GTU Department of Computer Engineering

CSE 312 / 504 - Spring 2023

Homework 1 Report

Reşit AYDIN

2001040004019

1. Introduction

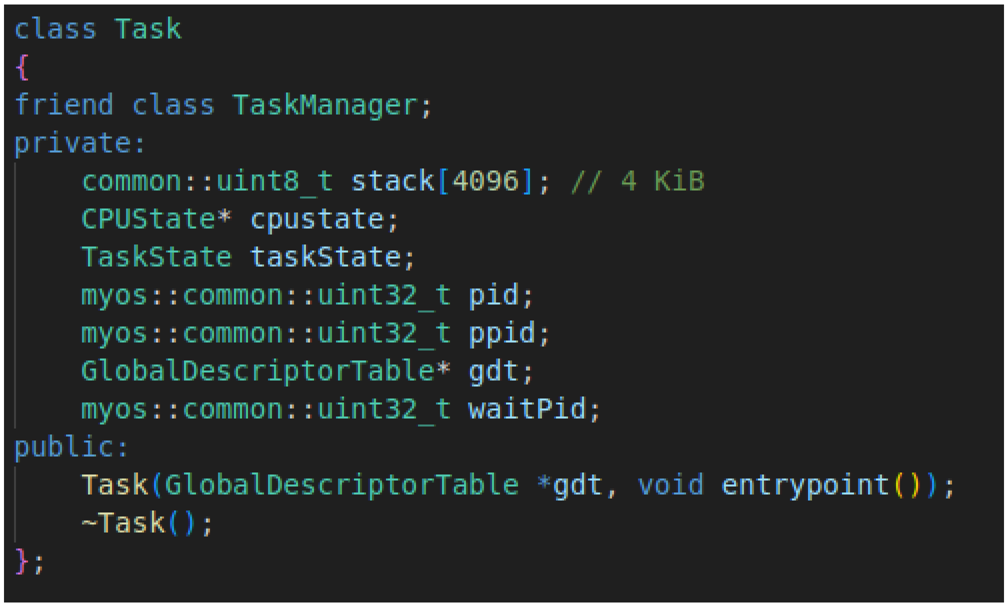
In collaboration with Engelmann videos and provided source codes, our task was to develop an operating system. While the initial project focused on implementing a basic operating system following the tutorials, we embarked on enhancing its functionalities and refining its design. In this documentation, I present an overview of the enhanced version of the operating system, detailing the key features, design decisions, and implementation details.

2. Design Decisions & Structures

The multitasking kernel is designed to support running multiple programs concurrently by utilizing processes and a task manager. Here are some key design decisions and structures:

2.1 Task

- The Task class represents an individual process in the system.
- Each task has its own CPU state (CPUState) which includes register values, instruction pointer, stack pointer, etc.
- Tasks are initialized with a stack of 4 KiB size to store local variables, function calls, and CPU state during execution.
- The task state can be one of READY, WAITING, or FINISHED, indicating whether the process is ready to run, waiting for another process, or has completed execution.



```cpp
class Task
{
friend class TaskManager;
private:
    common::uint8_t stack[4096]; // 4 KiB
    CPUState* cpustate;
    TaskState taskState;
    myos::common::uint32_t pid;
    myos::common::uint32_t ppid;
    GlobalDescriptorTable* gdt;
    myos::common::uint32_t waitPid;
public:
    Task(GlobalDescriptorTable *gdt, void entrypoint());
    ~Task();
};
```

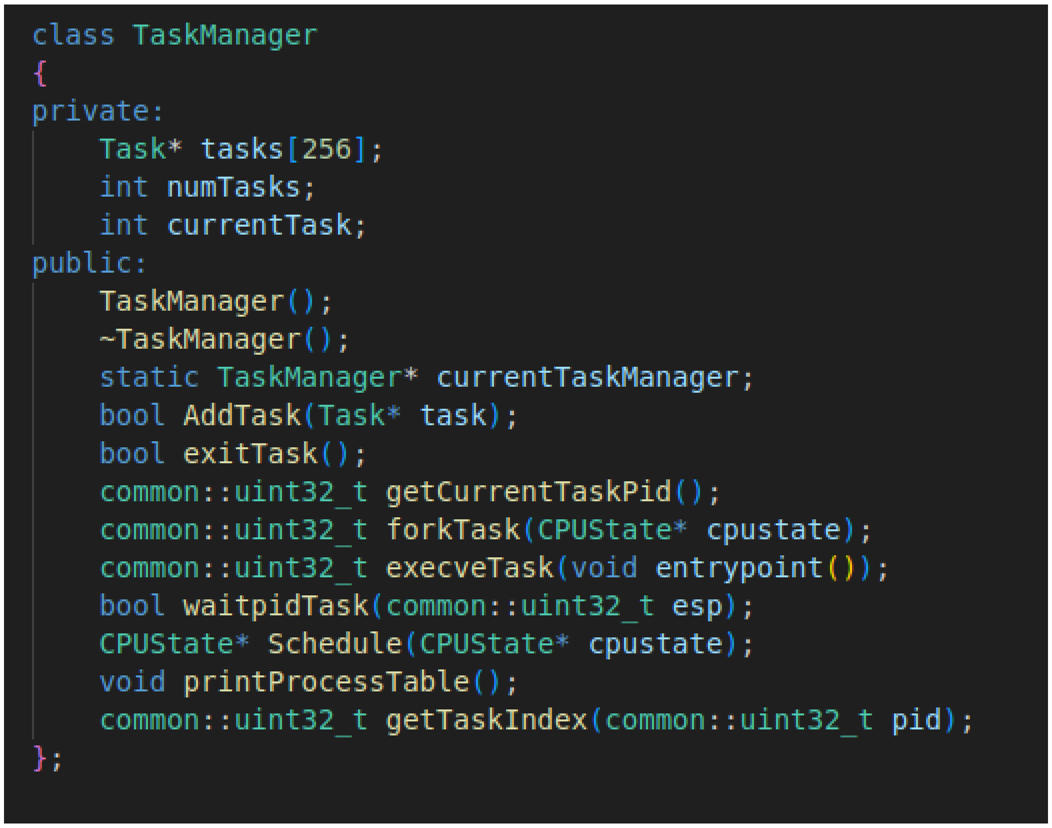
2.2 Task Manager

- The TaskManager class manages all tasks (processes) running in the system.

- It maintains an array of Task objects and keeps track of the number of tasks and the index of the current task.

- The task manager is responsible for adding tasks, scheduling tasks for execution, handling process exit, waiting for child processes, and printing the process table.

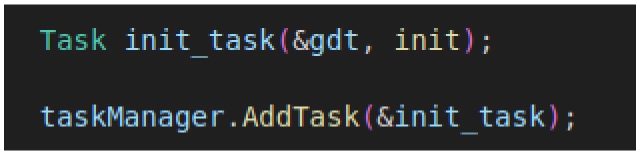
- Processes are represented by the Task class, each containing its own CPU state and stack. The task manager maintains a process table that keeps track of all running processes. The process table includes information such as process ID (PID), parent process ID (PPID), and the current state of each process. This allows the kernel to manage and schedule multiple processes effectively.



```cpp
class TaskManager
{
private:
    Task* tasks[256];
    int numTasks;
    int currentTask;
public:
    TaskManager();
    ~TaskManager();
    static TaskManager* currentTaskManager;
    bool AddTask(Task* task);
    bool exitTask();
    common::uint32_t getCurrentTaskPid();
    common::uint32_t forkTask(CPUState* cpustate);
    common::uint32_t execveTask(void entrypoint());
    bool waitpidTask(common::uint32_t esp);
    CPUState* Schedule(CPUState* cpustate);
    void printProcessTable();
    common::uint32_t getTaskIndex(common::uint32_t pid);
};
```

