CSE312 – Operating Systems

Assignment #1

Multithread Library Implementation

Design & Test Report

Date

07.04.2022

Author

Gökbey Gazi KESKİN

1901042631

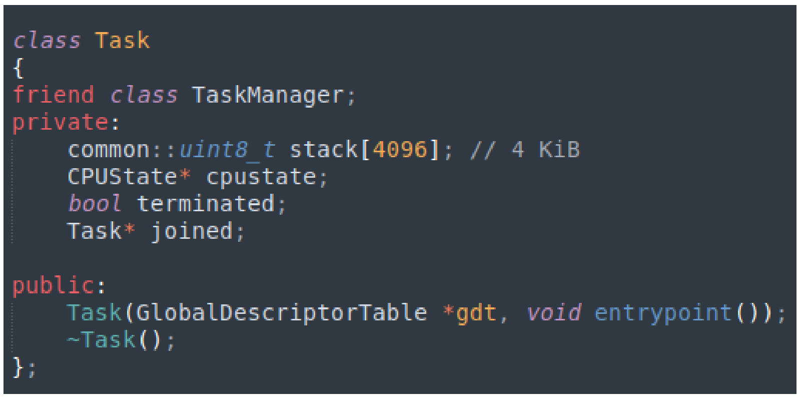
Table of Contents

Implementation

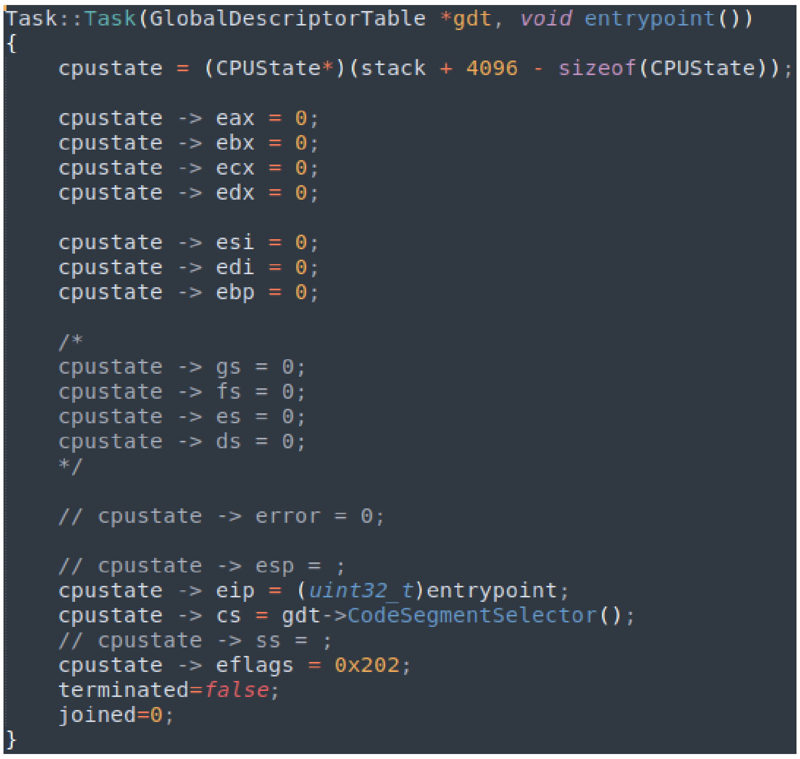
I used the already existing multitasking.cpp file which contains Task and TaskManager classes and extended these classes. First requirement of the assignment (Creating a thread) was already satisfied by the creator of the OS, so I added other 3 (terminating, joining, and yielding threads) and I also altered the already existing thread creation function (AddTask) to work properly with my new functions.

Note: Creator of the OS used the word Task while writing the multitasking library. So, I used the same word for compatibility with the original code. Words task and thread are used interchangeably through the code and the report.

Task Class



```
class Task
{
friend class TaskManager;
private:
    common::uint8_t stack[4096]; // 4 KiB
    CPUState* cpustate;
    bool terminated;
    Task* joined;

public:
    Task(GlobalDescriptorTable *gdt, void entrypoint());
    ~Task();
};
```

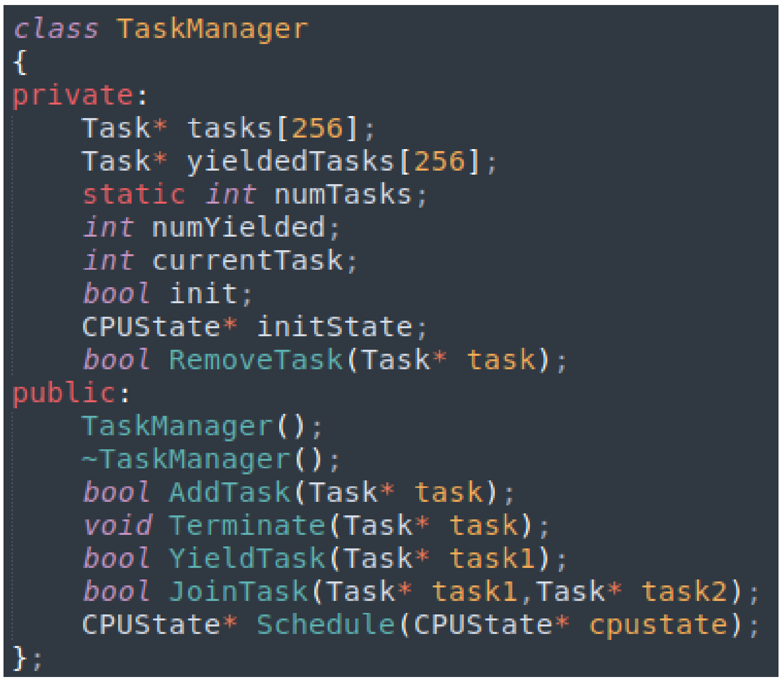


```
Task::Task(GlobalDescriptorTable *gdt, void entrypoint())
{
    cpustate = (CPUState*)(stack + 4096 - sizeof(CPUState));

    cpustate -> eax = 0;
    cpustate -> ebx = 0;
    cpustate -> ecx = 0;
    cpustate -> edx = 0;

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

    /*
    cpustate -> gs = 0;
    cpustate -> fs = 0;
    cpustate -> es = 0;
    cpustate -> ds = 0;
    */

    // cpustate -> error = 0;

    // cpustate -> esp = ;
    cpustate -> eip = (uint32_t)entrypoint;
    cpustate -> cs = gdt->CodeSegmentSelector();
    // cpustate -> ss = ;
    cpustate -> eflags = 0x202;
    terminated=false;
    joined=0;
}
```

