CSE 312
Homework 1 REPORT
ERSEL CELAL EREN
1901042673

1 - How to Run?

There is only single folder for all parts. I added macros into kernel.cpp and multitasking.cpp files. You need to remove comment of related macros for testing different Parts. Later I will explain these comments.
So, with “make mykernel.iso” command, you can create .iso file of operating system. You should add it into VirtualBox with “My Own Os” name if you want to use “make run”.

2 – General Design

In the homework, we have 3 different part.

Part – A : In this part, it is asked us to implement multiprogramming with round robin and fork(), wait(), exec() syscalls. Since this is my second time of taking this course, I reviewed the shared projects from last year and attended P.S which is done by İlhan Aytutuldu. So used some ideas from them in some part of my implementation.
Since, the fork syscall of shared projects are not seemed working truly, I have also another fork() approache. After implementing fork(), I implemented waitpid() syscall. Waitpid syscall uses ‘pid’ field in process class. It is needed in scheduler for processes with BLOCKED state.

init() process adds all processes with fork. There is timer interrupt with the number of 256 to make it more slow. At every task switching content of process table is printed.

I have implemented a sleep() function by using clock cycles. It became useful in printing and testing.

There is a TaskManager class which is responsible from adding tasks, updating processes and scheduling. It has a process table. Instead of pointer array of tasks, I used array of tasks.

Round robin is used in default scheduling. When interrupt happens, it searches next process with READY state and starts running it.

Syscalls make interrupt 0x80 and passes registers to interrupt handler. This is done by inline assembly codes.

Part – B) In this part to provide randomizing, I used “rdtsc”. With the help of clock I generated random numbers.
In third strategy it is asked us to implement priority scheduling. Because of that, I added ‘priority’ field into each process. There is possible priority from 1 to 255. If priority is lower by number, that task is

running until it is terminated. (for example task with 90 priority will be running, task with 100 will be waiting).
For dynamic priority scheduling strategy, I added a ‘streak’ field for each process. At each timer interrupt, if the current process is not changed and still be executing, streak value of that process is incremented. In the PDF, it is given as 5^{th} interrupt as an example but since programs are not executed that long in my tests, I changed it as 2. So if a process 2 times in a row, its priority became the highest priority among all processes and next lowest priority process starts running.

PART – C) In this part, I implemented only first strategy. As I understand, at each mouse click scheduling will be activated. It is also asked us to select a random program but to observing tests better, I added 3 same process which runs in while loop. I thought that would not be a problem because in previous parts I have already implemented random process loading structure.
So, in this part programs are running with a default scheduling algorithm and if user clicks mouse left click, task switching happens.

Code Explanation :

```
class Task
{
friend class TaskManager;
private:
    common::uint8_t stack[4096];
    CPUState* cpustate;
    common::uint32_t pid;
public:
    Task();
    ~Task();
    Task(GlobalDescriptorTable *gdt, void entrypoint());
    Task(GlobalDescriptorTable *gdt);
    void setEntryPoint(void entrypoint());
    void* GetEntryPoint();
    void Set(Task* task);
};
```

```
class TaskManager
{
private:
    Task tasks[256];
    int numTasks;
    int currentTask;
    int terminatedTaskCount;
    ProcessTable processTable;
public:
    TaskManager(GlobalDescriptorTable *gdt);
    ~TaskManager();
    bool AddTask(Task* task);
    bool AddTaskWithPriority(Task* task, common::uint32_t priority);
    CPUState* Schedule(CPUState* cpustate);
    CPUState* OriginalSchedule(CPUState* cpustate);
    void PrintProcessTable();
    void Terminate(int* returnValue);
    common::uint32_t Fork_1(CPUState* cpustate);
    common::uint32_t Fork_2(CPUState* cpustate);
    void Exec(CPUState* cpustate);
    void Wait(CPUState* cpustate);
    common::uint32_t GetCurrentPid();

};
```

Constructor, pid, setter function is implemented.
Process and ProcessTable added to TaskManager.
I updated AddTask and Schedule functions from shared projects. There is also fork,exec, terminate functions.

Process States :

```
enum ProcessState {RUNNING,BLOCKED,READY,TERMINATED};
```

```
class Process {

friend class TaskManager;

private:
    common::uint32_t pid;
    common::uint32_t ppid;
    common::uint32_t waitPid;
    common::uint32_t children[256];
    int* retval;
    ProcessState state;
    common::uint32_t numChildren;
```

TaskManager uses this class. So it is friend.

```
struct ProcessTable
{
    Process processes[256];
    int numProcesses;
};
```

Inactive process has pid with number 0.

```
public:
    Process(common::uint32_t pid, common::uint32_t ppid);
    Process();
    ~Process();
    bool IsChild(common::uint32_t pid);
    void AddChild(common::uint32_t pid);
    void RemoveChild(common::uint32_t pid);
    common::uint32_t GetPID();
    common::uint32_t GetPPID();
    int* GetReturnValue();
    ProcessState GetState();
    common::uint32_t GetNumChildren();
    void SetPID(common::uint32_t pid);
    void SetPPID(common::uint32_t ppid);
    void SetReturnValue(int* retval);
    void SetState(ProcessState state);
    void SetNumChildren(common::uint32_t numChildren);
    common::uint32_t priority;
    common::uint32_t streak;
};
```

AddTask implementation : It checks are there more than 256 task. Since task limit is 256. Then it finds mininum possible process ID. It gets process instance from table and sets its fields. New created process is set as Ready and its parent is ‘init’. If not init, add to parent’s child list.

```
bool TaskManager::AddTask(Task* task)
{
    if(numTasks >= 256)
        return false;

    for (int i = 1; i < 256; i++)
    {
        if (processTable.processes[i].GetPID() == 0)
        {
            task->pid = i;
            break;
        }
    }

    Process* process = &(processTable.processes[task->pid]);

    process->SetPID(task->pid);
    process->SetPPID(1);
    process->SetState(READY);
    process->SetNumChildren(0);

    if (task->pid != 1)
    {
        Process* parent = &(processTable.processes[process->GetPPID()]);
        parent->AddChild(task->pid);
    }

    tasks[numTasks++].Set(task);
    return true;
}
```

It checks currentTask is init or not. Init cannot be terminated. If not init, set its state as Terminated.

```
void TaskManager::Terminate(int* returnValue)
{
    if (tasks[currentTask].pid == 1)
        return;

    Process* process = &(processTable.processes[tasks[currentTask].pid]);
    process->SetReturnValue(returnValue);

    process->SetState(TERMINATED);
    terminatedTaskCount++;

    while(true);
}
```

As a difference from AddTask function, it takes priority as parameter and set it to process.

```
bool TaskManager::AddTaskWithPriority(Task* task, common::uint32_t priority)
{
    if(numTasks >= 256)
        return false;

    for (int i = 1; i < 256; i++)
    {
        if (processTable.processes[i].GetPID() == 0)
        {
            task->pid = i;
            break;
        }
    }

    Process* addedProcess = &(processTable.processes[task->pid]);

    addedProcess->SetPID(task->pid);
    addedProcess->SetPPID(1);
    addedProcess->SetState(READY);
    addedProcess->SetNumChildren(0);
    addedProcess->priority = priority;
    addedProcess->streak = 0;

    if (task->pid != 1)
    {
        Process* parent = &(processTable.processes[addedProcess->GetPPID()]);
        parent->AddChild(task->pid);
    }

    tasks[numTasks++].Set(task);
    return true;
}
```

