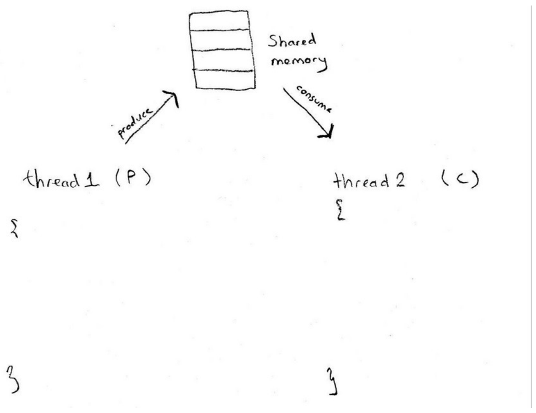
CSE312 – Operating Systems – HW1 Report
Ömer Faruk Bitikçioğlu - 161044010

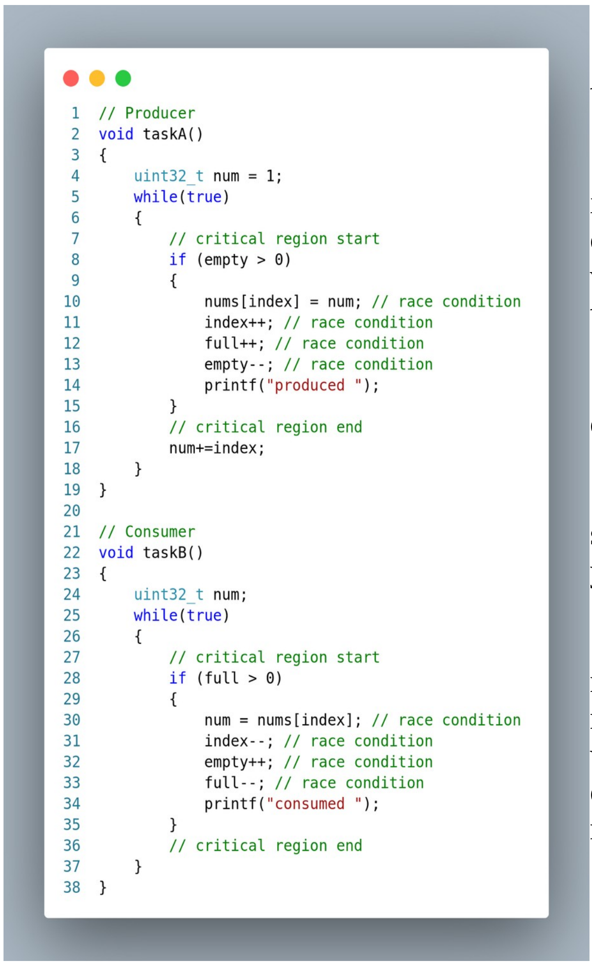
Preparation
For the first part of the homework, I successfully downloaded the 15th video's codes and run the operating system in my virtual machine.

Design
We as an operating system programmer, need to implement a multi threading library to achieve creating, terminating, yielding, and joining the threads.



We need at least two threads communicating with each other in a producer-consumer relationship. To do this I will implement two functions, one of them is producer which is producing some data and place it to the buffer, the other one is the consumer which uses the data in the buffer.

I used the multitasking library which includes task manager and tasks created earlier by the content creator. Edited the tasks to accomplish a producer-consumer fashion. They look like this right now:



```c
// Producer
void taskA()
{
    uint32_t num = 1;
    while(true)
    {
        // critical region start
        if (empty > 0)
        {
            nums[index] = num; // race condition
            index++; // race condition
            full++; // race condition
            empty--; // race condition
            printf("produced ");
        }
        // critical region end
        num+=index;
    }
}

// Consumer
void taskB()
{
    uint32_t num;
    while(true)
    {
        // critical region start
        if (full > 0)
        {
            num = nums[index]; // race condition
            index--; // race condition
            empty++; // race condition
            full--; // race condition
            printf("consumed ");
        }
        // critical region end
    }
}
```

In this design I used the taskA as producer, and taskB as the consumer thread.