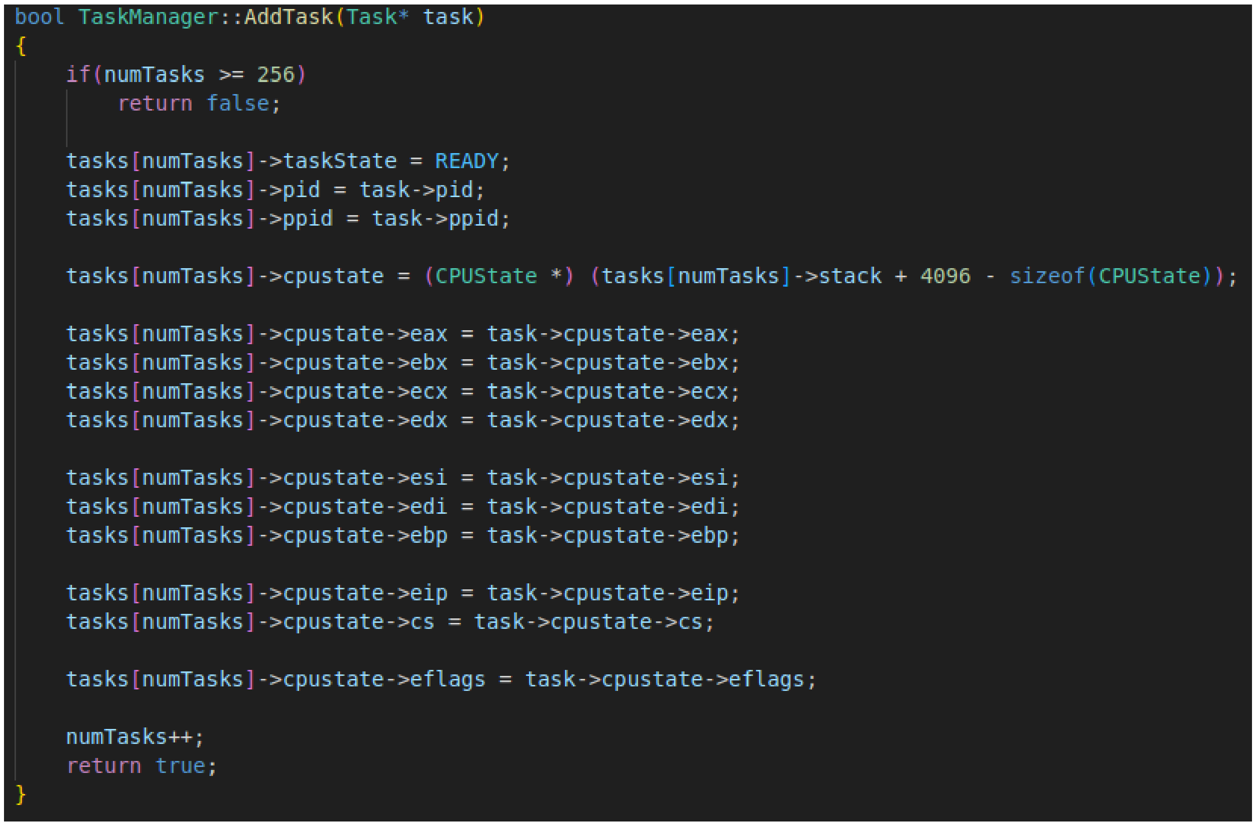
3. Loading Multiple Program into Memory

Tasks can be created from the kernelMain or from another task using fork syscall to be loaded into memory.



```cpp
Task init_task(&gdt, init);

taskManager.AddTask(&init_task);
```

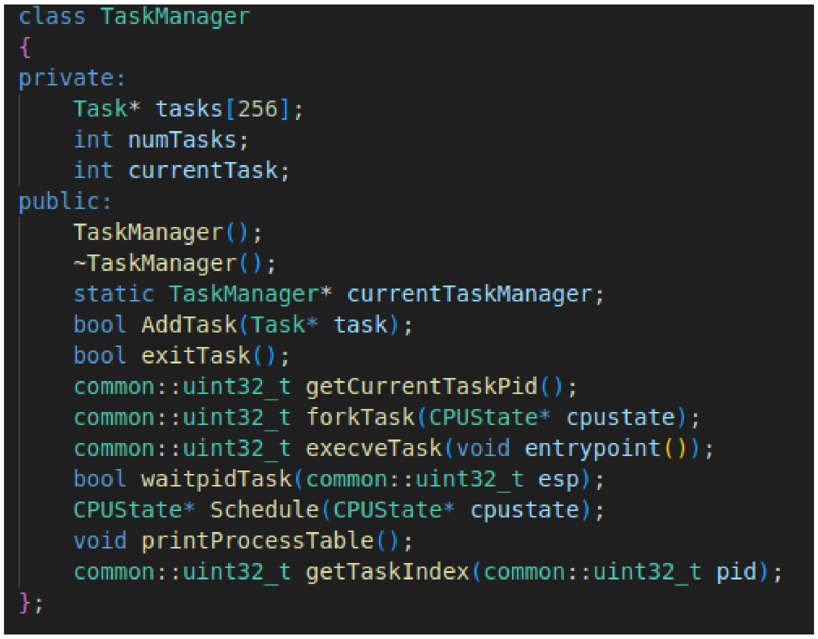
To load the programs into memory AddTask function is used. It takes pointer to a task and simply adds the task to the task array. With this functionality multiple programs can be loaded into memory by the TaskManager



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

    tasks[numTasks]->taskState = READY;
    tasks[numTasks]->pid = task->pid;
    tasks[numTasks]->ppid = task->ppid;

    tasks[numTasks]->cpustate = (CPUState *) (tasks[numTasks]->stack + 4096 - sizeof(CPUState));

    tasks[numTasks]->cpustate->eax = task->cpustate->eax;
    tasks[numTasks]->cpustate->ebx = task->cpustate->ebx;
    tasks[numTasks]->cpustate->ecx = task->cpustate->ecx;
    tasks[numTasks]->cpustate->edx = task->cpustate->edx;

    tasks[numTasks]->cpustate->esi = task->cpustate->esi;
    tasks[numTasks]->cpustate->edi = task->cpustate->edi;
    tasks[numTasks]->cpustate->ebp = task->cpustate->ebp;

    tasks[numTasks]->cpustate->eip = task->cpustate->eip;
    tasks[numTasks]->cpustate->cs = task->cpustate->cs;

    tasks[numTasks]->cpustate->eflags = task->cpustate->eflags;

    numTasks++;
    return true;
}
```

4. Multiprogramming

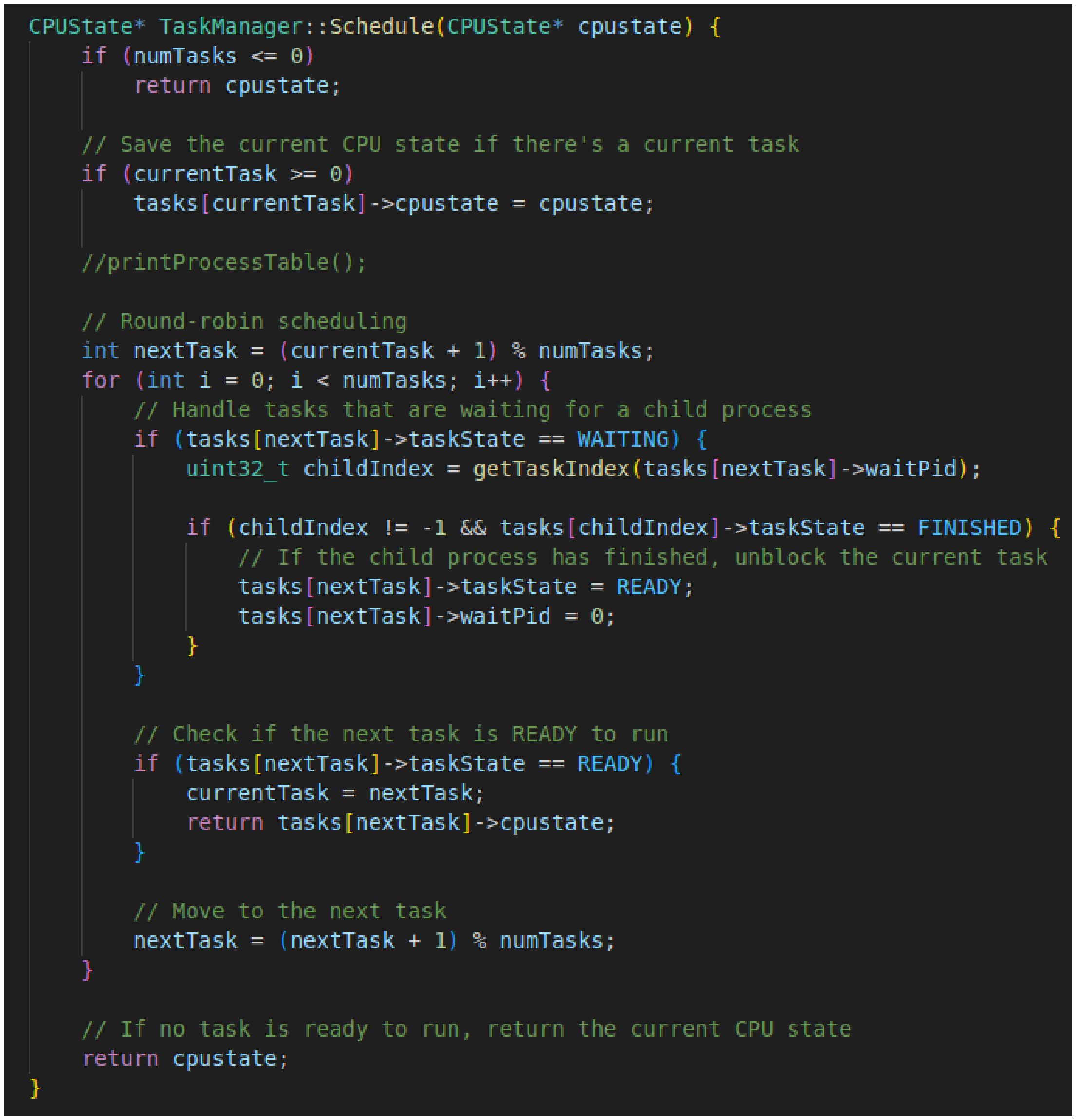
Multiprogramming is achieved through context switching and round-robin scheduling. The kernel periodically switches between tasks using a round-robin scheduling algorithm, allowing each process to execute for a fixed time slice before moving to the next process. This ensures fairness in CPU allocation and prevents any single process from monopolizing the CPU. It is done by TaskManager class. It has process table to keep multiple processes and scheduler functions switching between processes. In addition to that it contains implementation for the system call functions like fork, execve, waitpid etc.



