CSE312-OPERATING SYSTEM

HOMEWORK REPORT 1

1801042630

MEHMET AVNİ ÇELİK

Design Decisions

- The given source code by Viktor Engelmann provides some multitasking designs and functions. The implemented multithreading library based on his design. Some other functions which is required for the homework are also implemented by following his design.

Main Thread Structure

- Simple multithreading library is provided.
- CreateThread,TerminateThread,YieldThread and JoinThread functions are implemented to the source code.
- Basic producer-consumer correlation is presented.
- Peterson's Algorithm is used for synchronization.

Multithreading Library:

1. Create Thread:

Threads are created if there is not any working thread in the queue. This is also controlled by static isAvailable variable which is true by default. Static variable is used to see if any other object did changed it. If it is change to false,this means some other object is in progress, so there may be race condition that is also prevented. The implementation is accessible below.

```
bool TaskManager::isAvailable=true;
bool TaskManager::CreateThread(Task* task)
{
    if(numTasks >= 256)
        return false;
    if(isAvailable==true){          //if there is not any process in the queue,thread will be created.
        tasks[numTasks++] = task;
        task->counter++;
    }
    return true;
}
```

2. Terminate Thread:

When the thread that will be terminated is found in the queue, it will be deleted and instead of it, the queue will be shifted just like ArrayList data structure. The total amount of threads in the queue will be decremented by one. The implementation is accessible below.

```
bool TaskManager::TerminateThread(Task* task){
    if(numTasks < 0)
        return false;
    for(int i=0;i<numTasks;i++){
        if(tasks[i]==task){
            task->isEnded=true;
            TaskManager:: isAvailable=true;     //to make sure the thread is ended and some other thread can work
            while(i!=numTasks-1){
                tasks[i]=tasks[i+1];
                i++;
            }
            numTasks--;
            break;
        }
    }
    return true;
}
```

3. Yield Thread:

The thread that works for a long time will release the CPU and be sent to the end of the queue, even if the thread is not terminated. The implementation is similar to terminate thread function. The implementation is accessible below.

```
bool TaskManager::YieldThread(Task* task){ // thread that works for a long time pushed to the end of the queue without terminating
        tasks[numTasks]=task;
        int i=0;
        for(int i=0;i<numTasks;i++){
            if(tasks[i]==task){
                while(i!=numTasks){
                    tasks[i]=tasks[i+1];
                    i++;
                }
                break;
            }
        }
        task->isOvertime=false;
        return true;
}
```

4. Join Thread:

The task(thread) which is sent to join thread function should be terminated for resuming to thread execution. Join thread function provides that if the thread is not terminated, upcoming threads will not be executed until the task is finished.

```
bool TaskManager::JoinThread(Task* task,int joinValue){ // make sure it is terminated.If it is not terminated,next threads wont be exectued.
    if(task->isEnded==false)
        TaskManager:: isAvailable=false;

    else if(task->isEnded==true)
        TaskManager:: isAvailable=true;

    else isAvailable=false;

    return isAvailable;
}
```

Scheduling Algorithm

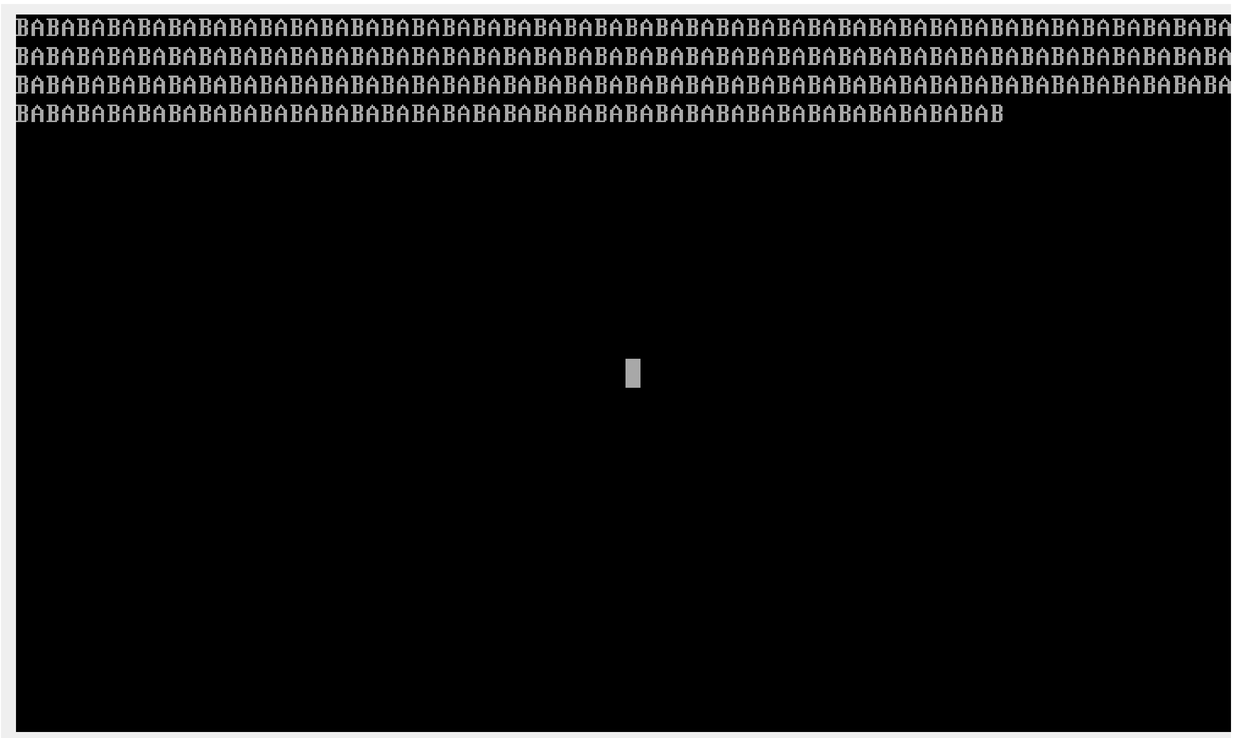
Peterson's Algorithm which is showed in the class is used to provide synchronization. Also the unit tests are implemented in the kernel file. The race conditions are prevented with this algorithm. You can see the outputs with and without race condition below.

```
//-----------------PETERSON ALGORITHM FOR SCHEDULING----------------------------------//
void enter_region(int process){
    int other = 1-process;
    interested[process] = TRUE;
    turn = process;
    while(turn == process && interested[other] == TRUE );
}

void leave_region(int process){
    interested[process] = FALSE;
}
//-----------------------------------------------------------------------------//
```

```
void taskA()
{
    while(true){
        enter_region(0);
        printf("A");
        leave_region(0);
    }

}
void taskB()
{
    while(true){
        enter_region(1);
        printf("B");
        leave_region(1);
    }
}
```

