GIT Department of Computer Engineering CSE 312/504 - Spring 2022 Homework # Report

Sefa Çiçek
1801042657

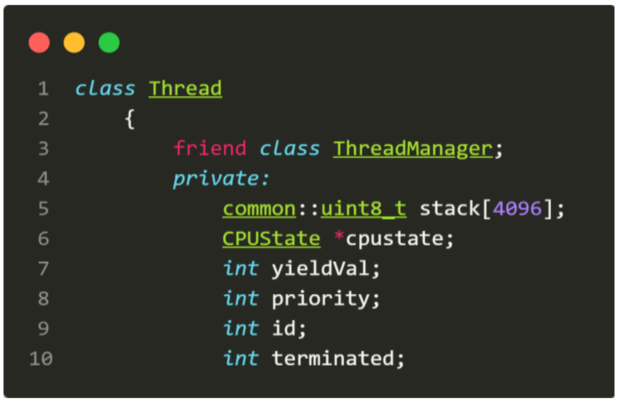
1. Design Decisions

Firstly, I have decided to solve binding of thread and process. I've figured out this situation with 'thread manager'. Thread manager behaves like a task manager. It has thread array and some properties to determine which thread will be run. A thread has id, priority num, yield and terminated value unlike the task. Id is used like an index for thread array. Priority helps to determine which thread will be run if yield occurs. Also, terminated and yield value is used for determining which threads will be terminated or yielded.

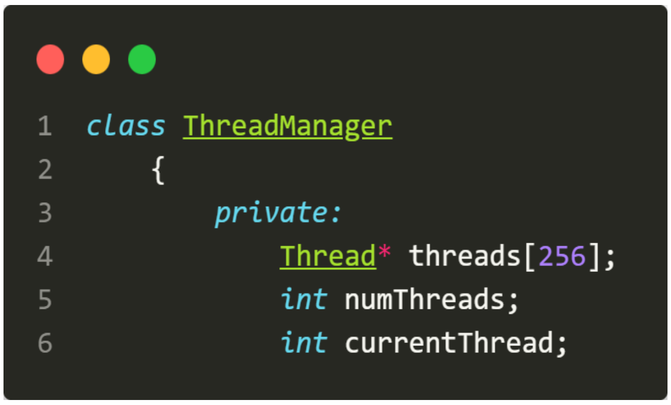
- threadCreate : I have assigned priority and id to some thread objects and I called threadCreate function to bind objects and tasks. Binding requires gdt and entrypoint. Entrypoint is our tasks. After binding, our objects are now actual threads.
- threadYield : I have decremented the priority of the thread which asks for yield and then I have set the yield value of thread to 1. When thread scheduler works, it checks yield value of thread. If yield value is 1 and next thread has same or higher priority than current thread, scheduler passes the current thread and executes next thread. Although yield value of thread is 1, the scheduler does not pass current thread, if next thread has lower priority.
- threadTerminate : I have set terminated value of thread to 1. Scheduler passes the thread if terminated value of thread is 1.
- threadJoin : I have checked the terminated value of thread1. If it is 0, it means thread1 is not terminated, that's why thread2 can not be executed.

2. Thread Structure

I've already mentioned the thread structure above. I also added the classes picture.



```
class Thread
    {
        friend class ThreadManager;
        private:
            common::uint8_t stack[4096];
            CPUState *cpustate;
            int yieldVal;
            int priority;
            int id;
            int terminated;
```