DEFAULT SCHEDULE :

```
#ifdef DEFAULT_SCHEDULE
CPUState* TaskManager::Schedule(CPUState* cpustate)
{

    if(numTasks - terminatedTaskCount <= 0)
        return cpustate;

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

    uint32_t from = -1;

    if (numTasks - terminatedTaskCount > 1 && currentTask >= 0)
    {
        from = tasks[currentTask].pid;
    }

    if (tasks[currentTask].pid != 0)
    {
        Process* process = &(processTable.processes[tasks[currentTask].pid]);
        if (process->GetState() == RUNNING)
            process->SetState(READY);
    }

    ProcessState state = TERMINATED;
    do
    {
        if(++currentTask >= numTasks){
            currentTask =currentTask % numTasks;
        }

        if (processTable.processes[tasks[currentTask].pid].GetState() == BLOCKED)
        {
            Process* process = &(processTable.processes[tasks[currentTask].pid]);
            if (processTable.processes[process->waitPid].GetState() == TERMINATED)
            {
                process->SetState(READY);
                process->waitPid = 0;
            }
        }

        state = processTable.processes[tasks[currentTask].pid].GetState();
    } while (state == TERMINATED || state == BLOCKED);
```

```
    uint32_t to = -1;

    if (numTasks - terminatedTaskCount > 1 && currentTask >= 0)
    {
        to = tasks[currentTask].pid;
    }

    if(from != -1 && to != -1){
        printf("Switching task from ");
        printfDec(from);
        printf(" to ");
        printfDec(to);
        printf("\n");
    }


    // current task is on the running state
    Process* process = &(processTable.processes[tasks[currentTask].pid]);
    if(process->GetState() == READY)
        process->SetState(RUNNING);

    PrintProcessTable();

    return tasks[currentTask].cpustate;
}
#endif
```

This is default Schedule function. It doesn't use priority of processes. It is simply switchs current process READY from RUNNING and starts search not TERMINATED and BLOCKED processes.When it finds READY process, it sets state of it and returns CpuState of it. It also saves PID of previous and next scheduled processes IDs for printing.

PRIORITY SCHEDULE :

```
#ifdef THIRD_STRATEGY_PRIORITY_SCHEDULE
CPUState* TaskManager::Schedule(CPUState* cpustate)
{
    if (numTasks - terminatedTaskCount <= 0)
        return cpustate;

    if (currentTask >= 0)
        tasks[currentTask].cpustate = cpustate;

    bool printFlag = false;

    #ifdef PRINT_FLAG
    if (numTasks - terminatedTaskCount > 1 && currentTask >= 0)
    {
        printf("Switching task from ");
        printfDec(tasks[currentTask].pid);
        printFlag = true;
    }
    #endif

    // Current task is in the ready state
    if (tasks[currentTask].pid != 0)
    {
        Process* process = &(processTable.processes[tasks[currentTask].pid]);
        if (process->GetState() == RUNNING)
            process->SetState(READY);
    }

    ProcessState state = TERMINATED;
    int lowestPriorityTask = -1;
    int lowestPriority = 255;
```

```
    for (int i = 0; i < numTasks; ++i)
    {
        if (processTable.processes[tasks[i].pid].GetState() != TERMINATED &&
            processTable.processes[tasks[i].pid].priority < lowestPriority)
        {
            lowestPriorityTask = i;
            lowestPriority = processTable.processes[tasks[i].pid].priority;
        }
    }

    currentTask = lowestPriorityTask;

    #ifdef PRINT_FLAG
    if (numTasks - terminatedTaskCount > 1 && currentTask >= 0 && printFlag)
    {
        printf(" to ");
        printfDec(tasks[currentTask].pid);
        printf("\n");
        printFlag = false;
    }
    #endif

    // Current task is in the running state
    Process* process = &(processTable.processes[tasks[currentTask].pid]);
    if (process->GetState() == READY)
        process->SetState(RUNNING);


    PrintProcessTable();

    return tasks[currentTask].cpustate;
}
#endif
```

It is very similar to default schedule but as a difference, it finds a process with lowest priority. Process with a small priority number is chosen here. Also running process is not switched until it is terminated in this strategy.

```
#ifdef DYNAMIC_PRIORITY_SCHEDULE
CPUState* TaskManager::Schedule(CPUState* cpustate)
{

    if (numTasks - terminatedTaskCount <= 0)
        return cpustate;

    if (currentTask >= 0)
        tasks[currentTask].cpustate = cpustate;

    uint32_t from = -1;

    if (numTasks - terminatedTaskCount > 1 && currentTask >= 0)
    {
        from = tasks[currentTask].pid;
    }

    if (tasks[currentTask].pid != 0)
    {
        Process* process = &(processTable.processes[tasks[currentTask].pid]);
        if (process->GetState() == RUNNING)
            process->SetState(READY);
    }

    ProcessState state = TERMINATED;
    int lowestPriorityTask = -1;
    int lowestPriority = 255;
    int tempCurrentTaskIndex = currentTask;

    for (int i = 0; i < numTasks; ++i)
    {
        if (processTable.processes[tasks[i].pid].GetState() != TERMINATED &&
            processTable.processes[tasks[i].pid].priority < lowestPriority
            && processTable.processes[tasks[i].pid].streak < 2)
        {
            lowestPriorityTask = i;
            lowestPriority = processTable.processes[tasks[i].pid].priority;
        }

    }
```

```
    if(processTable.processes[tasks[currentTask].pid].streak >= 2){
        processTable.processes[tasks[currentTask].pid].streak = 0;
        int highestPriorityTask = -1;
        int highestPriority = 0;
        for (int i = 1; i < numTasks; ++i)
        {
            if (processTable.processes[tasks[i].pid].GetState() != TERMINATED &&
                processTable.processes[tasks[i].pid].priority > highestPriority)
            {
                highestPriorityTask = i;
                highestPriority = processTable.processes[tasks[i].pid].priority;
            }

        }

        if(highestPriority > 254){
            processTable.processes[tasks[currentTask].pid].priority = 254;
        }
        else{
            processTable.processes[tasks[currentTask].pid].priority = highestPriority + 1;
        }
    }


    if(lowestPriorityTask == currentTask){
        printf("=====Current task is the lowest priority task=====\n");
        //increment streak of currenttask
        processTable.processes[tasks[currentTask].pid].streak++;
    }
    else{
        //reset streak of current task
        processTable.processes[tasks[tempCurrentTaskIndex].pid].streak = 0;
    }

    // Set the next task based on priority
    currentTask = lowestPriorityTask;

    uint32_t to = -1;

    if (numTasks - terminatedTaskCount > 1 && currentTask >= 0)
    {
        to = tasks[currentTask].pid;
    }
```

It is again another modified version of previous ones. I added ‘streak’ field into process. When a process Scheduled again, streak counter is incremented. When its streak becomes 2 (in PDF it is 5 but my timer is slower, so programs may not be run 5 times in a row.). When a process exceed 2 streak, its priority becomes largest priority in process list. Also its streak is resetted. In this function there are two loop: first loop finds lowest priority process. Second loop is executed if chosen process is exceed streak limit.

```
    if(from != -1 && to != -1){
        printf("Switching task from ");
        printfDec(from);
        printf(" to ");
        printfDec(to);
        printf("\n");
    }

    // Current task is in the running state
    Process* process = &(processTable.processes[tasks[currentTask].pid]);
    if (process->GetState() == READY)
        process->SetState(RUNNING);

    PrintProcessTable();

    return tasks[currentTask].cpustate;
}
#endif
```