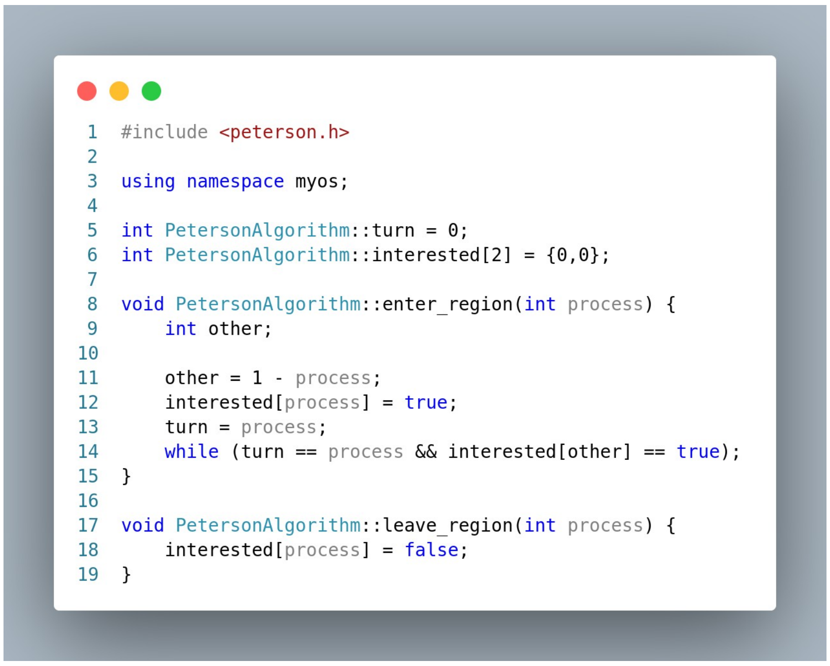
Producer puts a number to the empty slot and increments the index, full count, decrement the empty count; consumer takes the number indicated by index value, decrements the index and full count, increments the empty count.

Full, counts the full spaces in the buffer. Empty, counts the empty spaces in the buffer.

I showed the places where critical region should start and end. I didn't implement the critical regions yet.

Critical regions should protect the shared memory, and when interrupt happens the other thread must be avoided to change the final result. I mean, whenever some thread enters the critical region, the other threads should be avoided to enter the critical region and change the values of shared memories.

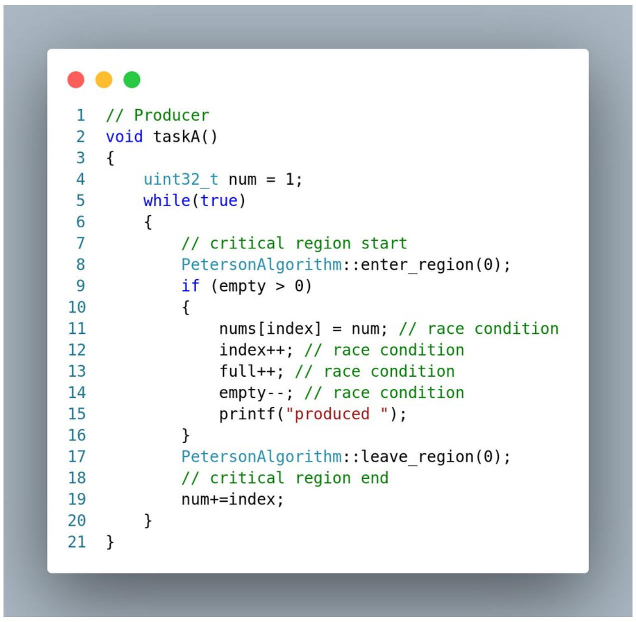
To achieve this, we will implement the Peterson's algorithm as below:



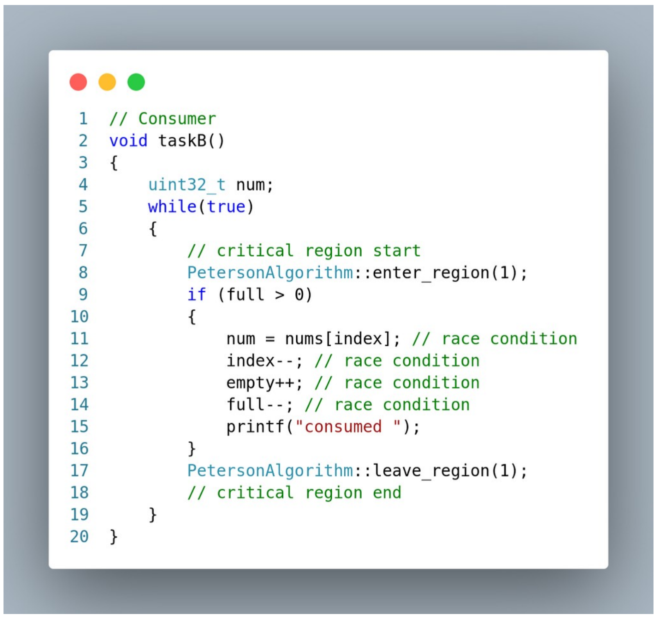
```cpp
#include <peterson.h>

using namespace myos;

int PetersonAlgorithm::turn = 0;
int PetersonAlgorithm::interested[2] = {0,0};

void PetersonAlgorithm::enter_region(int process) {
    int other;

    other = 1 - process;
    interested[process] = true;
    turn = process;
    while (turn == process && interested[other] == true);
}

void PetersonAlgorithm::leave_region(int process) {
    interested[process] = false;
}
```

And , I used the peterson's algorithm to lock critical regions to avoid race conditions.