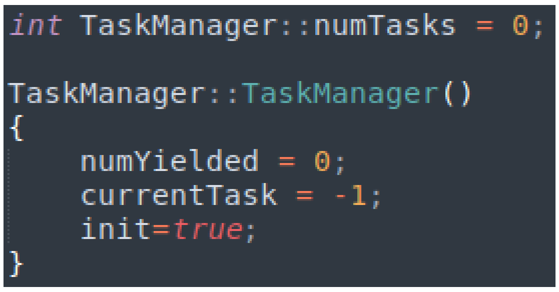
I added 2 private members to Task class. Boolean member terminated is used in two scenarios. First is regular termination of a thread and the second one is when 2 functions are joined, and one is waiting for other one to terminate.

When thread A is joined to thread B, thread A is assigned to thread B's joined field to re-add it to ready queue after B terminates. Joined is initially assigned to 0 (null) and terminated is assigned to false.

TaskManager Class



```
class TaskManager
{
private:
    Task* tasks[256];
    Task* yieldedTasks[256];
    static int numTasks;
    int numYielded;
    int currentTask;
    bool init;
    CPUState* initState;
    bool RemoveTask(Task* task);
public:
    TaskManager();
    ~TaskManager();
    bool AddTask(Task* task);
    void Terminate(Task* task);
    bool YieldTask(Task* task1);
    bool JoinTask(Task* task1,Task* task2);
    CPUState* Schedule(CPUState* cpustate);
};
```



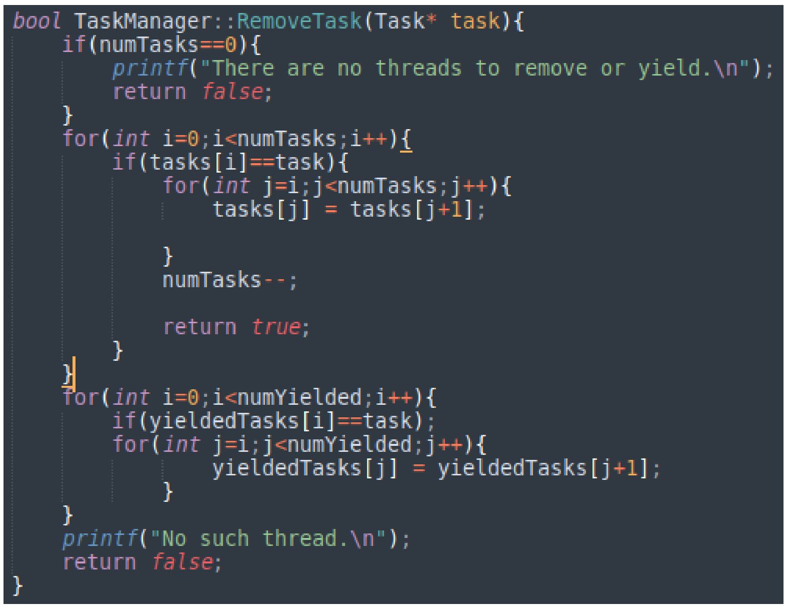
```
int TaskManager::numTasks = 0;

TaskManager::TaskManager()
{
    numYielded = 0;
    currentTask = -1;
    init=true;
}
```

Additional data fields

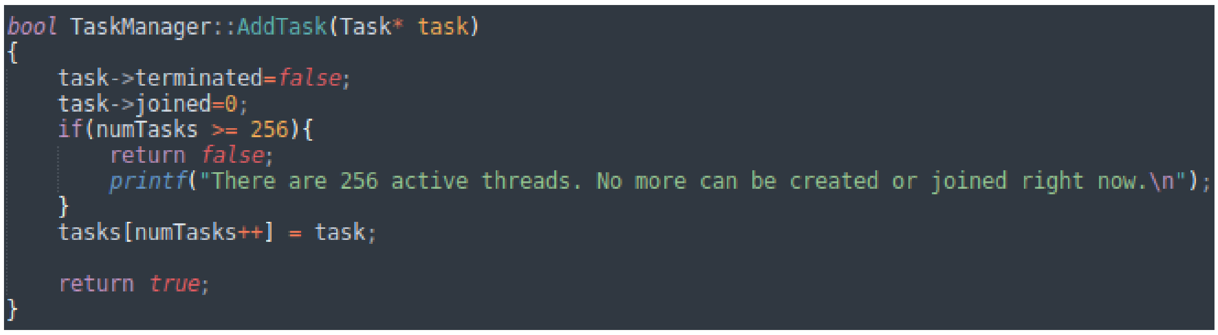
yieldedTasks & numYielded: These variables are used when a task is yielded. Elements of yieldedTasks are re-added to ready queue (tasks array) at the end of the clock cycle.

Init & initState: These 2 are used for storing the initial state of the cpu when the scheduling is done for the first time. When all threads are removed, cpustate is assigned to initState.

RemoveTask: Used frequently by the TaskManager functions. When a thread is terminated or yielded, it is used for taking it out from the ready queue. If the thread doesn't exist in ready queue but does exist in yieldedTasks array, takes it out from the yieldedTasks array since a thread can be removed while it is yielded.