Here there are other simple functions like getters and setters.

```
> void Process::AddChild(uint32_t pid) ···
> void Process::RemoveChild(uint32_t pid) ···
> bool Process::IsChild(uint32_t pid) ···
> common::uint32_t Process::GetPID() {···
> common::uint32_t Process::GetPPID() {···
> common::uint32_t Process::GetNumChildren() {···
> ProcessState Process::GetState() {···
> int* Process::GetReturnValue() {···
```

```
> void Process::SetPID(common::uint32_t pid) {···
> void Process::SetPPID(common::uint32_t ppid) {···
> void Process::SetNumChildren(common::uint32_t numChildren) {···
> void Process::SetState(ProcessState state) {···
> void Process::SetReturnValue(int* retval) {···
```

Syscalls :

```
void fork(int *pid)
{
    asm("int $0x80" : "=c" (*pid) : "a" (1));
}
```

```
int forkWithExec(void entrypoint())
{
    int ret;
    asm("int $0x80" : "=a" (ret) : "a" (5), "b" ((uint32_t)entrypoint));
    return ret;
}
```

Here inline assembly syscall functions. Parameters passed and received through eax, ebx, ecx registers.

```
void sysprintf(char* str)
{
    asm("int $0x80" : : "a" (4), "b" (str));
}

void sysexecve(void entry())
{
    asm("int $0x80" : : "a" (3), "b" (entry));
}
```

```
void sys_waitpid(uint32_t pid)
{
    printf("sys_waitpid() called for pid: ");
    printfDec(pid);
    asm("int $0x80" : : "a" (2), "b" (pid));
}
```

==========

Syscall Handler : Here the right side, HandleInterrupt is implemented.

```
enum SYSCALLS
{
    SYS_FORK = 1,
    SYS_WAIT = 2,
    SYS_EXECVE = 3,
    SYS_PRINT = 4,
    SYS_FORK_EXEC = 5
};
```

```
uint32_t SyscallHandler::HandleInterrupt(uint32_t esp)
{
    CPUState* cpu = (CPUState*)esp;

    if (cpu->eax == SYS_FORK) {
        cpu->ecx = fork(cpu);
        return InterruptHandler::HandleInterrupt((uint32_t) cpu);
    }
    else if (cpu->eax == SYS_WAIT) {
        wait(cpu);
        return InterruptHandler::HandleInterrupt((uint32_t) cpu);
    }
    else if (cpu->eax == SYS_EXECVE) {
        execve(cpu);
        cpu->eax = 0;
        cpu->eip = cpu->ebx;
        esp = (uint32_t) cpu;
    }
    else if (cpu->eax == SYS_PRINT) {
        printf((char*)cpu->ebx);
    }
    else if (cpu->eax == SYS_FORK_EXEC) {
        cpu->eax = fork2(cpu);
        return InterruptHandler::HandleInterrupt((uint32_t) cpu);
    }
    else {
        // default case, nothing to do
    }

    return esp;
}
```

kernel.cpp :

```
extern "C" void kernelMain(const void* multiboot_structure, uint32_t /*multiboot_magic*/)
{
    printf(">>>>>>> Welcome <<<<<<<\n");

    init();
```

init() is called in kernel.main.

In init, InitProcess is added into TaskManager. For Part B, first #ifdef will be used.

```
void init() {
    Task task_init(&gdt, &InitProcess);

    #ifdef PRIORITY_FLAG
        taskManager.AddTaskWithPriority(&task_init,254);
    #endif

    #ifdef NON_PRIORITY_FLAG
        taskManager.AddTask(&task_init);
    #endif

}
```

Here InitProcess. It calls specific functions inside. I divided the tests with macros.

```
void InitProcess() {
    printf("Init process started\n");

    #ifdef PART_A ···
    #endif

    #ifdef PART_B_First_Strategy ···
    #endif

    #ifdef PART_B_Second_Strategy ···
    #endif

    #ifdef PART_B_Third_Strategy ···
    #endif

    #ifdef PART_B_DYNAMIC_STRATEGY ···
    #endif

    #ifdef PART_C_First_Strategy ···
    #endif
```

```
    #ifdef FORK_TEST_1 ···
    #endif

    #ifdef FORK_TEST_2 ···
    #endif

    #ifdef GETPID_TEST ···
    #endif

    #ifdef WAITPID_TEST ···
    #endif

    #ifdef EXECVE_TEST ···
    #endif

    while (true);
}
```

For Part-A, I am creating 3 long_running and 3 my_own_collatz program.

```
#ifdef PART_A
    void (*process[])() = {long_running, my_own_collatz};

    for (int i = 0 ; i < 3; ++i) {
        forkWithExec(process[0]);
        forkWithExec(process[1]);
    }
#endif
```

For Part-B First Strategy, I used randomNumberGenerator function that I implemented
Then program chose one program and loads it 10 times with fork.

```
#ifdef PART_B_First_Strategy

    int randomNumber = randomNumberGenerator(4);
    printf("Random number: ");
    printfDec(randomNumber);
    printf("\n");
    custom_sleep(1.0);
    if(randomNumber == 0){
        printf("Collatz is chosen\n");
        custom_sleep(3.0);
        for(int i=0;i<10;i++){
            forkWithExec(&my_own_collatz);
        }
    }
    else if(randomNumber == 1){
        printf("Binary Search is chosen\n");
        custom_sleep(3.0);
        for(int i=0;i<10;i++)
            forkWithExec(&my_own_binarySearch);
    }
    else if(randomNumber == 2){
        printf("Linear Search is chosen\n");
        custom_sleep(3.0);
        for(int i=0;i<10;i++)
            forkWithExec(&my_own_linearSearch);
    }
    else if(randomNumber == 3){
        printf("Long Running is chosen\n");
        custom_sleep(3.0);
        for(int i=0;i<10;i++)
            forkWithExec(&long_running);
    }
#endif
```

In Part B Second Strategy, I generated two different random number and chose 2 different program and load it 3 times with fork.

```
#ifdef PART_B_Second_Strategy
    int processCount = 4;
    int randomIndex1 = randomNumberGenerator(processCount);
    int randomIndex2;
    uint32_t pid1 = 0;
    uint32_t pid2 = 0;

    do {
    randomIndex2 = randomNumberGenerator(processCount);
    } while (randomIndex2 == randomIndex1);

    randomIndex1 = 0;
    randomIndex2 = 3;

    //write switch case
    if(randomIndex1 == 0)printf("CollbinarySearchHelperatz is chosen\n");
    else if(randomIndex1 == 1)printf("Binary Search is chosen\n");
    else if(randomIndex1 == 2)printf("Linear Search is chosen\n");
    else if(randomIndex1 == 3)printf("Long Running is chosen\n");

    if(randomIndex2 == 0)printf("Collatz is chosen\n");
    else if(randomIndex2 == 1)printf("Binary Search is chosen\n");
    else if(randomIndex2 == 2)printf("Linear Search is chosen\n");
    else if(randomIndex2 == 3)printf("Long Running is chosen\n");

    void (*process[])() = {my_own_collatz, my_own_binarySearch, my_own_linearSearch,long_running};

    for (int i = 0; i < 3; ++i) {
        pid1 = forkWithExec(process[randomIndex1]);
        printf("PID1: ");
        printfDec(pid1);
        printf("\n");
        pid2 = forkWithExec(process[randomIndex2]);
        printf("PID2: ");
        printfDec(pid2);
        printf("\n");

    }
    // TODO check these sys_waitpids
    sys_waitpid(pid1);
    sys_waitpid(pid2);

#endif
```