```
class TaskManager
{
private:
    Task* tasks[256];
    int numTasks;
    int currentTask;
public:
    TaskManager();
    ~TaskManager();
    static TaskManager* currentTaskManager;
    bool AddTask(Task* task);
    bool exitTask();
    common::uint32_t getCurrentTaskPid();
    common::uint32_t forkTask(CPUState* cpustate);
    common::uint32_t execveTask(void entrypoint());
    bool waitpidTask(common::uint32_t esp);
    CPUState* Schedule(CPUState* cpustate);
    void printProcessTable();
    common::uint32_t getTaskIndex(common::uint32_t pid);
};
```

4.1 Context Switching and Round Robin Scheduling

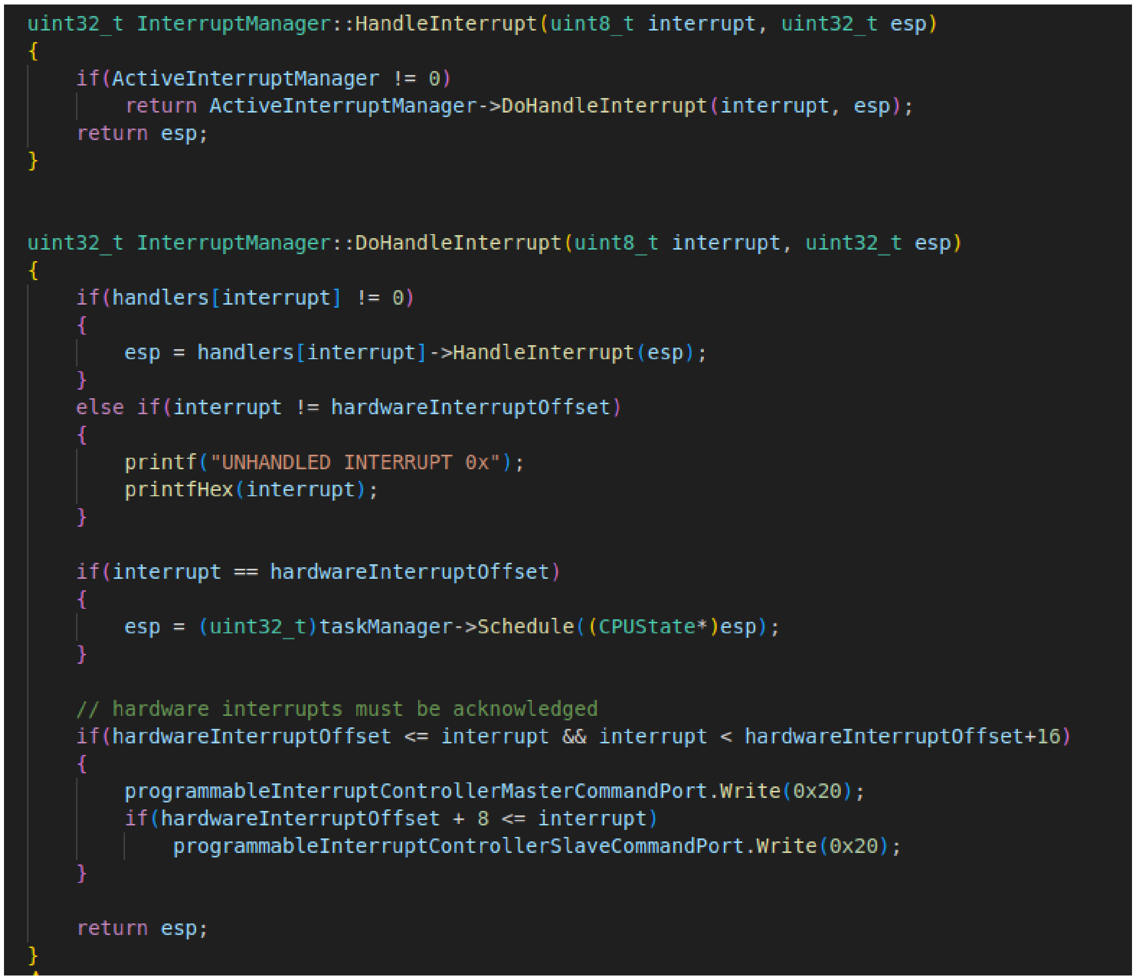
Context switching involves saving the CPU state of the current task and loading the CPU state of the next task. This is performed by the Schedule method in the TaskManager class. Round-robin scheduling ensures that tasks are executed in a cyclic order, with each task receiving an equal share of CPU time. There is also a logic added to schedule to execute waitpid correctly.



```
CPUState* TaskManager::Schedule(CPUState* cpustate) {
    if (numTasks <= 0)
        return cpustate;

    // Save the current CPU state if there's a current task
    if (currentTask >= 0)
        tasks[currentTask]->cpustate = cpustate;

    //printProcessTable();

    // Round-robin scheduling
    int nextTask = (currentTask + 1) % numTasks;
    for (int i = 0; i < numTasks; i++) {
        // Handle tasks that are waiting for a child process
        if (tasks[nextTask]->taskState == WAITING) {
            uint32_t childIndex = getTaskIndex(tasks[nextTask]->waitPid);

            if (childIndex != -1 && tasks[childIndex]->taskState == FINISHED) {
                // If the child process has finished, unblock the current task
                tasks[nextTask]->taskState = READY;
                tasks[nextTask]->waitPid = 0;
            }
        }

        // Check if the next task is READY to run
        if (tasks[nextTask]->taskState == READY) {
            currentTask = nextTask;
            return tasks[nextTask]->cpustate;
        }

        // Move to the next task
        nextTask = (nextTask + 1) % numTasks;
    }

    // If no task is ready to run, return the current CPU state
    return cpustate;
}
```

5. Interrupt Handling

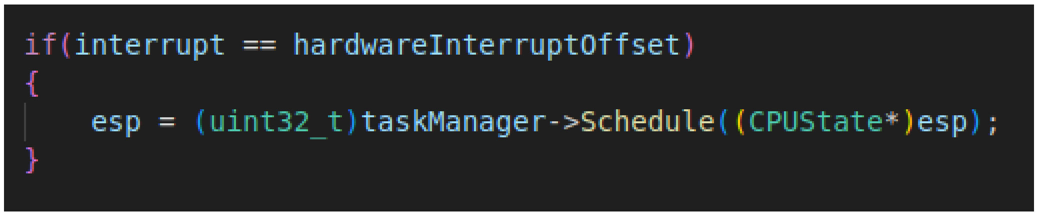
The kernel sets up interrupt handlers to handle hardware interrupts such as the timer interrupt, which triggers context switches between tasks. Additionally, system calls are used to request kernel services such as process creation, termination, and synchronization. I have InterruptManager class which contains IDT (interrupt descriptor table).



```
uint32_t InterruptManager::HandleInterrupt(uint8_t interrupt, uint32_t esp)
{
    if(ActiveInterruptManager != 0)
        return ActiveInterruptManager->DoHandleInterrupt(interrupt, esp);
    return esp;
}


uint32_t InterruptManager::DoHandleInterrupt(uint8_t interrupt, uint32_t esp)
{
    if(handlers[interrupt] != 0)
    {
        esp = handlers[interrupt]->HandleInterrupt(esp);
    }
    else if(interrupt != hardwareInterruptOffset)
    {
        printf("UNHANDLED INTERRUPT 0x");
        printfHex(interrupt);
    }

    if(interrupt == hardwareInterruptOffset)
    {
        esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
    }