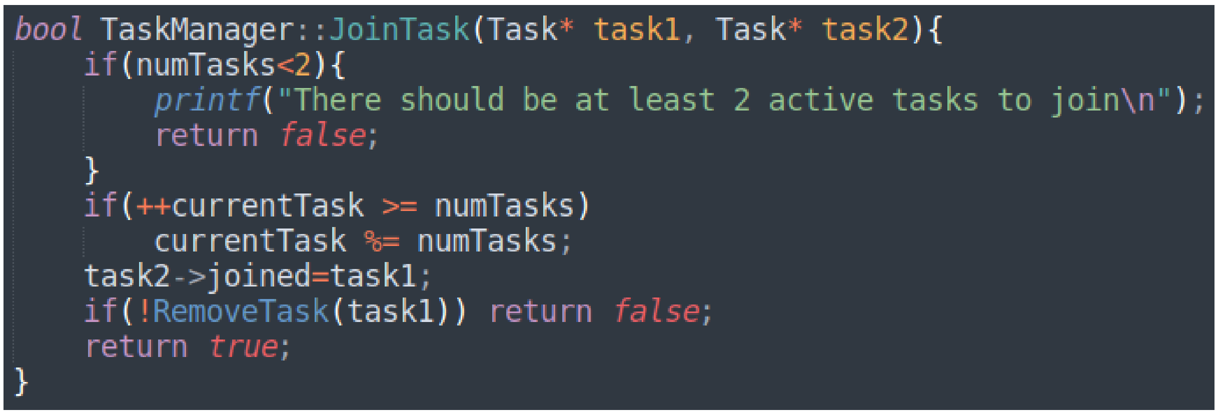


```
bool TaskManager::RemoveTask(Task* task){
    if(numTasks==0){
        printf("There are no threads to remove or yield.\n");
        return false;
    }
    for(int i=0;i<numTasks;i++){
        if(tasks[i]==task){
            for(int j=i;j<numTasks;j++){
                tasks[j] = tasks[j+1];

            }
            numTasks--;

            return true;
        }
    }
    for(int i=0;i<numYielded;i++){
        if(yieldedTasks[i]==task);
        for(int j=i;j<numYielded;j++){
                yieldedTasks[j] = yieldedTasks[j+1];
            }
    }
    printf("No such thread.\n");
    return false;
}
```



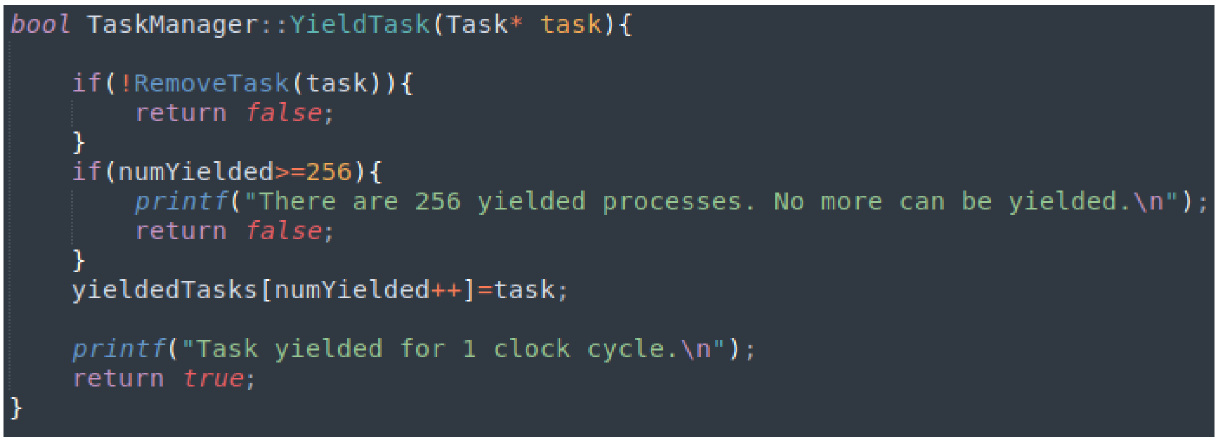
```
bool TaskManager::AddTask(Task* task)
{
    task->terminated=false;
    task->joined=0;
    if(numTasks >= 256){
        return false;
        printf("There are 256 active threads. No more can be created or joined right now.\n");
    }
    tasks[numTasks++] = task;

    return true;
}
```

AddTask: When a task is added or re-added, its fields related to JoinTask function are set to their initial values, rest is the same with the creator's implementation.



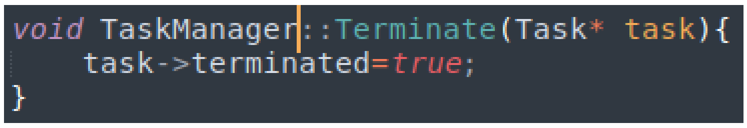
```
bool TaskManager::JoinTask(Task* task1, Task* task2){
    if(numTasks<2){
        printf("There should be at least 2 active tasks to join\n");
        return false;
    }
    if(++currentTask >= numTasks)
        currentTask %= numTasks;
    task2->joined=task1;
    if(!RemoveTask(task1)) return false;
    return true;
}
```

JoinTask: Makes task1 wait wherever it was on execution until task2 is finished. It basically fills the task2's joined field with task 1 and removes the task1 from ready-queue. Schedule function deals with the rest of the joining.