In this strategy, I am adding 4 different programs with 4 different priority. ‘task_third_strategy1’ will be run first.

```
#ifdef PART_B_Third_Strategy
    printf("Third Strategy\n");
    custom_sleep(3.0);
    Task task_third_strategy1(&gdt, &my_own_collatz);
    Task task_third_strategy2(&gdt, &my_own_binarySearch);
    Task task_third_strategy3(&gdt, &my_own_linearSearch);
    Task task_third_strategy4(&gdt, &long_running);

    taskManager.AddTaskWithPriority(&task_third_strategy1, 10);
    taskManager.AddTaskWithPriority(&task_third_strategy2, 40);
    taskManager.AddTaskWithPriority(&task_third_strategy3, 30);
    taskManager.AddTaskWithPriority(&task_third_strategy4, 20);
#endif
```

Again for Dynamic Priority strategy, 3 collatz program is loaded with different priorities.

```
#ifdef PART_B_DYNAMIC_STRATEGY
    Task task_priority1(&gdt, &my_own_collatz);
    Task task_priority2(&gdt, &my_own_collatz);
    Task task_priority3(&gdt, &my_own_collatz);
    taskManager.AddTaskWithPriority(&task_priority1, 100);
    taskManager.AddTaskWithPriority(&task_priority2, 110);
    taskManager.AddTaskWithPriority(&task_priority3, 90);
#endif
```

Here other test cases for other implementations.

```
#ifdef PART_C_First_Strategy
    for(int i=0; i<3; i++){
        forkWithExec(&taskE);
    }
#endif

#ifdef FORK_TEST_1
    taskFork();
#endif

#ifdef FORK_TEST_2
    forkWithExec(&taskFork);
#endif
```

```
#ifdef GETPID_TEST
    my_own_testGetPID();
#endif

#ifdef WAITPID_TEST
    int pid = forkWithExec(&taskWait);
    sys_waitpid(pid);
#endif

#ifdef EXECVE_TEST
    forkWithExec(&taskExecve);
#endif
```

At each scheduling, this print function is used to print process table.

```
void TaskManager::PrintProcessTable()
{
    printf("\nProcess table:\n");
    for (int i = 0; i < 256; i++)
    {
        if (processTable.processes[i].GetPID() != 0)
        {
            printf("Priority: ");
            printfDec(processTable.processes[i].priority);
            printf(" |Streak: ");
            printfDec(processTable.processes[i].streak);
            printf(" |PID: ");
            printfDec(processTable.processes[i].GetPID());
            printf(" |PPID: ");
            printfDec(processTable.processes[i].GetPPID());
            printf(" |State: ");

            if (processTable.processes[i].GetState() == READY) {
                printf("READY");
            } else if (processTable.processes[i].GetState() == RUNNING) {
                printf("RUNNING");
            } else if (processTable.processes[i].GetState() == TERMINATED) {
                printf("TERMINATED");
            } else if (processTable.processes[i].GetState() == BLOCKED) {
                printf("BLOCKED");
            }

            printf(" RetVal: ");
            int* val = processTable.processes[i].GetReturnValue();
            if (*val < 0)
            {
                printf("-");
                printfDec(*val * -1);
            }
            else
                printfDec(*val);
            printf(" #Child: ");
            printfDec(processTable.processes[i].GetNumChildren());
            printf("\n");
        }
    }
    printf("\n");
}
```

Added printing for timer interrupt. Also slowed “UNHANDLED INTERRUPT” prints by using interruptCounter.

```
uint32_t InterruptManager::DoHandleInterrupt(uint8_t interrupt, uint32_t esp)
{
    interruptCounter++;

    if(handlers[interrupt] != 0)
    {
        esp = handlers[interrupt]->HandleInterrupt(esp);
    }
    else if(interrupt != hardwareInterruptOffset && interruptCounter%10000000==0)
    {
        printf("UNHANDLED INTERRUPT 0x");
        printfHex(interrupt);
    }

    if(interrupt == hardwareInterruptOffset)
    {
        interruptCounter++;
        if(interruptCounter >= MAX_INTERRUPT_NUMBER && !readIOFlag && !criticalFlag)
        {
            printf("\n-------------------------------------------------");
            printf("\n-----> Timer interrupt occurred! Switching<-----");
            printf("\n-------------------------------------------------\n");
            interruptCounter = 0;
            esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
        }
    }

    if(hardwareInterruptOffset <= interrupt && interrupt < hardwareInterruptOffset+16)
    {
        programmableInterruptControllerMasterCommandPort.Write(0x20);
        if(hardwareInterruptOffset + 8 <= interrupt)
            programmableInterruptControllerSlaveCommandPort.Write(0x20);
    }

    return esp;
}
```

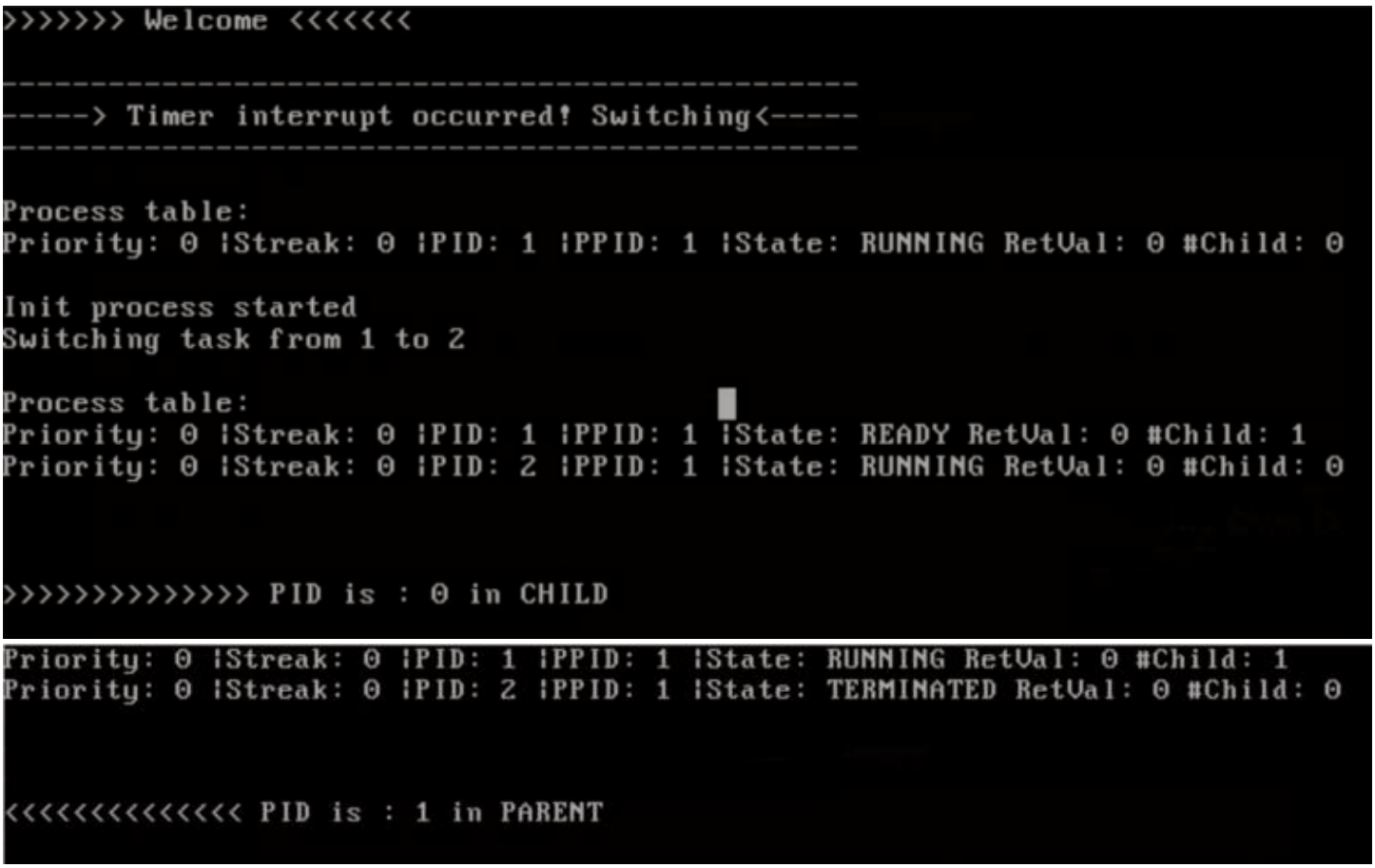
I used the below part for left mouse click.

```
if (buffer[0] & 0x1)
{
    if (!readIOFlag)
        printf("\n------> Left Mouse Click Detected - Switching Task <-----\n");
    clickFlag = true;
}
```

```
if (clickFlag && !readIOFlag)
    esp = (uint32_t) interruptManager->ScheduleTransmitter((CPUState*)esp);
```