    // hardware interrupts must be acknowledged
    if(hardwareInterruptOffset <= interrupt && interrupt < hardwareInterruptOffset+16)
    {
        programmableInterruptControllerMasterCommandPort.Write(0x20);
        if(hardwareInterruptOffset + 8 <= interrupt)
            programmableInterruptControllerSlaveCommandPort.Write(0x20);
    }

    return esp;
}
```

5.1 Timer Interrupt

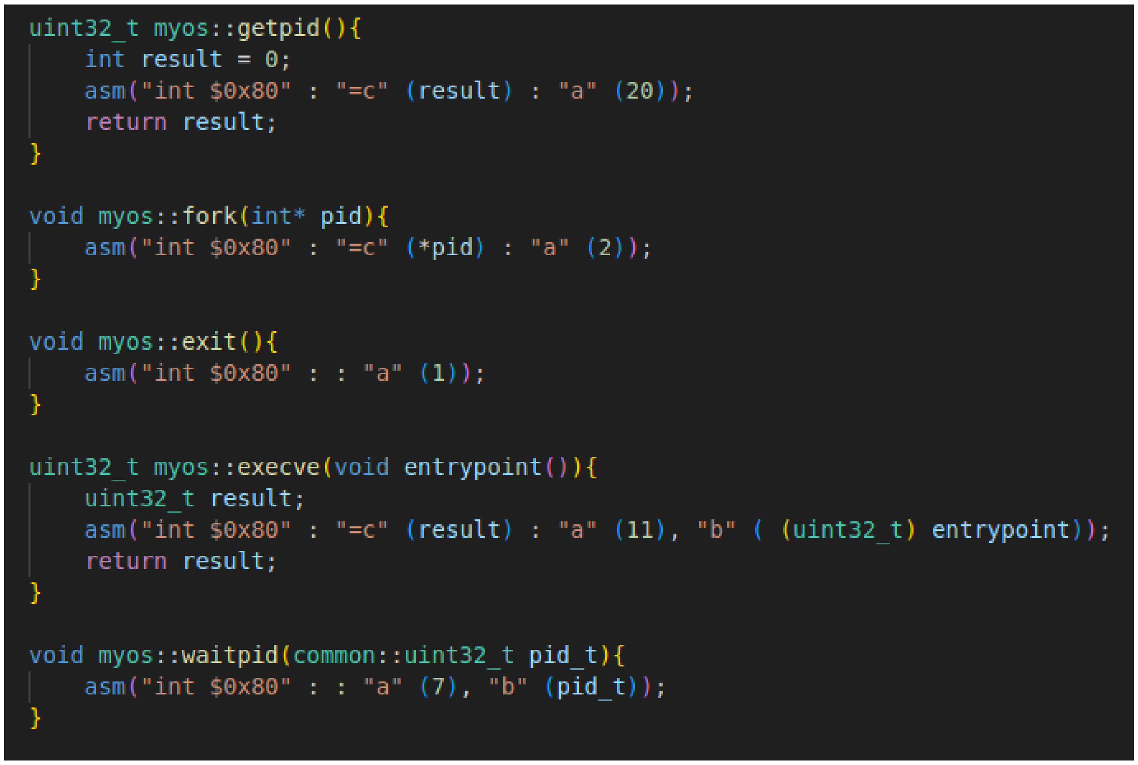
The timer interrupt is used to implement preemptive multitasking by periodically interrupting the CPU to trigger context switches. When the timer interrupt occurs, the kernel saves the CPU state of the current task and selects the next task to run based on the round-robin scheduling algorithm. Every time a timer interrupt occurs, scheduler runs.



```
if(interrupt == hardwareInterruptOffset)
{
    esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
}
```

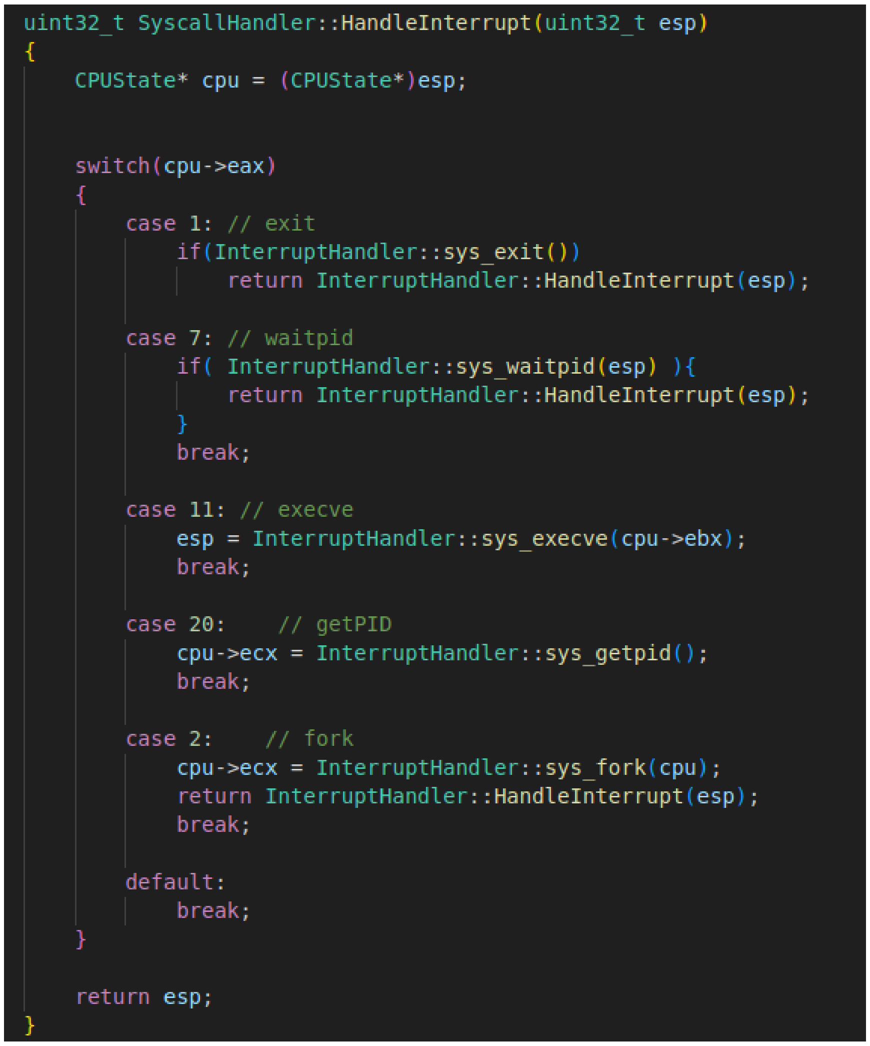
6. POSIX System Calls

System calls are done using asm (assembly) codes inside a cpp function. In these functions, the necessary arguments provided to registers and kernel is being called using “$0x80”.



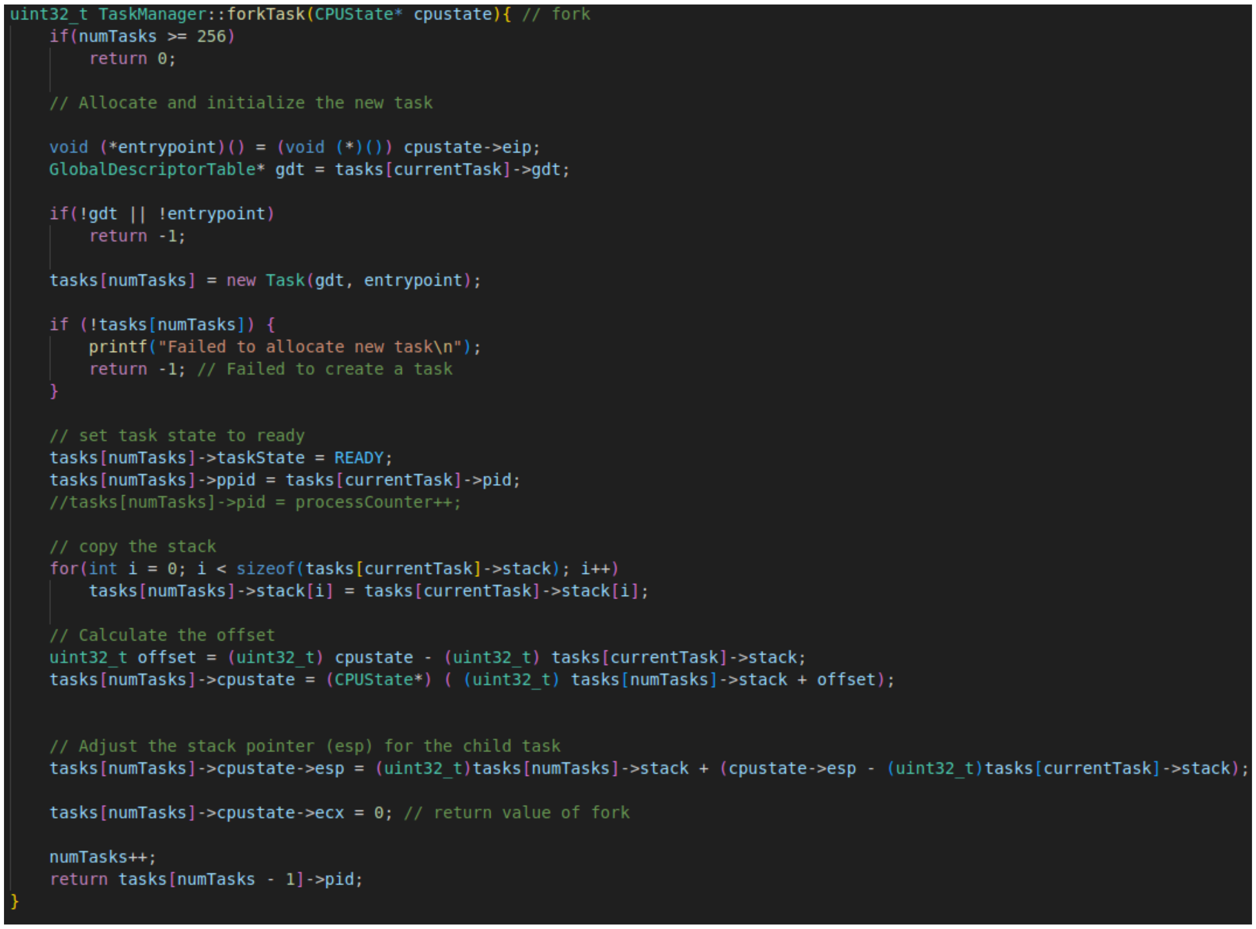
```
uint32_t myos::getpid(){
    int result = 0;
    asm("int $0x80" : "=c" (result) : "a" (20));
    return result;
}

