it cannot find the id and can go out from loop and continue to work.

- ## Thread Yield :

```
void TaskManager::pthread_yield(int num){
    int counter;
    for (int i = 0; i < numTasks; i++)
    {
        if (num==tasks[i]->taskNum)
        {
            Task *t=tasks[i];

            for (counter=i ; counter <numTasks ; counter++)
            {
                tasks[counter]=tasks[counter+1];
            }
            tasks[numTasks]=t;
            return;
        }
    }
}
```

Here, scrolling is done towards the yielding thread. The yielding thread is put at the end of the list. This will change the order.

- ## Thread Create :

```
bool TaskManager::AddTask(Task* task)
{
    if(numTasks >= 256)
        return false;
    tasks[numTasks++] = task;
    return true;
}

bool TaskManager::pthread_create(Task* task)
{
    AddTask(task);
}
```

I got help from the library which is already written here.

- Thread Terminate :

```
void TaskManager::pthread_terminate(int num){
    int counter;
    for (int i = 0; i < numTasks; i++)
    {
        if (num==tasks[i]->taskNum)
        {
            tasks[i]->taskNum=-99;
            counter=i;
            for ( ;counter <numTasks ; )
            {

            }

            numTasks--;
            return;
        }
    }
}
```

Whichever thread is calling here, it stays in an endless loop and becomes unusable, so we can count the thread as terminated.