TEST CASES : (Printings may be disordered or shifted because of screen limitations)

FORK_1 :



FORK_2 :

EXEC :

```
>>>>>>> Welcome <<<<<<<

------------------------------------------------
-----> Timer interrupt occurred! Switching<-----
------------------------------------------------

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0

Init process started
Switching task from 1 to 2

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: READY RetVal: 0 #Child: 1
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0

- Before Exec -
EXEC TEST FUNCTION !
```

GET_PID :

```
>>>>>>> Welcome <<<<<<<

------------------------------------------------
-----> Timer interrupt occurred! Switching<-----
------------------------------------------------

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0

Init process started
GET PID TEST
Switching task from 1 to 2

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: READY RetVal: 0 #Child: 1
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0

PID of Task-A: 00000002
|0 | Task-A |
|1 | Task-A |
|2 | Task-A |
|3 | Task-A |
|4 | Task-A |
```

```
------------------------------------------------
-----> Timer interrupt occurred! Switching<-----
------------------------------------------------
Switching task from 2 to 3

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: READY RetVal: 0 #Child: 2
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 3 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0

PID of Task-B : 00000003
|0 | Task-B |
|1 | Task-B |
|2 | Task-B |
|3 | Task-B |
|4 | Task-B |
```

WAITPID :

```
Priority: 0 |Streak: 0 |PID: 4 |PPID: 3 |State: TERMINATED RetVal: 0 #Child: 0

sys_waitpid() called for pid: 3Switching task from 2 to 3

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: BLOCKED RetVal: 0 #Child: 1
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: BLOCKED RetVal: 0 #Child: 1
Priority: 0 |Streak: 0 |PID: 3 |PPID: 2 |State: RUNNING RetVal: 0 #Child: 1
Priority: 0 |Streak: 0 |PID: 4 |PPID: 3 |State: TERMINATED RetVal: 0 #Child: 0

sys_waitpid() called for pid: 4Switching task from 3 to 3

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: BLOCKED RetVal: 0 #Child: 1
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: BLOCKED RetVal: 0 #Child: 1
Priority: 0 |Streak: 0 |PID: 3 |PPID: 2 |State: RUNNING RetVal: 0 #Child: 1
Priority: 0 |Streak: 0 |PID: 4 |PPID: 3 |State: TERMINATED RetVal: 0 #Child: 0
```

PART_A :

```
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: READY RetVal: 0 #Child: 6
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 3 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 4 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 5 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 6 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 7 |PPID: 1 |State: READY RetVal: 0 #Child: 0

 1405194336
-----------------------------------------------
-----> Timer interrupt occurred! Switching<-----
-----------------------------------------------
Switching task from 2 to 3

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: READY RetVal: 0 #Child: 6
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 3 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 4 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 5 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 6 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 7 |PPID: 1 |State: READY RetVal: 0 #Child: 0

| Cltz(63) = 593
```

PART_B_FIRST_STRATEGY :

```
>>>>>>> Welcome <<<<<<<

-----------------------------------------------
-----> Timer interrupt occurred! Switching<-----
-----------------------------------------------

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0

Init process started
Random number: 01
Collatz is chosen
Switching task from 1 to 2

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: READY RetVal: 0 #Child: 1
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0

CALCULATING COLLATZ > 83 : | Cltz(83) = 250 | Cltz(83) = 125
```

```
Switching task from 5 to 1

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 4
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 3 |PPID: 1 |State: TERMINATED RetVal: 3 #Child: 0
Priority: 0 |Streak: 0 |PID: 4 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 5 |PPID: 1 |State: READY RetVal: 0 #Child: 0

Switching task from 1 to 2

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: READY RetVal: 0 #Child: 5
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 3 |PPID: 1 |State: TERMINATED RetVal: 3 #Child: 0
Priority: 0 |Streak: 0 |PID: 4 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 5 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 6 |PPID: 1 |State: READY RetVal: 0 #Child: 0

| Cltz(83) = 700 | Cltz(83) = 350 | Cltz(83) = 175 | Cltz(83) = 526 | Cltz(83) =
 263
```

```
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 3 |PPID: 1 |State: TERMINATED RetVal: 3 #Child: 0
Priority: 0 |Streak: 0 |PID: 4 |PPID: 1 |State: TERMINATED RetVal: 4 #Child: 0
Priority: 0 |Streak: 0 |PID: 5 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 6 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 7 |PPID: 1 |State: READY RetVal: 0 #Child: 0

| Cltz(99) = 40 | Cltz(99) = 20 | Cltz(99) = 10 | Cltz(99) = 5 | Cltz(99) = 16 |
 Cltz(99) = 8 | Cltz(99) = 4 | Cltz(99) = 2
-----------------------------------------------
-----> Timer interrupt occurred! Switching<-----
-----------------------------------------------
Switching task from 5 to 6

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: READY RetVal: 0 #Child: 6
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 3 |PPID: 1 |State: TERMINATED RetVal: 3 #Child: 0
Priority: 0 |Streak: 0 |PID: 4 |PPID: 1 |State: TERMINATED RetVal: 4 #Child: 0
Priority: 0 |Streak: 0 |PID: 5 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 6 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 7 |PPID: 1 |State: READY RetVal: 0 #Child: 0

| Cltz(94) = 242 | Cltz(94) = 121 | Cltz(94) = 364 | Cltz(94) = 182 | Cltz(94) =
 91 | Cltz(94) = 274
```