void myos::fork(int* pid){
    asm("int $0x80" : "=c" (*pid) : "a" (2));
}

void myos::exit(){
    asm("int $0x80" : : "a" (1));
}

uint32_t myos::execve(void entrypoint()){
    uint32_t result;
    asm("int $0x80" : "=c" (result) : "a" (11), "b" ( (uint32_t) entrypoint));
    return result;
}

void myos::waitpid(common::uint32_t pid_t){
    asm("int $0x80" : : "a" (7), "b" (pid_t));
}
```

After the system calls happened, syscallHandler handles the interrupt by calling relevant functions for each interrupt type.


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


    switch(cpu->eax)
    {
        case 1: // exit
            if(InterruptHandler::sys_exit())
                return InterruptHandler::HandleInterrupt(esp);

        case 7: // waitpid
            if( InterruptHandler::sys_waitpid(esp) ){
                return InterruptHandler::HandleInterrupt(esp);
            }
            break;

        case 11: // execve
            esp = InterruptHandler::sys_execve(cpu->ebx);
            break;

        case 20:    // getPID
            cpu->ecx = InterruptHandler::sys_getpid();
            break;

        case 2:    // fork
            cpu->ecx = InterruptHandler::sys_fork(cpu);
            return InterruptHandler::HandleInterrupt(esp);
            break;

        default:
            break;
    }

    return esp;
}
```

6.1 Fork

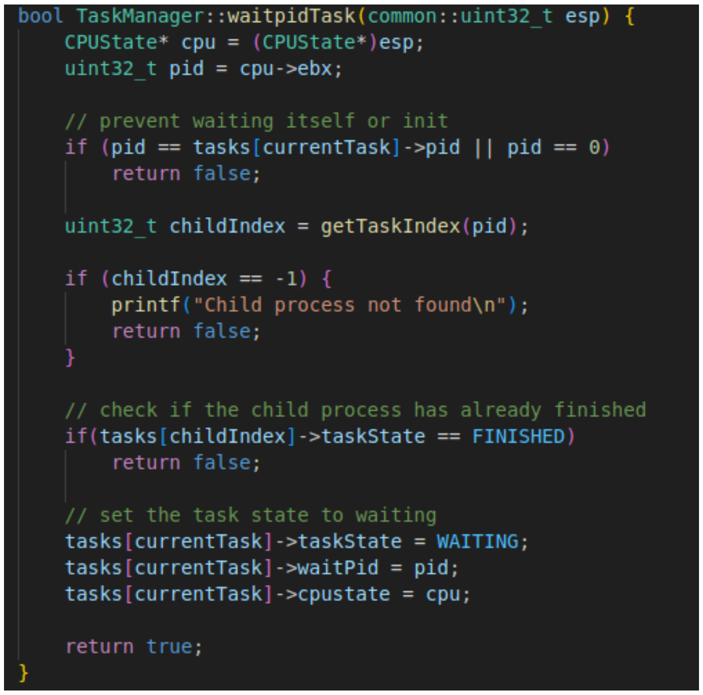
In my implementation, fork creates a new task by duplicating the current task's stack and CPU state. It first checks if the maximum number of tasks is reached, then initializes a new task with the same entry point and global descriptor table (GDT) as the current task. The new task's stack is a copy of the current task's stack, and the CPU state is adjusted to reflect the new stack location. The new task is set to the READY state, its parent process ID (PPID) is set to the current task's process ID (PID), and the fork returns the new task's PID.



```
uint32_t TaskManager::forkTask(CPUState* cpustate){ // fork
    if(numTasks >= 256)
        return 0;

    // Allocate and initialize the new task

    void (*entrypoint)() = (void (*)()) cpustate->eip;
    GlobalDescriptorTable* gdt = tasks[currentTask]->gdt;

    if(!gdt || !entrypoint)
        return -1;

    tasks[numTasks] = new Task(gdt, entrypoint);

    if (!tasks[numTasks]) {
        printf("Failed to allocate new task\n");
        return -1; // Failed to create a task
    }

    // set task state to ready
    tasks[numTasks]->taskState = READY;
    tasks[numTasks]->ppid = tasks[currentTask]->pid;
    //tasks[numTasks]->pid = processCounter++;

    // copy the stack
    for(int i = 0; i < sizeof(tasks[currentTask]->stack); i++)
        tasks[numTasks]->stack[i] = tasks[currentTask]->stack[i];

    // Calculate the offset
    uint32_t offset = (uint32_t) cpustate - (uint32_t) tasks[currentTask]->stack;
    tasks[numTasks]->cpustate = (CPUState*) ( (uint32_t) tasks[numTasks]->stack + offset);


    // Adjust the stack pointer (esp) for the child task
    tasks[numTasks]->cpustate->esp = (uint32_t)tasks[numTasks]->stack + (cpustate->esp - (uint32_t)tasks[currentTask]->stack);

    tasks[numTasks]->cpustate->ecx = 0; // return value of fork

    numTasks++;
    return tasks[numTasks - 1]->pid;
}
```

6.2 Waitpid

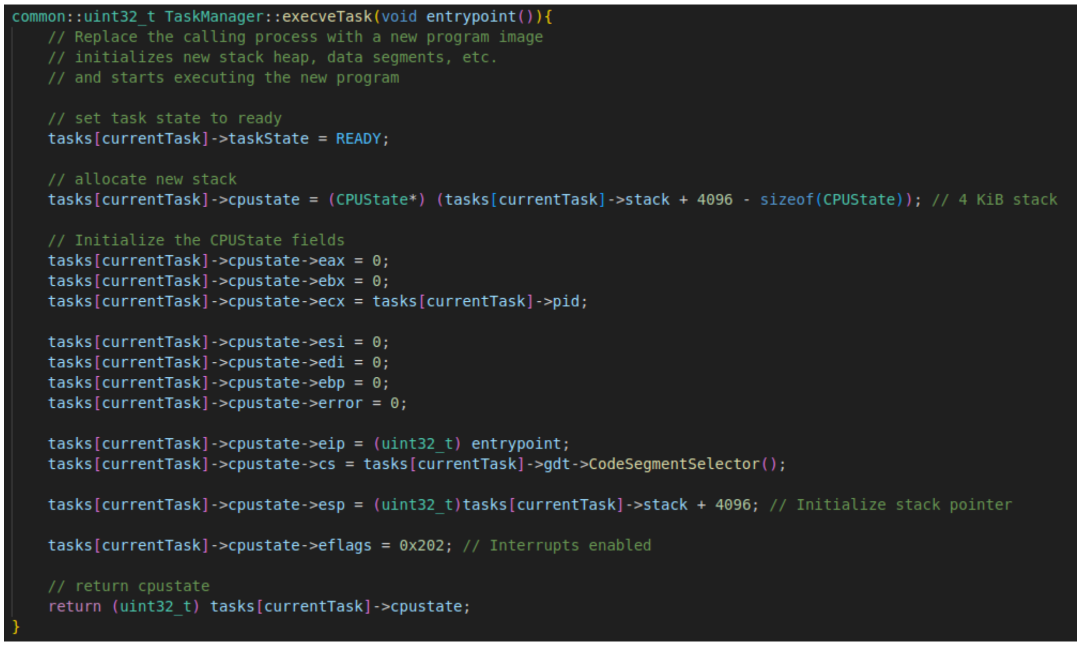
Waitpid system call, which allows a task to wait for a specific child process to finish. It first retrieves the process ID (PID) from the CPU state and checks if the PID is valid (not the current task or the init task with PID 0). It then searches for the child process in the task list. If the child process is not found or has already finished, the function returns false. If the child process is still running, the current task's state is set to WAITING, and it saves the CPU state to resume later. The current task also records the PID of the child process it is waiting for.



```cpp
bool TaskManager::waitpidTask(common::uint32_t esp) {
    CPUState* cpu = (CPUState*)esp;
    uint32_t pid = cpu->ebx;

    // prevent waiting itself or init
    if (pid == tasks[currentTask]->pid || pid == 0)
        return false;

    uint32_t childIndex = getTaskIndex(pid);

    if (childIndex == -1) {
        printf("Child process not found\n");
        return false;
    }

    // check if the child process has already finished
    if(tasks[childIndex]->taskState == FINISHED)
        return false;

    // set the task state to waiting
    tasks[currentTask]->taskState = WAITING;
    tasks[currentTask]->waitPid = pid;
    tasks[currentTask]->cpustate = cpu;

    return true;
}
```