The Output <u>without</u> Peterson Algorithm is showed below(for race condition).

The Output <u>with</u> Peterson Algorithm is showed below(for race condition).



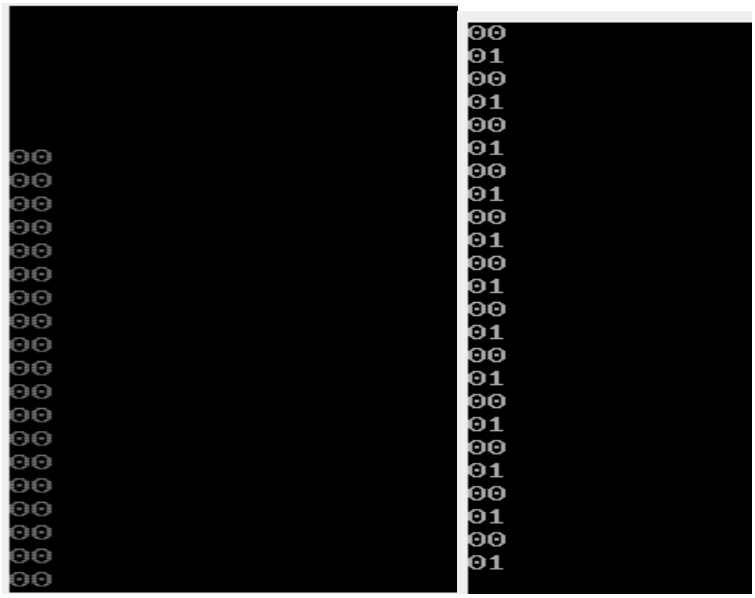
Producer-Consumer Correlation

Producer-consumer correlation is implemented as it is showed in the class. There are 2 global variable,one of them is an array and the other is integer to keep index. Producer produces and puts it into the array, and increment the index by 1. Consumer consumes and decrement the index by 1. Race condition is also prevented by Peterson Algorithm.

```
void producer(){
    while(true){
        if(total<1024){
            enter_region(0);
            productBuffer[total++]=1;
            printfHex(productBuffer[total]);
            printf("\n");
            leave_region(0);
        }
    }
}
```

```
void consumer(){
    while(true){
        if(total>0){
            enter_region(1);
            productBuffer[total--]=0;
            printfHex(productBuffer[total]);
            printf("\n");
            leave_region(1);
        }
    }
}
```

The Output <u>without</u> and <u>with</u> Peterson Algorithm is showed below(for race condition).



Note:

In the kernel.cpp file, all the unit tests are commented out to have a decent code to show producer-consumer relationship. Since the testing is difficult, more than two threads are creted and the functions are called by them for debugging.

You can see create,terminate,yield and join functions in the kernel file as shown below. Those are left in comment deliberately to execute in the demo. All my test cases are passed, I couldn't see anything unexpected.

```
//-----------------UNIT TESTING FOR JOIN THREAD---------------------------------//
//    taskManager.CreateThread(&thread1);          //thread A will be created.
//    taskManager.CreateThread(&thread2);          //thread B will be created.
//    taskManager.CreateThread(&thread3);          //thread C will be created. if you also comment thi
//    taskManager.JoinThread(&thread1,0);          //thread1.join will be called.
//    taskManager.CreateThread(&thread4);          //thread D will be called but not created due to jo
//------------------------------------------------------------------------//

//-----------------UNIT TESTING FOR YIELD THREAD---------------------------------//
//    taskManager.CreateThread(&thread1);          //thread A will be created.
//    taskManager.CreateThread(&thread2);          //thread B will be created.
//    taskManager.CreateThread(&thread3);          //thread C will be created.
//    taskManager.CreateThread(&thread4);          //thread D will be created.
//    taskManager.YieldThread(&thread3);          //thread C will be sent to the end.
//------------------------------------------------------------------------//

//-----------------UNIT TESTING FOR PETERSON ALGORITHM---------------------------------//
//    taskManager.CreateThread(&threadP1);          //thread A will be created.
//    taskManager.CreateThread(&threadP2);          //thread B will be created.
//------------------------------------------------------------------------//


//-----------------UNIT TESTING FOR PRODUCER-CONSUMER---------------------------------//
    taskManager.CreateThread(&thread_producer);         //producing
    taskManager.CreateThread(&thread_consumer);         //consuming
//------------------------------------------------------------------------//
```

Thank you for your time...