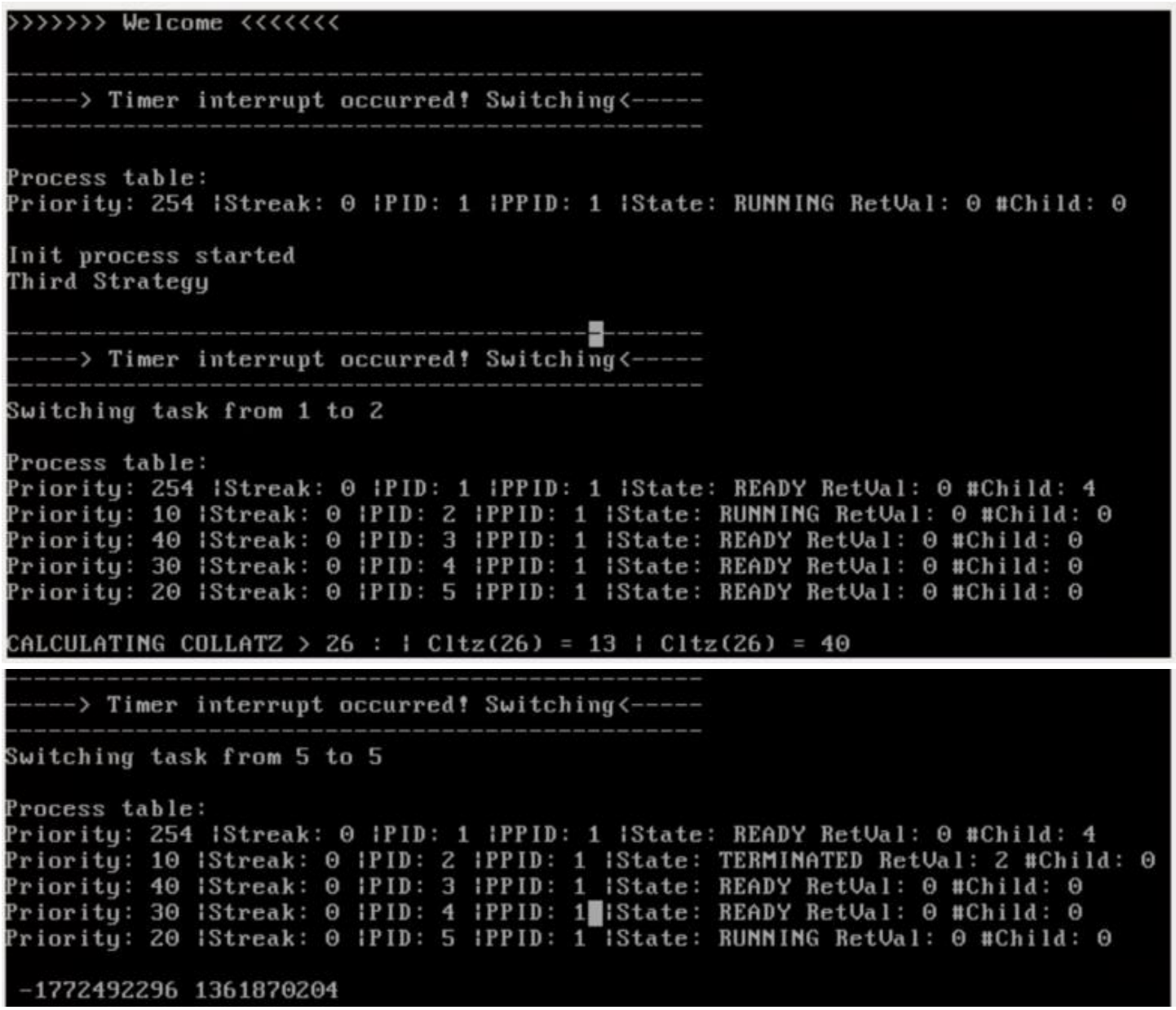
```
>>>>>>> Welcome <<<<<<<

------------------------------------------------
-----> Timer interrupt occurred! Switching<-----
------------------------------------------------

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0

Init process started
CollbinarySearchHelperatz is chosen
Long Running is chosen
Switching task from 1 to 2

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: READY RetVal: 0 #Child: 1
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0

CALCULATING COLLATZ > 45 : | Cltz(45) = 136 | Cltz(45) = 68 | Cltz(45) = 34 | Cl
tz(45) = 17 | Cltz(45) = 52 | Cltz(45) = 26 | Cltz(45) = 13
```

```
Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: READY RetVal: 0 #Child: 3
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: TERMINATED RetVal: 2 #Child: 0
Priority: 0 |Streak: 0 |PID: 3 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 4 |PPID: 1 |State: READY RetVal: 0 #Child: 0

 -1772492296 1361870204 1450015408
------------------------------------------------
-----> Timer interrupt occurred! Switching<-----
------------------------------------------------
Switching task from 3 to 4

Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: READY RetVal: 0 #Child: 3
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: TERMINATED RetVal: 2 #Child: 0
Priority: 0 |Streak: 0 |PID: 3 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 4 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0

CALCULATING COLLATZ > 63 : | Cltz(63) = 190 | Cltz(63) = 95
```