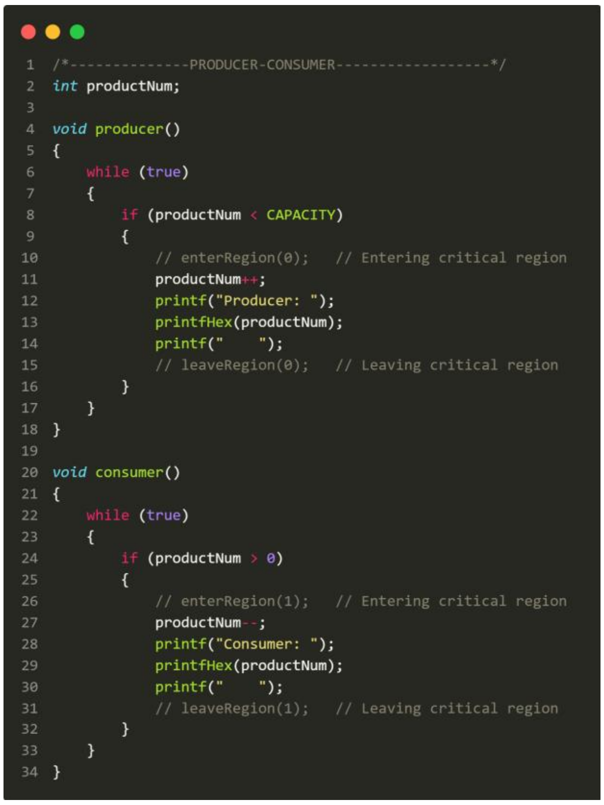


```
class ThreadManager
    {
        private:
            Thread* threads[256];
            int numThreads;
            int currentThread;
```

3. Scheduling

Interrupt manager calls schedule function of task manager. I have added schedule function to thread manager. When task manager scheduler executes, it calls thread scheduler. I have checked terminate and yield value of threads in addition to the task manager. I have incremented current thread number to execute next thread, if termination of thread requires. Also, current thread number is incremented or stayed same according to the yield and priority values of thread.

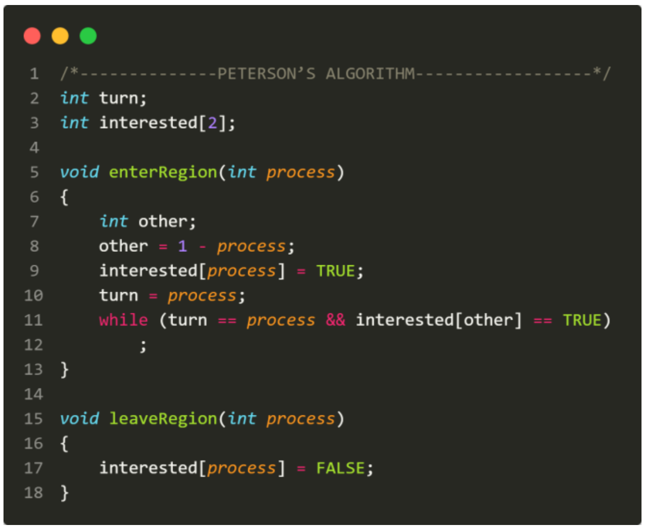
4. Race Condition



```c
/*--------------PRODUCER-CONSUMER-----------------*/
int productNum;

void producer()
{
    while (true)
    {
        if (productNum < CAPACITY)
        {
            // enterRegion(0);   // Entering critical region
            productNum++;
            printf("Producer: ");
            printfHex(productNum);
            printf("    ");
            // leaveRegion(0);   // Leaving critical region
        }
    }
}

void consumer()
{
    while (true)
    {
        if (productNum > 0)
        {
            // enterRegion(1);   // Entering critical region
            productNum--;
            printf("Consumer: ");
            printfHex(productNum);
            printf("    ");
            // leaveRegion(1);   // Leaving critical region
        }
    }
}
```



```c
/*--------------PETERSON’S ALGORITHM------------------*/
int turn;
int interested[2];

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

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

There is a possibility of race condition existence. If an interrupt occurs in producer or consumer function, there may be unexpected result. productNum is a global variable which is used by producer and consumer. While producer increments productNum, maybe interrupt can occur, if this happens, consumer executes. Consumer cannot know producer’ll increment productNum value. After consuming operation (decrementing of productNum), interrupt occurs again. Producer cannot know that consumer consumes the product that’s why there may be unexpected result.

Peterson’s solution can solve this problem with adding two functions (enterRegion and leaveRegion). Critical regions have been identified. Thus, threads can used same data without conflict.