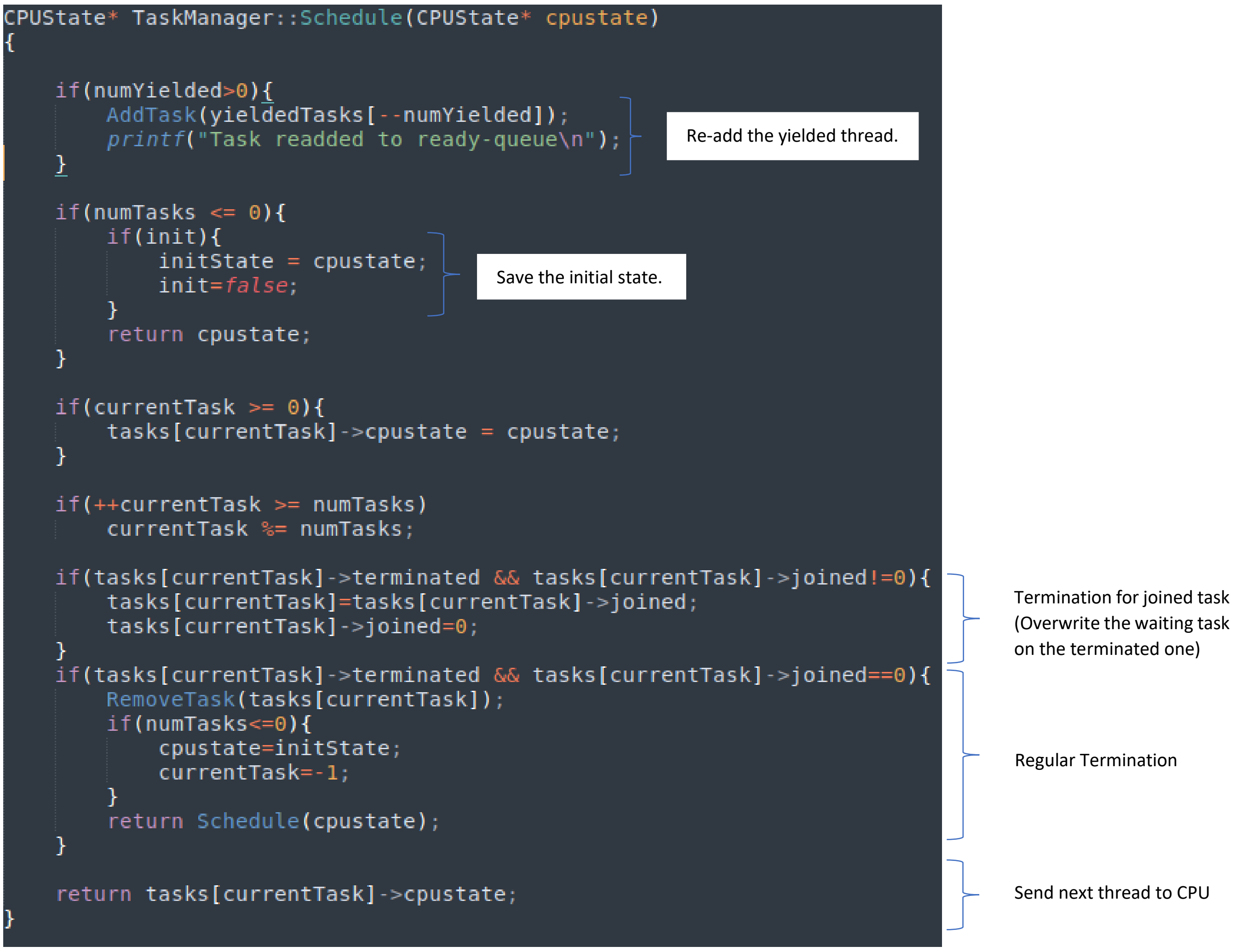


```
bool TaskManager::YieldTask(Task* task){

    if(!RemoveTask(task)){
        return false;
    }
    if(numYielded>=256){
        printf("There are 256 yielded processes. No more can be yielded.\n");
        return false;
    }
    yieldedTasks[numYielded++]=task;

    printf("Task yielded for 1 clock cycle.\n");
    return true;
}
```

YieldTask: Removes the task from the ready-queue and puts in in the yieldedTasks array. Schedule function puts the task back into the ready queue for the next cycle.


```
void TaskManager::Terminate(Task* task){
    task->terminated=true;
}
```

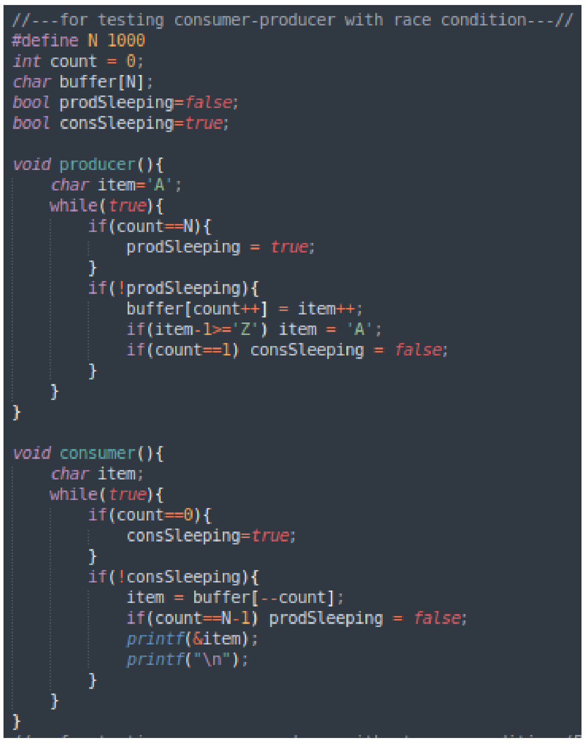
Terminate: sets task as terminated. Schedule function deals with the rest of the termination by calling the RemoveTask function if terminated is true.


```
CPUState* TaskManager::Schedule(CPUState* cpustate)
{

    if(numYielded>0){
        AddTask(yieldedTasks[--numYielded]);
        printf("Task readded to ready-queue\n");
    }

    if(numTasks <= 0){
        if(init){
            initState = cpustate;
            init=false;
        }
        return cpustate;
    }

    if(currentTask >= 0){
        tasks[currentTask]->cpustate = cpustate;
    }

    if(++currentTask >= numTasks)
        currentTask %= numTasks;

    if(tasks[currentTask]->terminated && tasks[currentTask]->joined!=0){
        tasks[currentTask]=tasks[currentTask]->joined;
        tasks[currentTask]->joined=0;
    }
    if(tasks[currentTask]->terminated && tasks[currentTask]->joined==0){
        RemoveTask(tasks[currentTask]);
        if(numTasks<=0){
            cpustate=initState;
            currentTask=-1;
        }
        return Schedule(cpustate);
    }

    return tasks[currentTask]->cpustate;
}
```
Re-add the yielded thread.
Save the initial state.
Termination for joined task (Overwrite the waiting task on the terminated one)
Regular Termination
Send next thread to CPU

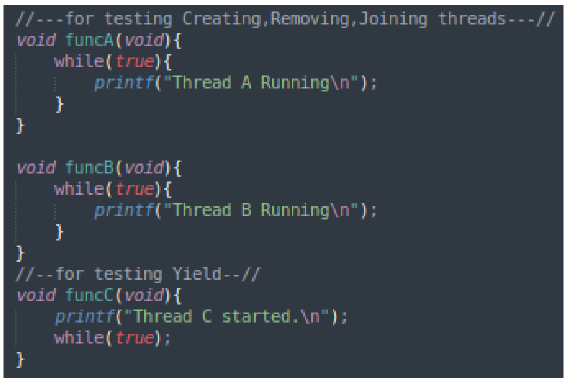
Schedule: I added the parts shown on the above image to Schedule function in order to make it work with my additional functions.