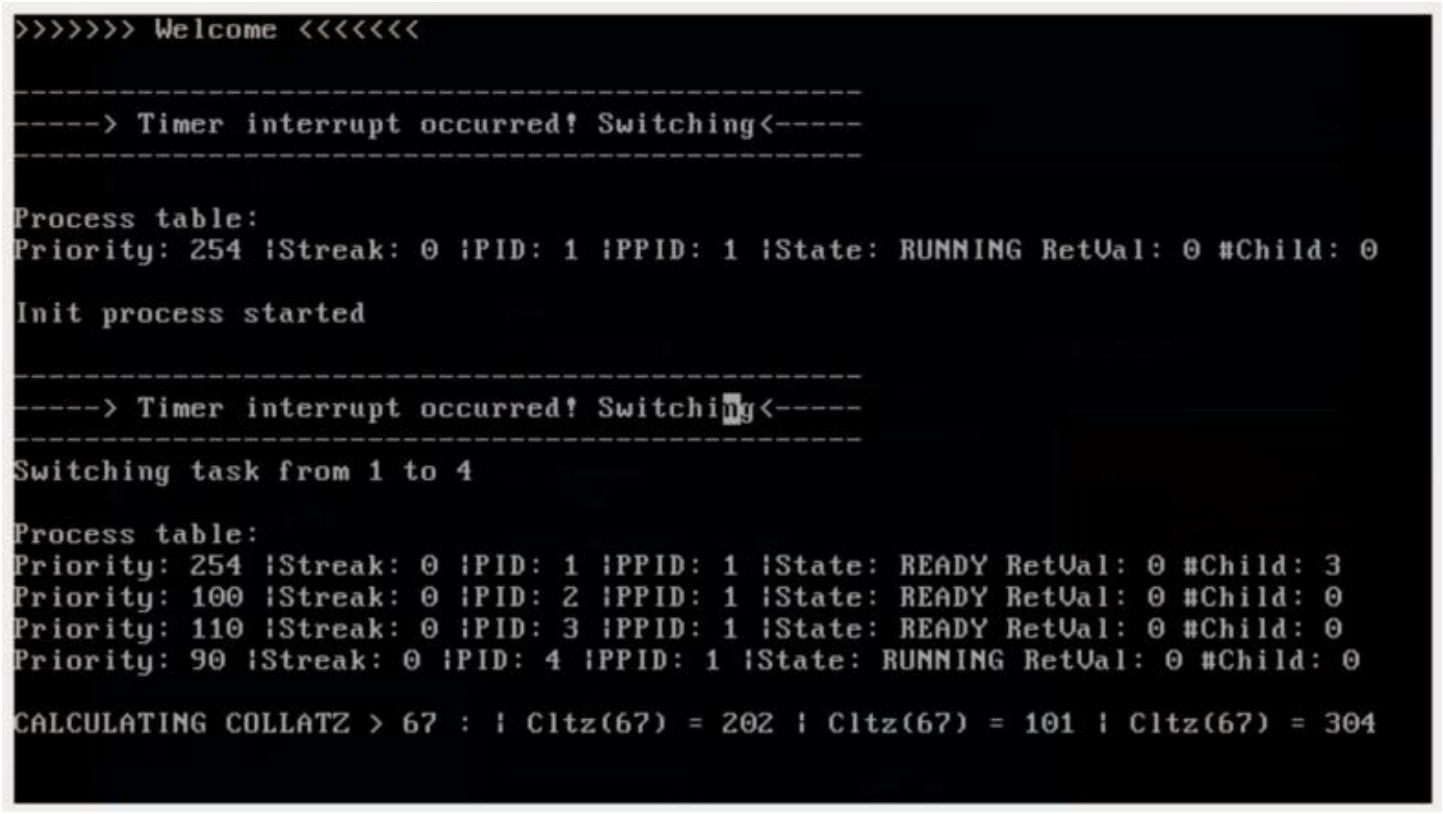
Long_running_program

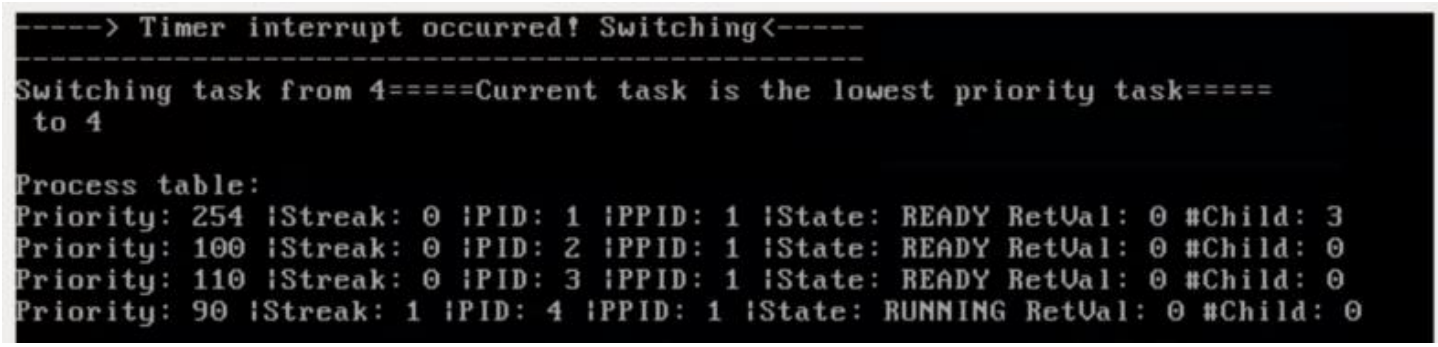
```
Process table:
Priority: 0 |Streak: 0 |PID: 1 |PPID: 1 |State: BLOCKED RetVal: 0 #Child: 6
Priority: 0 |Streak: 0 |PID: 2 |PPID: 1 |State: TERMINATED RetVal: 2 #Child: 0
Priority: 0 |Streak: 0 |PID: 3 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 4 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 5 |PPID: 1 |State: RUNNING RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 6 |PPID: 1 |State: READY RetVal: 0 #Child: 0
Priority: 0 |Streak: 0 |PID: 7 |PPID: 1 |State: READY RetVal: 0 #Child: 0
```

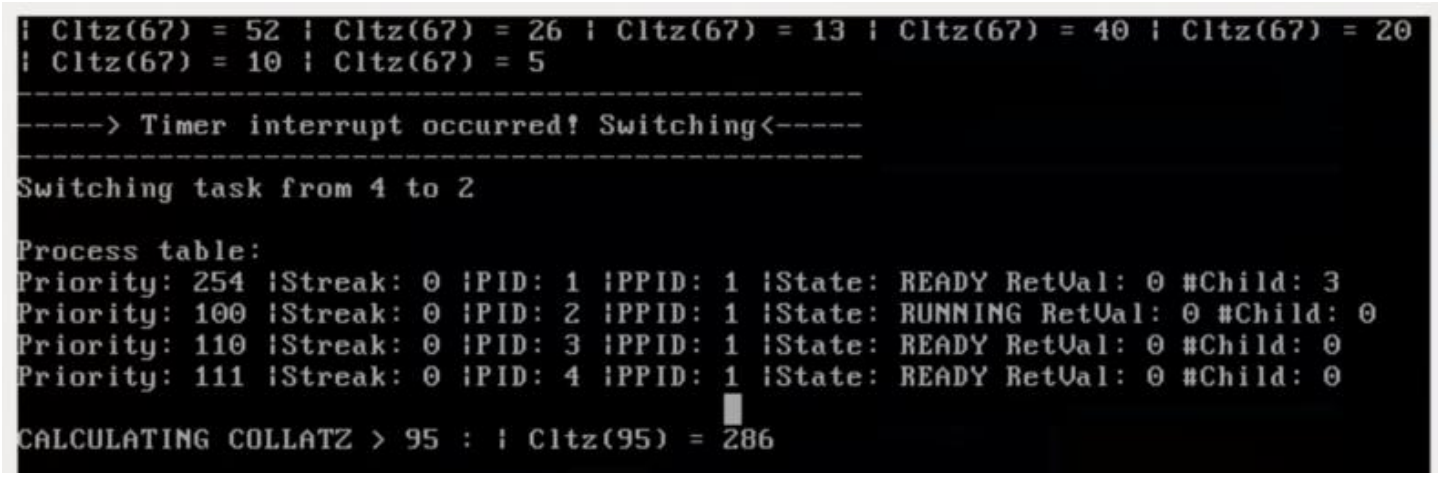
PART_B_THIRD_STRATEGY :



PART_B_DYNAMIC_PRIORITY_STRATEGY :