6.3. Execve

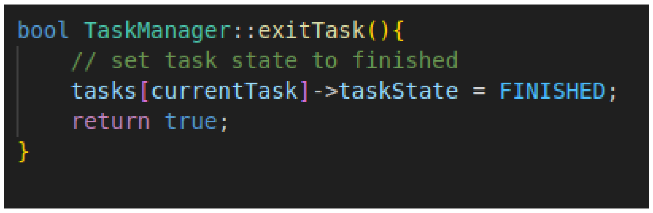
The execveTask function replaces the current task's program with a new one specified by entrypoint. It reinitializes the task's stack and CPU state, setting up the new program's entry point and ensuring the task is ready to run with the new context.



```cpp
common::uint32_t TaskManager::execveTask(void entrypoint()){
    // Replace the calling process with a new program image
    // initializes new stack heap, data segments, etc.
    // and starts executing the new program

    // set task state to ready
    tasks[currentTask]->taskState = READY;

    // allocate new stack
    tasks[currentTask]->cpustate = (CPUState*) (tasks[currentTask]->stack + 4096 - sizeof(CPUState)); // 4 KiB stack

    // Initialize the CPUState fields
    tasks[currentTask]->cpustate->eax = 0;
    tasks[currentTask]->cpustate->ebx = 0;
    tasks[currentTask]->cpustate->ecx = tasks[currentTask]->pid;

    tasks[currentTask]->cpustate->esi = 0;
    tasks[currentTask]->cpustate->edi = 0;
    tasks[currentTask]->cpustate->ebp = 0;
    tasks[currentTask]->cpustate->error = 0;

    tasks[currentTask]->cpustate->eip = (uint32_t) entrypoint;
    tasks[currentTask]->cpustate->cs = tasks[currentTask]->gdt->CodeSegmentSelector();

    tasks[currentTask]->cpustate->esp = (uint32_t)tasks[currentTask]->stack + 4096; // Initialize stack pointer

    tasks[currentTask]->cpustate->eflags = 0x202; // Interrupts enabled

    // return cpustate
    return (uint32_t) tasks[currentTask]->cpustate;
}
```

6.4 Exit

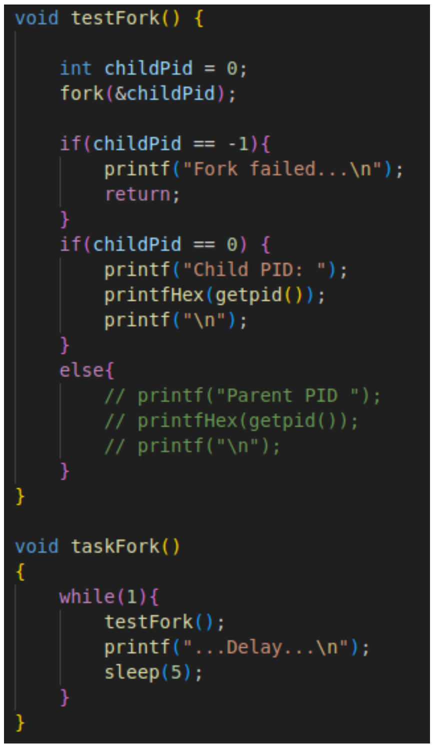
Exit function simply set task's state to finished.



```cpp
bool TaskManager::exitTask(){
    // set task state to finished
    tasks[currentTask]->taskState = FINISHED;
    return true;
}
```

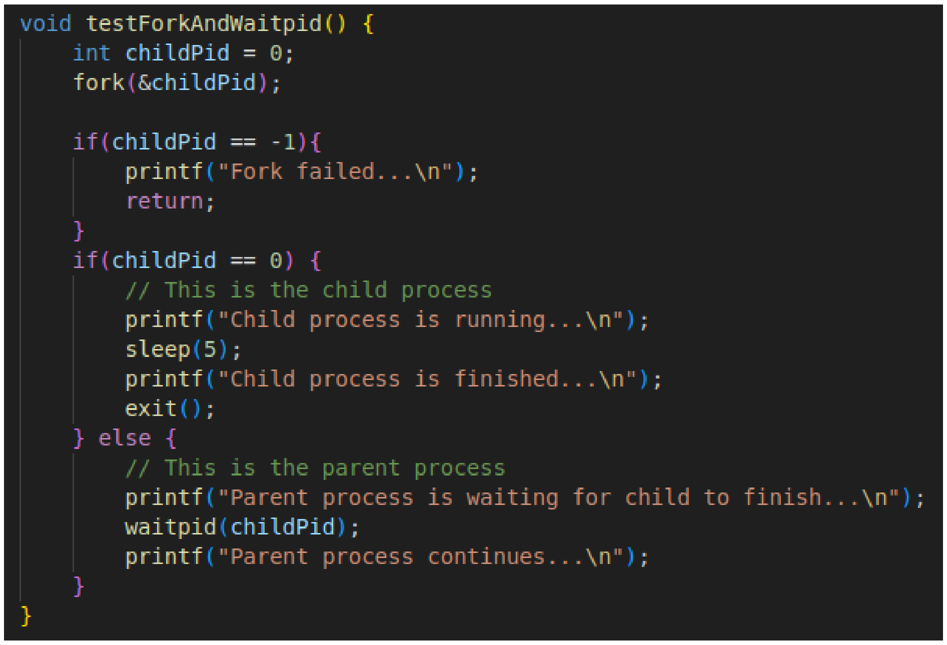
7. Test Cases

7.1 Fork Test Function



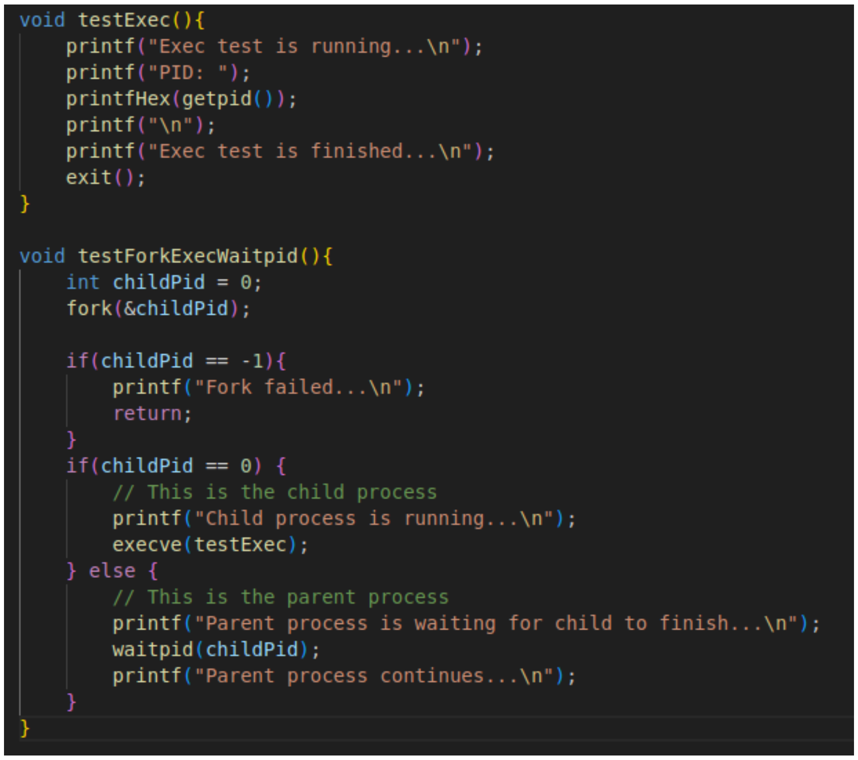
```cpp
void testFork() {

    int childPid = 0;
    fork(&childPid);

    if(childPid == -1){
        printf("Fork failed...\n");
        return;
    }
    if(childPid == 0) {
        printf("Child PID: ");
        printfHex(getpid());
        printf("\n");
    }
    else{
        // printf("Parent PID ");
        // printfHex(getpid());
        // printf("\n");
    }
}

void taskFork()
{
    while(1){
        testFork();
        printf("...Delay...\n");
        sleep(5);
    }
}
```

7.2 Waitpid with fork Test Function



```c
void testForkAndWaitpid() {
    int childPid = 0;
    fork(&childPid);

    if(childPid == -1){
        printf("Fork failed...\n");
        return;
    }
    if(childPid == 0) {
        // This is the child process
        printf("Child process is running...\n");
        sleep(5);
        printf("Child process is finished...\n");
        exit();
    } else {
        // This is the parent process
        printf("Parent process is waiting for child to finish...\n");
        waitpid(childPid);
        printf("Parent process continues...\n");
    }
}
```