Tests

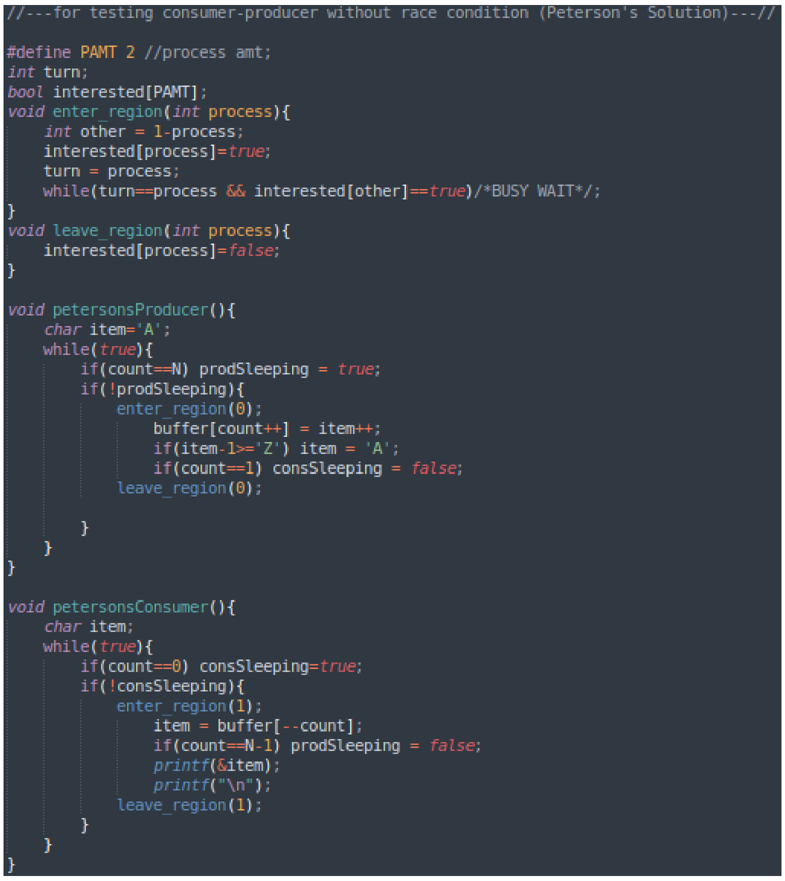
Threads



```c
//---for testing consumer-producer with race condition---//
#define N 1000
int count = 0;
char buffer[N];
bool prodSleeping=false;
bool consSleeping=true;

void producer(){
    char item='A';
    while(true){
        if(count==N){
            prodSleeping = true;
        }
        if(!prodSleeping){
            buffer[count++] = item++;
            if(item-1>='Z') item = 'A';
            if(count==1) consSleeping = false;
        }
    }
}

void consumer(){
    char item;
    while(true){
        if(count==0){
            consSleeping=true;
        }
        if(!consSleeping){
            item = buffer[--count];
            if(count==N-1) prodSleeping = false;
            printf(&item);
            printf("\n");
        }
    }
}
```

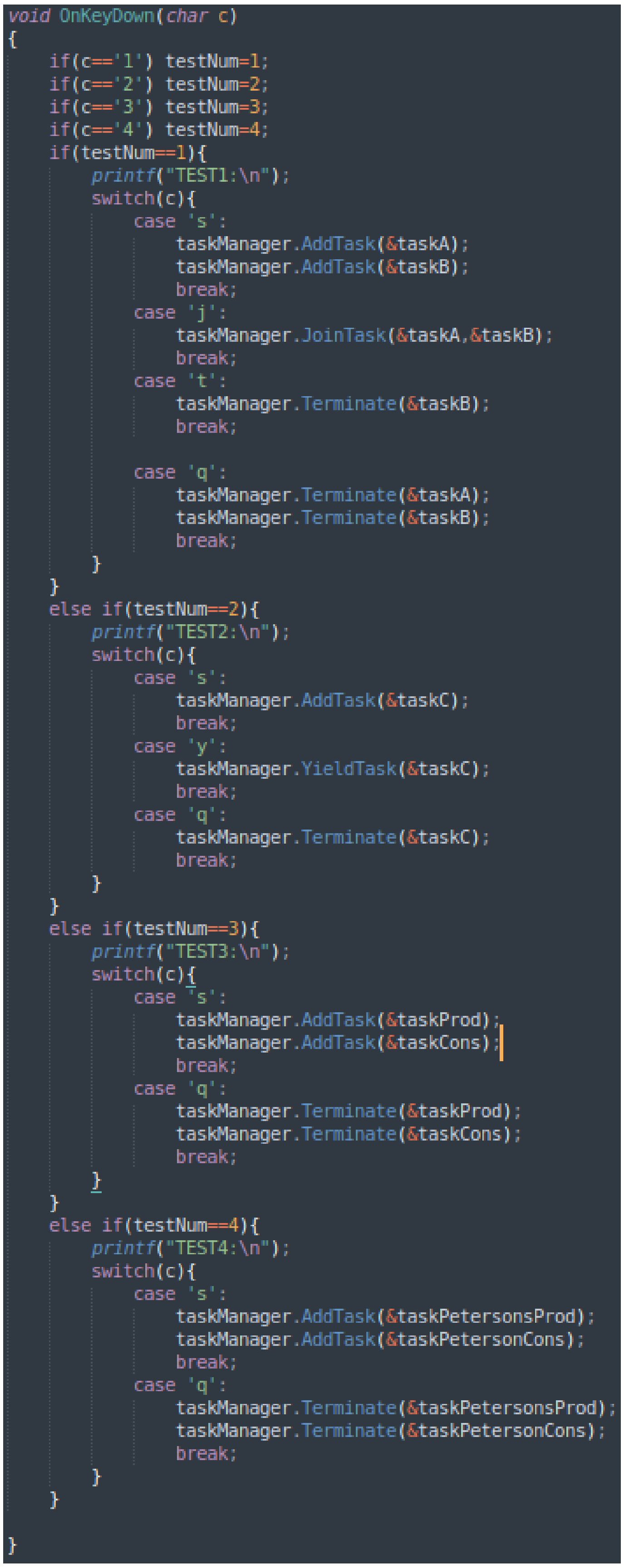


```c
//---for testing Creating,Removing,Joining threads---//
void funcA(void){
    while(true){
        printf("Thread A Running\n");
    }
}

void funcB(void){
    while(true){
        printf("Thread B Running\n");
    }
}
//--for testing Yield--//
void funcC(void){
    printf("Thread C started.\n");
    while(true);
}
```



```c
//---for testing consumer-producer without race condition (Peterson's Solution)---//

#define PAMT 2 //process amt;
int turn;
bool interested[PAMT];
void enter_region(int process){
    int other = 1-process;
    interested[process]=true;
    turn = process;
    while(turn==process && interested[other]==true)/*BUSY WAIT*/;
}
void leave_region(int process){
    interested[process]=false;
}

void petersonsProducer(){
    char item='A';
    while(true){
        if(count==N) prodSleeping = true;
        if(!prodSleeping){
            enter_region(0);
                buffer[count++] = item++;
                if(item-1>='Z') item = 'A';
                if(count==1) consSleeping = false;
            leave_region(0);

        }
    }
}

void petersonsConsumer(){
    char item;
    while(true){
        if(count==0) consSleeping=true;
        if(!consSleeping){
            enter_region(1);
                item = buffer[--count];
                if(count==N-1) prodSleeping = false;
                printf(&item);
                printf("\n");
            leave_region(1);
        }
    }
}
```