Now, threads looks like this:



```cpp
// Producer
void taskA()
{
    uint32_t num = 1;
    while(true)
    {
        // critical region start
        PetersonAlgorithm::enter_region(0);
        if (empty > 0)
        {
            nums[index] = num; // race condition
            index++; // race condition
            full++; // race condition
            empty--; // race condition
            printf("produced ");
        }
        PetersonAlgorithm::leave_region(0);
        // critical region end
        num+=index;
    }
}
```

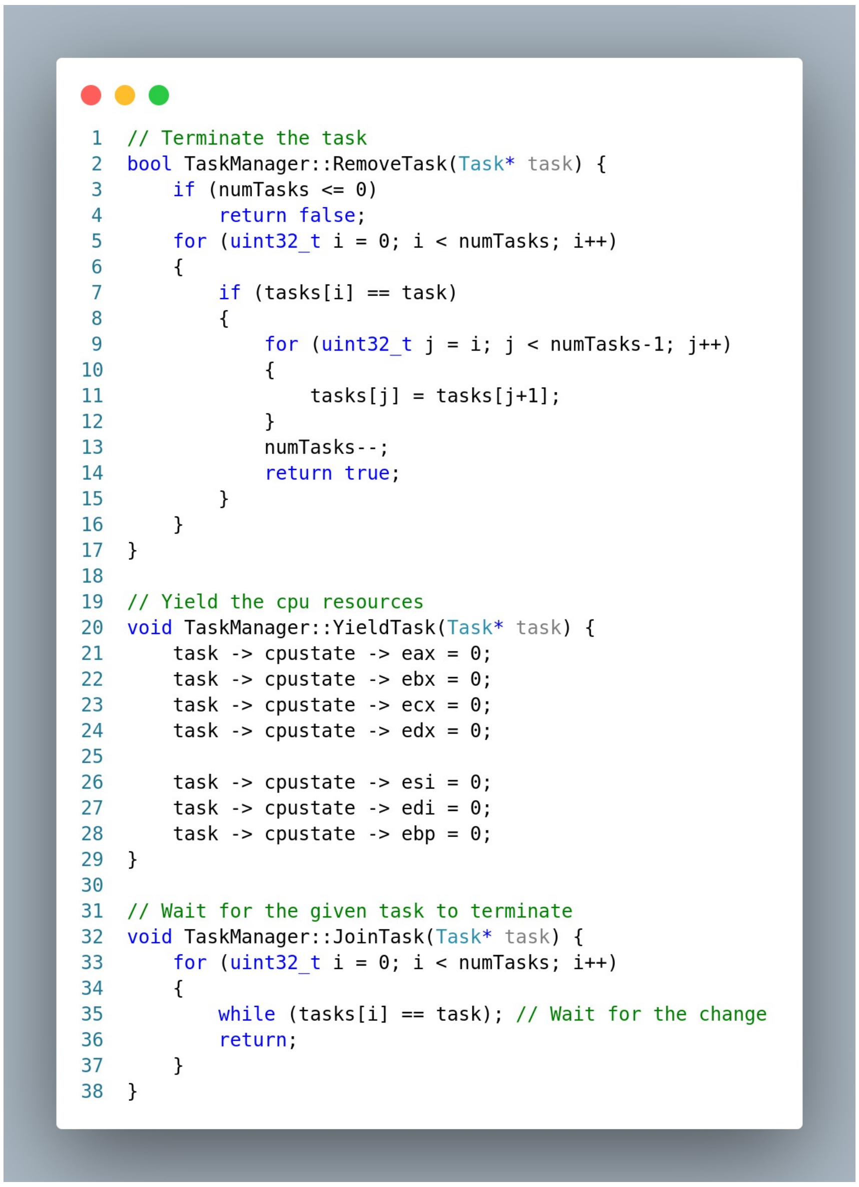


```cpp
// Consumer
void taskB()
{
    uint32_t num;
    while(true)
    {
        // critical region start
        PetersonAlgorithm::enter_region(1);
        if (full > 0)
        {
            num = nums[index]; // race condition
            index--; // race condition
            empty++; // race condition
            full--; // race condition
            printf("consumed ");
        }
        PetersonAlgorithm::leave_region(1);
        // critical region end
    }
}
```

We need 3 more functions to terminate, yield and join the threads. I simply implemented the terminate function as it searches the given task and if it finds shifts the rest of the tasks to override the desired task. We decrement the numTasks to ignore the remaining additional task.