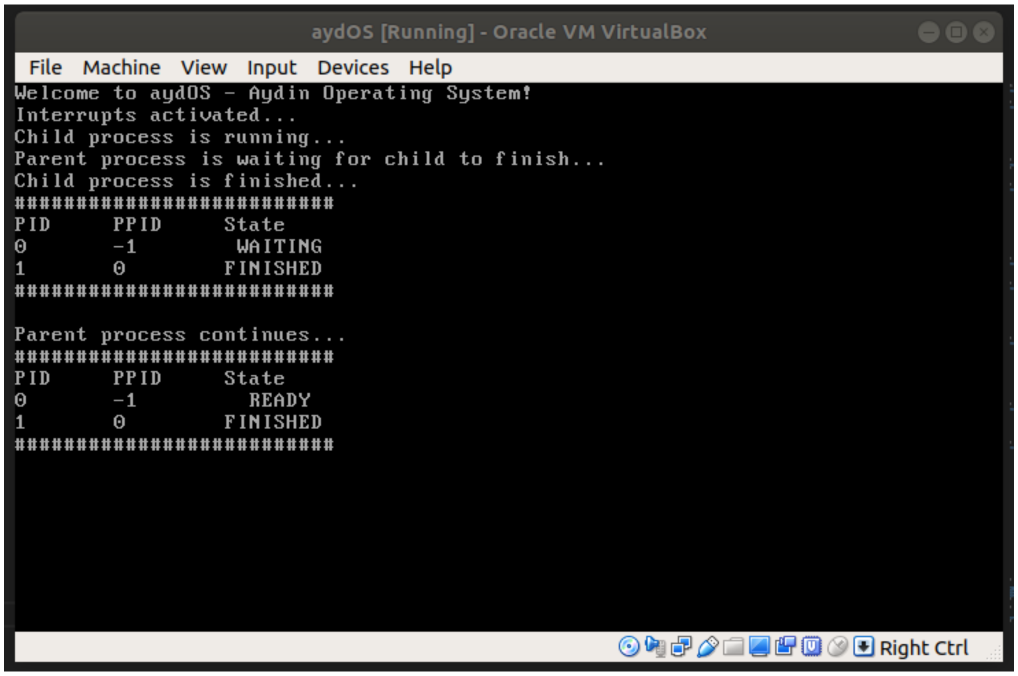
7.3 Waitpid with fork and execv Test Function:



```c
void testExec(){
    printf("Exec test is running...\n");
    printf("PID: ");
    printfHex(getpid());
    printf("\n");
    printf("Exec test is finished...\n");
    exit();
}

void testForkExecWaitpid(){
    int childPid = 0;
    fork(&childPid);

    if(childPid == -1){
        printf("Fork failed...\n");
        return;
    }
    if(childPid == 0) {
        // This is the child process
        printf("Child process is running...\n");
        execve(testExec);
    } else {
        // This is the parent process
        printf("Parent process is waiting for child to finish...\n");
        waitpid(childPid);
        printf("Parent process continues...\n");
    }
}
```

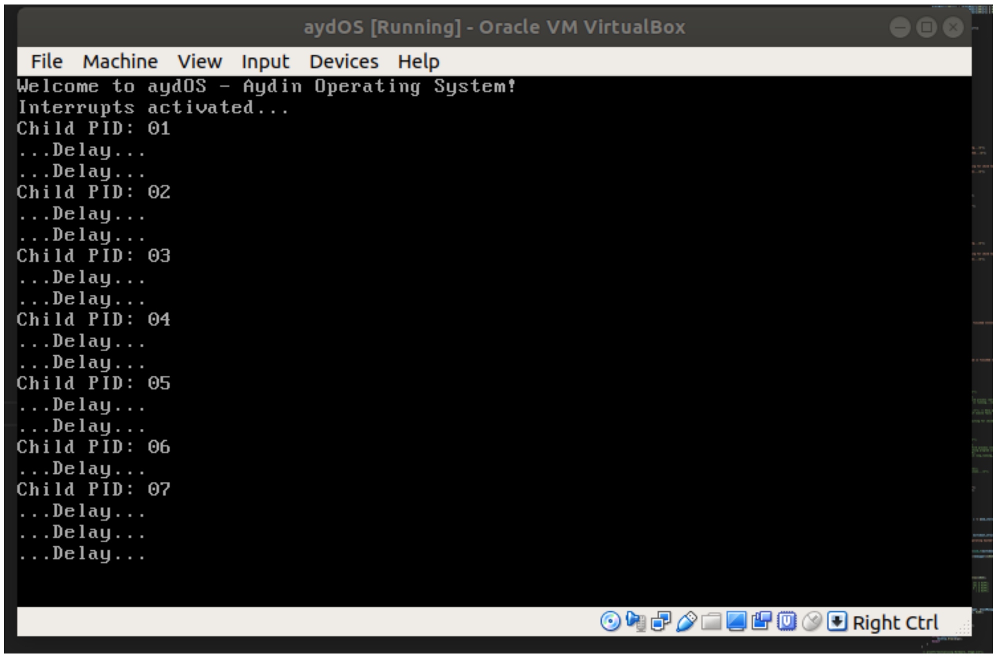
8. Running Results

8.1 Process Table with Sample System Call


aydOS [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

```
Welcome to aydOS - Aydin Operating System!
Interrupts activated...
Child process is running...
Parent process is waiting for child to finish...
Child process is finished...
##########################
PID      PPID      State
0        -1         WAITING
1        0         FINISHED
##########################

Parent process continues...
##########################
PID      PPID      State
0        -1          READY
1        0         FINISHED
##########################
```

8.2 Fork

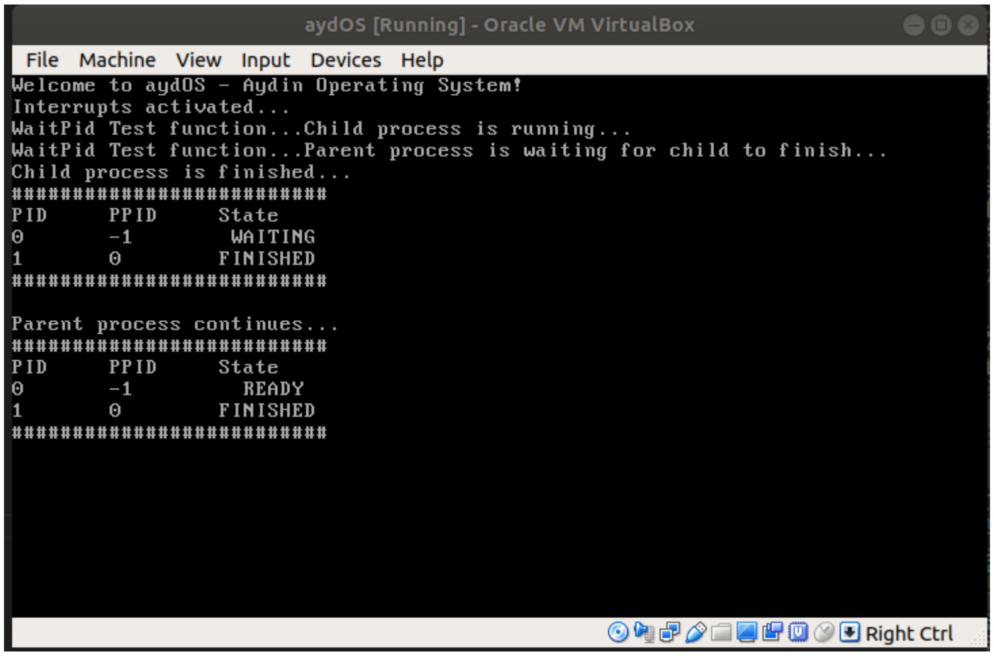

aydOS [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

```
Welcome to aydOS - Aydin Operating System!
Interrupts activated...
Child PID: 01
...Delay...
...Delay...
Child PID: 02
...Delay...
...Delay...
Child PID: 03
...Delay...
...Delay...
Child PID: 04
...Delay...
...Delay...
Child PID: 05
...Delay...
...Delay...
Child PID: 06
...Delay...
Child PID: 07
...Delay...
...Delay...
...Delay...
```

At the moment I took screenshot, 3 fork happened which there should be 8 tasks in total which in my case it executes correctly.

8.3 Waitpid

For better illustration, I tested waitpid by creating a new process using fork. In parent, I call waitpid to wait for child to terminate to continue its execution.


aydOS [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

```
Welcome to aydOS - Aydin Operating System!
Interrupts activated...
WaitPid Test function...Child process is running...
WaitPid Test function...Parent process is waiting for child to finish...
Child process is finished...
##########################
PID     PPID     State
0       -1        WAITING
1       0        FINISHED
##########################

Parent process continues...
##########################
PID     PPID     State
0       -1          READY
1       0        FINISHED
##########################
```