Test Cases



```
void OnKeyDown(char c)
{
    if(c=='1') testNum=1;
    if(c=='2') testNum=2;
    if(c=='3') testNum=3;
    if(c=='4') testNum=4;
    if(testNum==1){
        printf("TEST1:\n");
        switch(c){
            case 's':
                taskManager.AddTask(&taskA);
                taskManager.AddTask(&taskB);
                break;
            case 'j':
                taskManager.JoinTask(&taskA,&taskB);
                break;
            case 't':
                taskManager.Terminate(&taskB);
                break;

            case 'q':
                taskManager.Terminate(&taskA);
                taskManager.Terminate(&taskB);
                break;
        }
    }
    else if(testNum==2){
        printf("TEST2:\n");
        switch(c){
            case 's':
                taskManager.AddTask(&taskC);
                break;
            case 'y':
                taskManager.YieldTask(&taskC);
                break;
            case 'q':
                taskManager.Terminate(&taskC);
                break;
        }
    }
    else if(testNum==3){
        printf("TEST3:\n");
        switch(c){
            case 's':
                taskManager.AddTask(&taskProd);
                taskManager.AddTask(&taskCons);
                break;
            case 'q':
                taskManager.Terminate(&taskProd);
                taskManager.Terminate(&taskCons);
                break;
        }
    }
    else if(testNum==4){
        printf("TEST4:\n");
        switch(c){
            case 's':
                taskManager.AddTask(&taskPetersonsProd);
                taskManager.AddTask(&taskPetersonCons);
                break;
            case 'q':
                taskManager.Terminate(&taskPetersonsProd);
                taskManager.Terminate(&taskPetersonCons);
                break;
        }
    }

}
```

Test1
Adding, joining and termination are tested.

Test2
Yielding is tested.

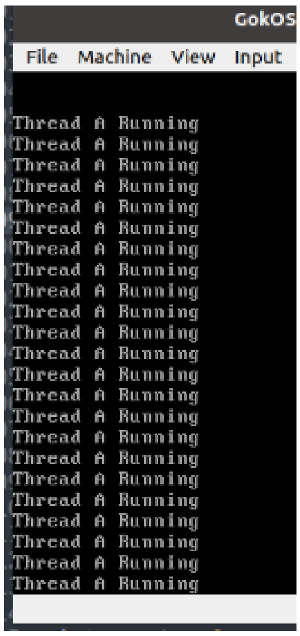
Test3
Producer-Consumer functions with Race Condition is Tested

Test4
Producer-Consumer functions with Peterson’s solution (no race condition) is tested.

Results

Test1 (Press 1 to start)

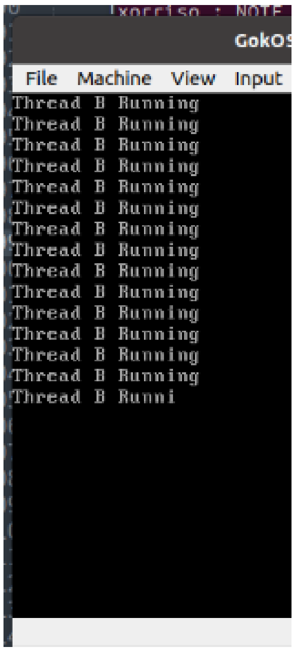
Step1: s is pressed (started taskA and TaskB)



```
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
```

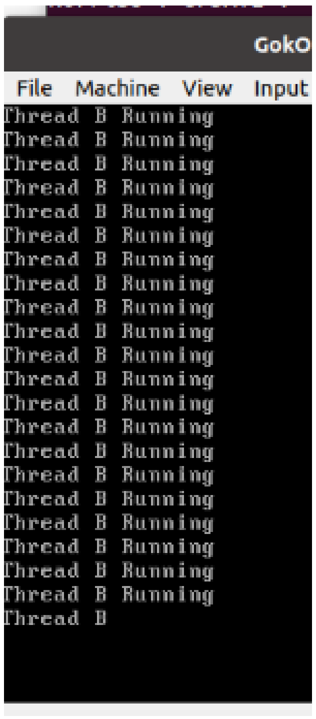
Changes every clock cycle

<--->



```
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Runni
```

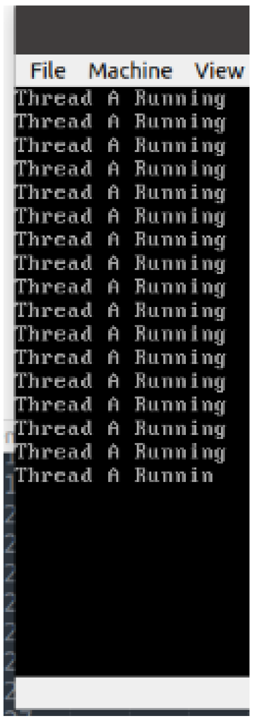
Step2: j is pressed (joined B to A)



```
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B Running
Thread B
```

Only B is running

Step4: t is pressed (terminated B)