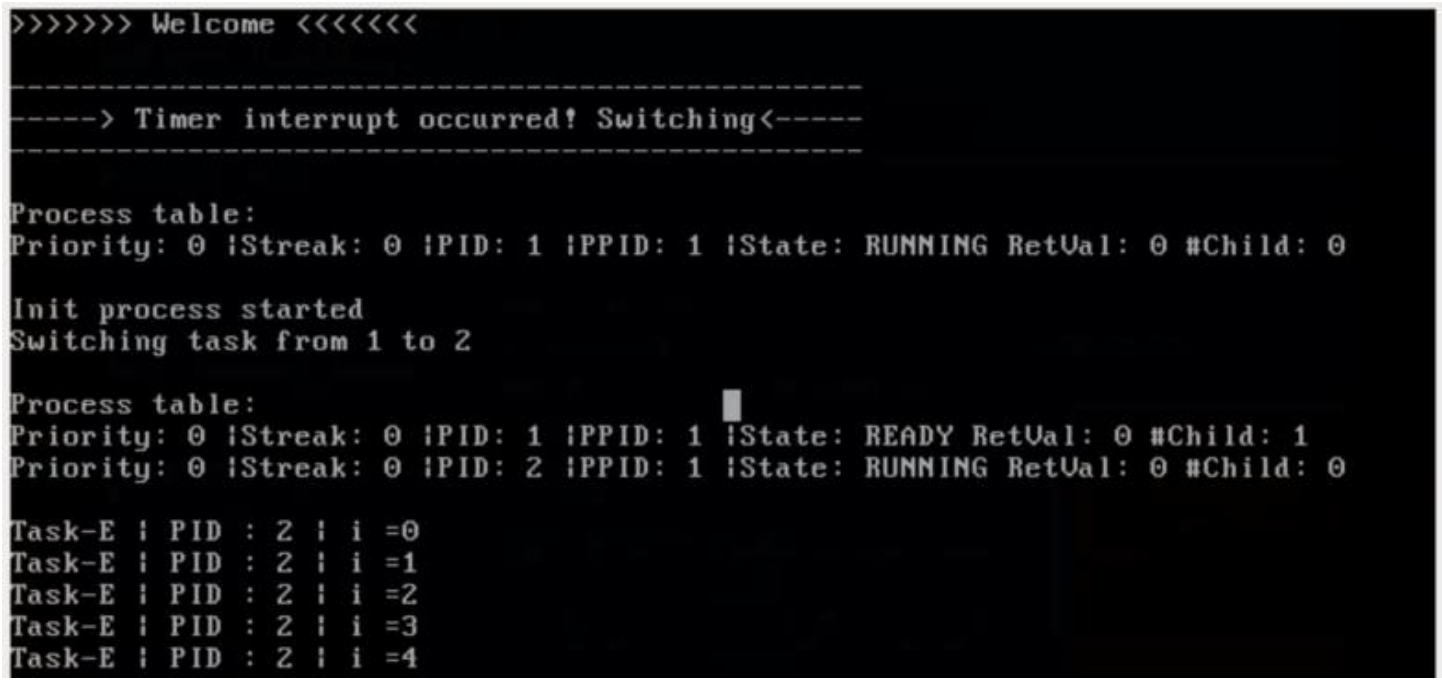


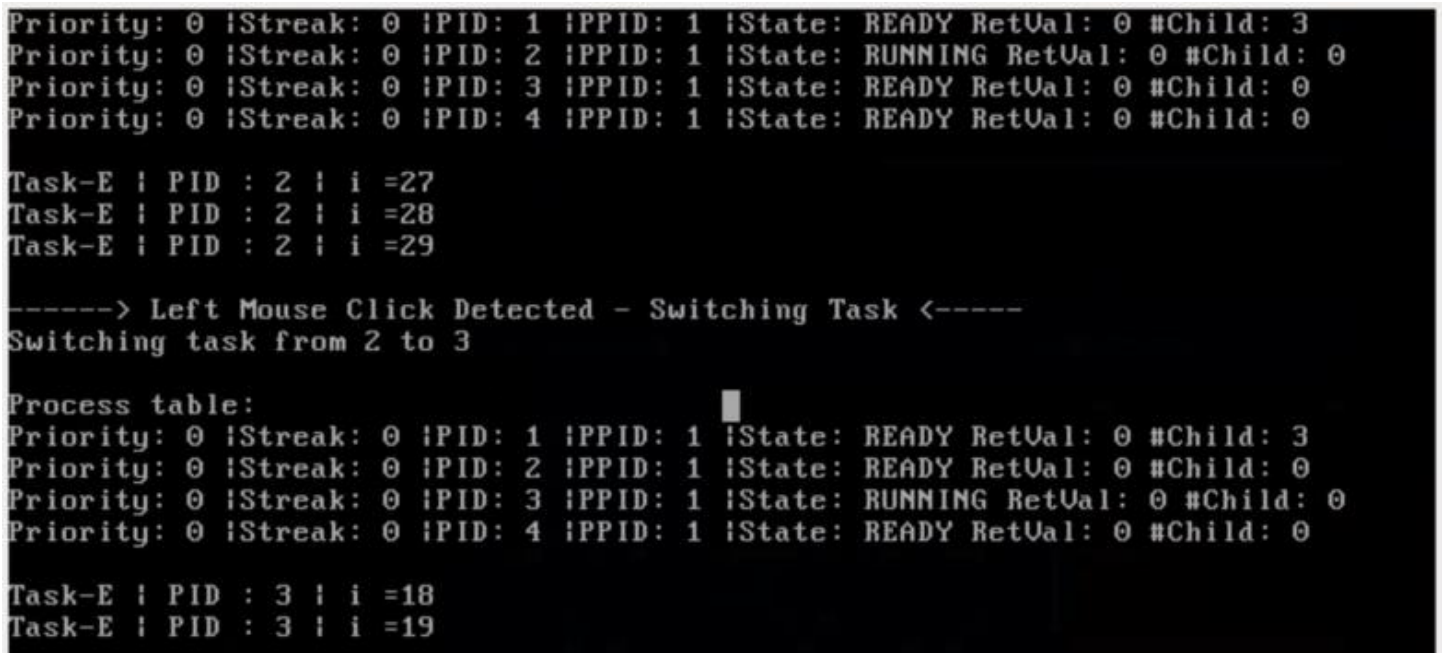




Its streak becomes from 1 to 2. Since streak limit is 2, task is switched and its priority becomes 111. Because highest priority was 110. Priority of init process is not considered. It will always have highest priority.

PART_C_INTERACTIVE_INPUT_STRATEGY :





MACRO COMMENT INSTRUCTIONS :

Kernel.cpp

```
// change scheduling type in multitasking.cpp
// #define PRIORITY_FLAG
#define NON_PRIORITY_FLAG

#define PART_A

// #define PART_B_First_Strategy
// #define PART_B_Second_Strategy
// #define PART_B_Third_Strategy
// #define PART_B_DYNAMIC_STRATEGY

// #define PART_C_First_Strategy

// #define FORK_TEST_1
// #define FORK_TEST_2
// #define GETPID_TEST
// #define WAITPID_TEST
// #define EXECVE_TEST
```

Multitasking.cpp

```
#define PRINT_FLAG

#define DEFAULT_SCHEDULE
//#define THIRD_STRATEGY_PRIORITY_SCHEDULE
//#define DYNAMIC_PRIORITY_SCHEDULE
```

For each part, remove comment from related parts. For example, for PART_A, remove comment from **#define PART_A** macro.

Also to adjust proper scheduling for:
- PART_A
- PART_B_First_Strategy
- PART_B_Second_Strategy
- PART_C_First_Strategy
- FORK_TEST_1
- FORK_TEST_2
- GETPID_TEST
- WAITPID_TEST
- EXECVE_TEST

remove comments only from **#define DEFAULT_SCHEDULE** macro in multitasking.cpp and **NON_PRIORITY_FLAG** macro in kernel.cpp. Because these tests use default schedule.

- Remove comment from **THIRD_STRATEGY_PRIORITY_SCHEDULE macro** in multitasking.cpp and **PRIORITY_FLAG** macro in kernel.cpp for PART_B_THIRD_STRATEGY.

- Remove comment from **DYNAMIC_PRIORITY_SCHEDULE** macro in multitasking.cpp and **PRIORITY_FLAG** macro in kernel.cpp for PART_B_DYNAMIC_STRATEGY.

!! Run each macro separately. Don't use multiple at the same time.
!! Don't use **NON_PRIORITY_FLAG** and **PRIORITY_FLAG** at the same time.
!! Don't use **DEFAULT_SCHEDULE** , **THIRD_STRATEGY_PRIORITY_SCHEDULE** and **DYNAMIC_PRIORITY_SCHEDULE at the same time.**

!! After adjusting macros, create .iso file with "make mykernel.iso" for each test.