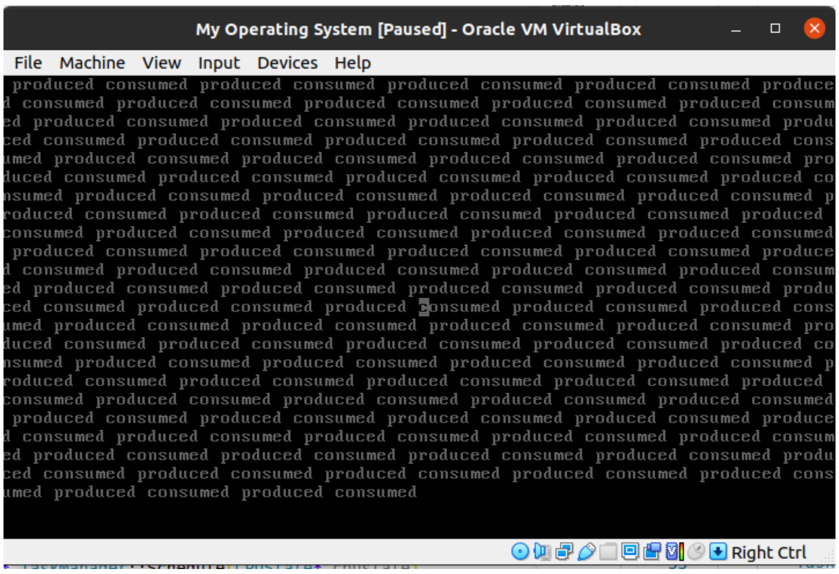
For the yield part I implemented the function as basically setting the task's cpustate values to 0. By doing this we yield the cpu resources.

To join a thread, I first searched the given task in the tasks array, when I find it I busy waited for this task to change. Because when the task is terminated we set the associated field as another task (shifting the tasks).



```cpp
// Terminate the task
bool TaskManager::RemoveTask(Task* task) {
    if (numTasks <= 0)
        return false;
    for (uint32_t i = 0; i < numTasks; i++)
    {
        if (tasks[i] == task)
        {
            for (uint32_t j = i; j < numTasks-1; j++)
            {
                tasks[j] = tasks[j+1];
            }
            numTasks--;
            return true;
        }
    }
}

// Yield the cpu resources
void TaskManager::YieldTask(Task* task) {
    task -> cpustate -> eax = 0;
    task -> cpustate -> ebx = 0;
    task -> cpustate -> ecx = 0;
    task -> cpustate -> edx = 0;

    task -> cpustate -> esi = 0;
    task -> cpustate -> edi = 0;
    task -> cpustate -> ebp = 0;
}

// Wait for the given task to terminate
void TaskManager::JoinTask(Task* task) {
    for (uint32_t i = 0; i < numTasks; i++)
    {
        while (tasks[i] == task); // Wait for the change
        return;
    }
}
```

Our operating system's output now looks like this:



```
 produced consumed produced consumed produced consumed produced consumed produce
d consumed produced consumed produced consumed produced consumed produced consum
ed produced consumed produced consumed produced consumed produced consumed produ
ced consumed produced consumed produced consumed produced consumed produced cons
umed produced consumed produced consumed produced consumed produced consumed pro
duced consumed produced consumed produced consumed produced consumed produced co
nsumed produced consumed produced consumed produced consumed produced consumed p
roduced consumed produced consumed produced consumed produced consumed produced
consumed produced consumed produced consumed produced consumed produced consumed
 produced consumed produced consumed produced consumed produced consumed produce
d consumed produced consumed produced consumed produced consumed produced consum
ed produced consumed produced consumed produced consumed produced consumed produ
ced consumed produced consumed produced consumed produced consumed produced cons
umed produced consumed produced consumed produced consumed produced consumed pro
duced consumed produced consumed produced consumed produced consumed produced co
nsumed produced consumed produced consumed produced consumed produced consumed p
roduced consumed produced consumed produced consumed produced consumed produced
consumed produced consumed produced consumed produced consumed produced consumed
 produced consumed produced consumed produced consumed produced consumed produce
d consumed produced consumed produced consumed produced consumed produced consum
ed produced consumed produced consumed produced consumed produced consumed produ
ced consumed produced consumed produced consumed produced consumed produced cons
umed produced consumed produced consumed
```