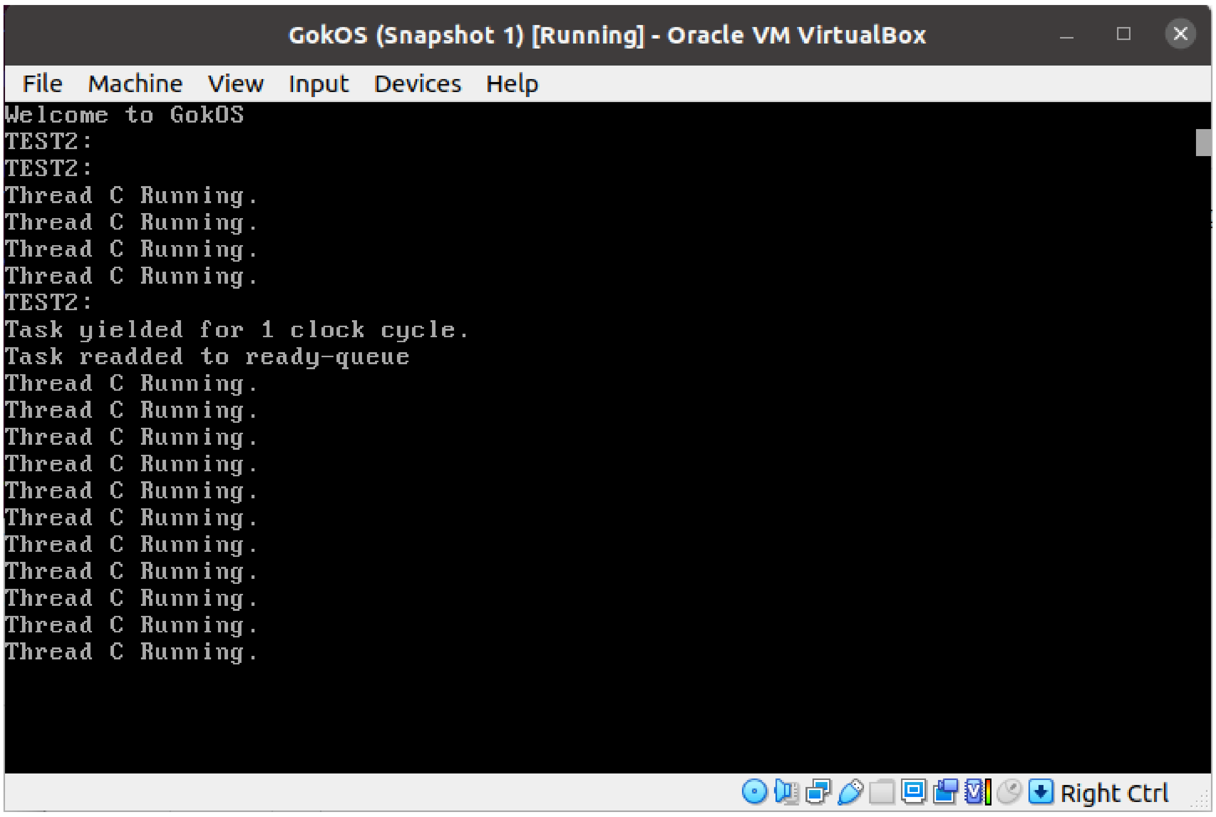
```
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Running
Thread A Runnin
```

Only A is running

When q is pressed, both threads are stopped.

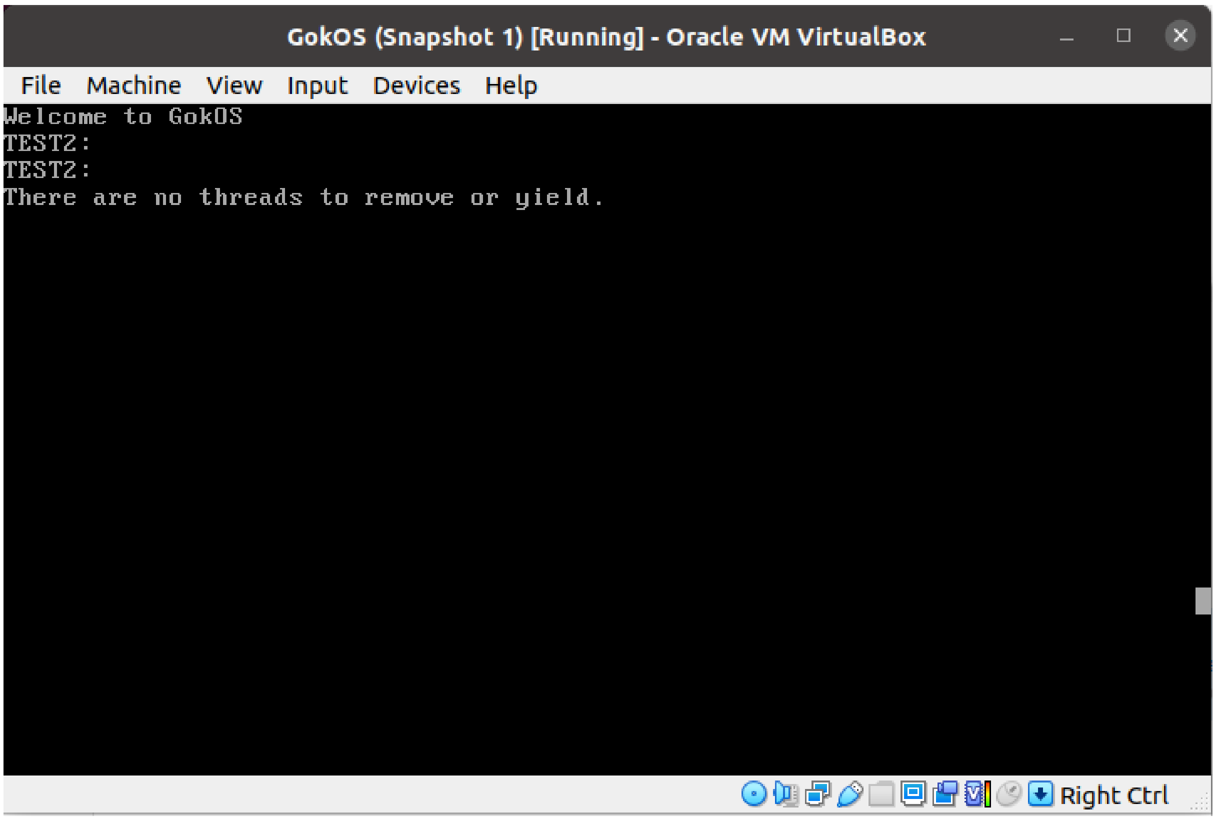
Test 2 (press 2 to start)

s and y are pressed in order:


GokOS (Snapshot 1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

```
Welcome to GokOS
TEST2:
TEST2:
Thread C Running.
Thread C Running.
Thread C Running.
Thread C Running.
TEST2:
Task yielded for 1 clock cycle.
Task readded to ready-queue
Thread C Running.
Thread C Running.
Thread C Running.
Thread C Running.
Thread C Running.
Thread C Running.
Thread C Running.
Thread C Running.
Thread C Running.
Thread C Running.
Thread C Running.
```

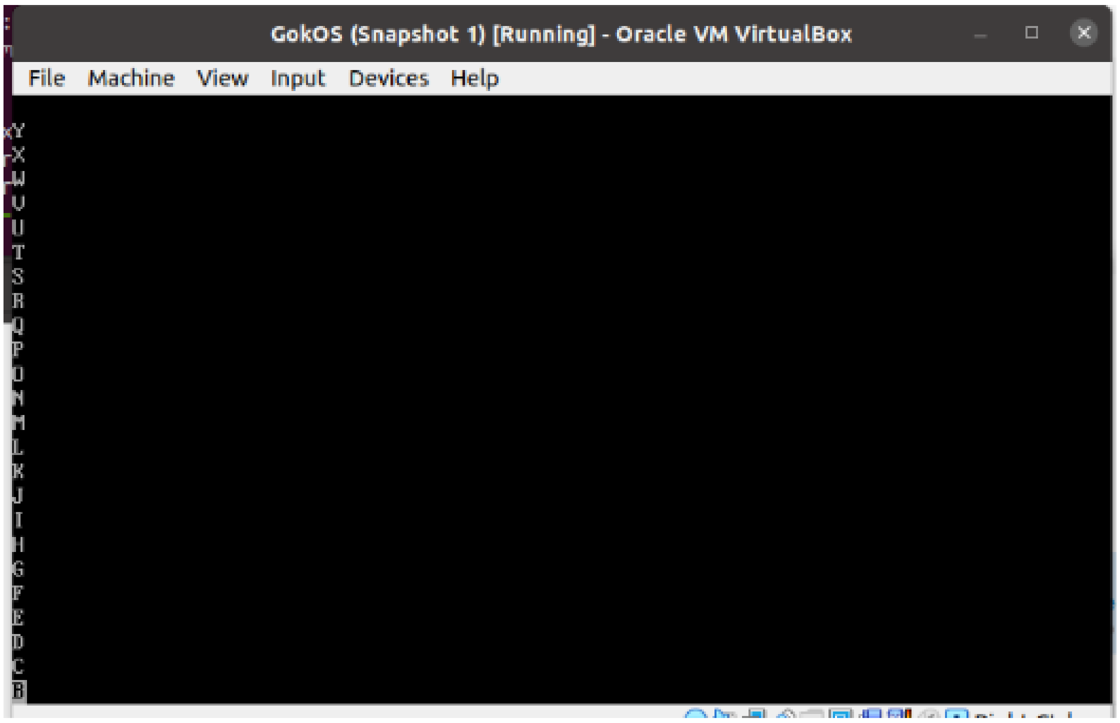
Only y is pressed:


GokOS (Snapshot 1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

```
Welcome to GokOS
TEST2:
TEST2:
There are no threads to remove or yield.
```

Test 3 (press 3 to start)

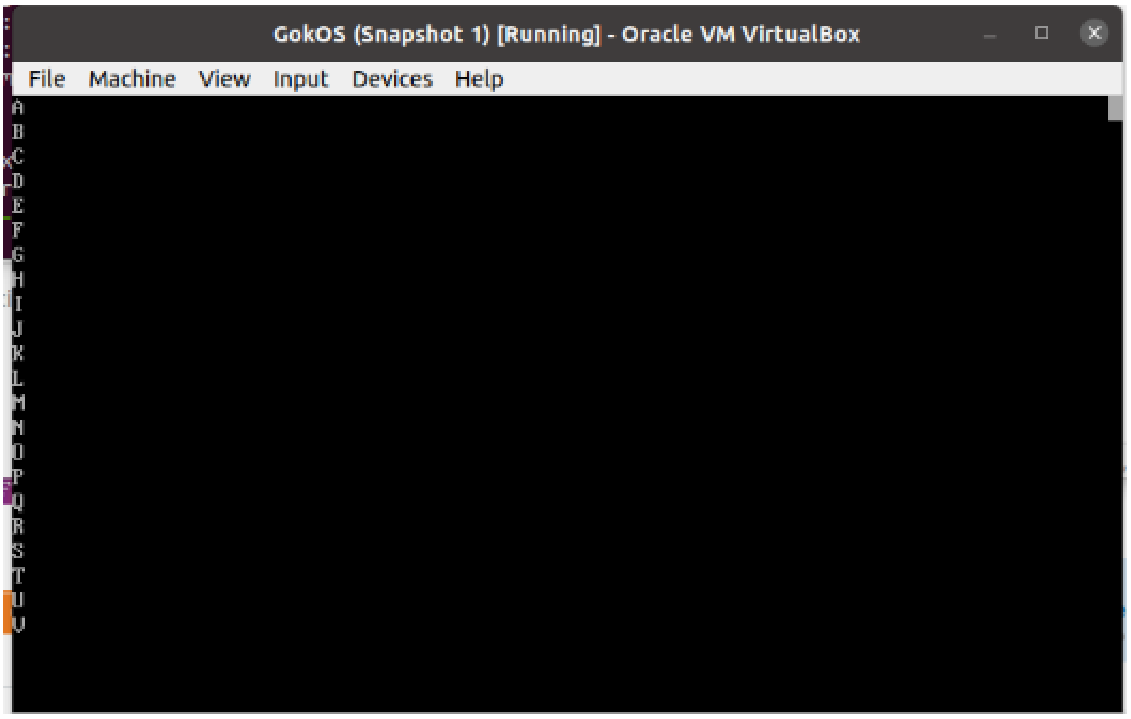
s is pressed:



```
Y
X
W
V
U
T
S
R
Q
P
O
N
M
L
K
J
I
H
G
F
E
D
C
B
```

Output is random since there is a Race Condition between the Producer and the Consumer.

Test 4 (press 4 to start)



```
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
```

Output is in alphabetical order since Race Condition is solved by Peterson's Solution.