8.4 Execve Test with Fork and Waitpid

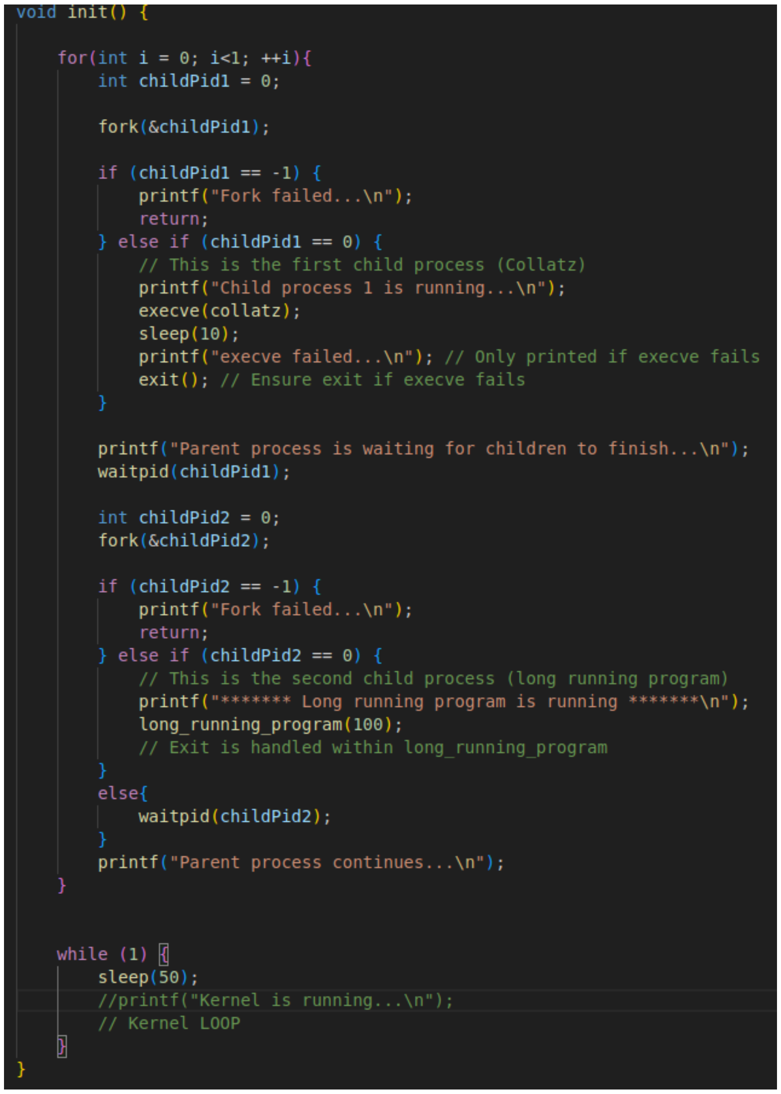
I tested this for the programs given in pdf. (Collatz and Long Running Program)

Strategy

I loaded each program 3 times, starting them and will enter an

infinite loop until all the processes terminate.

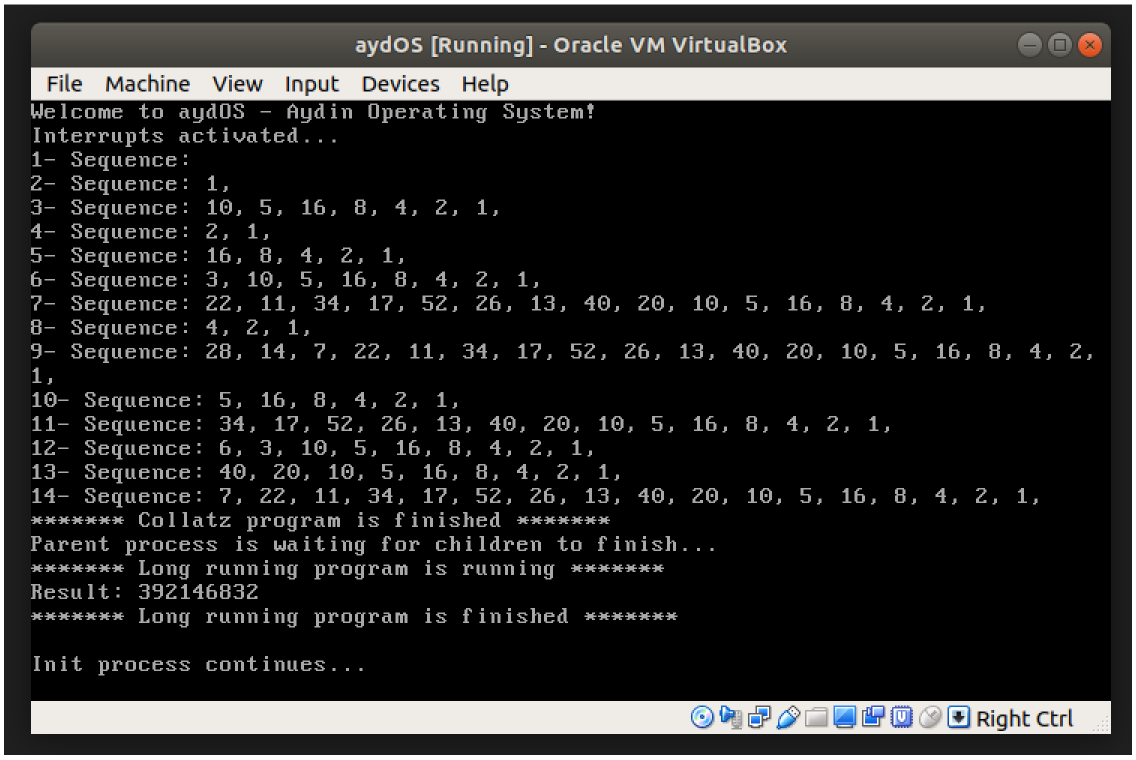
The function:



```
void init() {

    for(int i = 0; i<1; ++i){
        int childPid1 = 0;

        fork(&childPid1);

        if (childPid1 == -1) {
            printf("Fork failed...\n");
            return;
        } else if (childPid1 == 0) {
            // This is the first child process (Collatz)
            printf("Child process 1 is running...\n");
            execve(collatz);
            sleep(10);
            printf("execve failed...\n"); // Only printed if execve fails
            exit(); // Ensure exit if execve fails
        }

        printf("Parent process is waiting for children to finish...\n");
        waitpid(childPid1);

        int childPid2 = 0;
        fork(&childPid2);

        if (childPid2 == -1) {
            printf("Fork failed...\n");
            return;
        } else if (childPid2 == 0) {
            // This is the second child process (long running program)
            printf("******* Long running program is running *******\n");
            long_running_program(100);
            // Exit is handled within long_running_program
        }
        else{
            waitpid(childPid2);
        }
        printf("Parent process continues...\n");
    }


    while (1) {
        sleep(50);
        //printf("Kernel is running...\n");
        // Kernel LOOP
    }
}
```

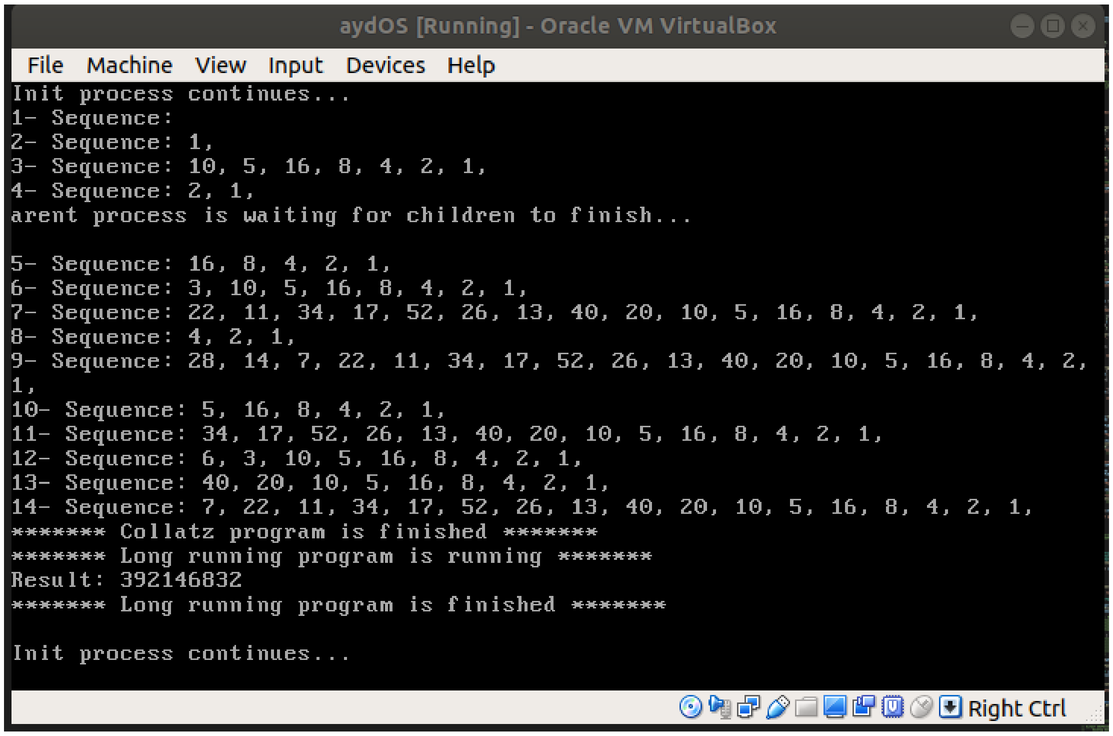
The result:

First execution of loop:


aydOS [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

```
Welcome to aydOS - Aydin Operating System!
Interrupts activated...
1- Sequence:
2- Sequence: 1,
3- Sequence: 10, 5, 16, 8, 4, 2, 1,
4- Sequence: 2, 1,
5- Sequence: 16, 8, 4, 2, 1,
6- Sequence: 3, 10, 5, 16, 8, 4, 2, 1,
7- Sequence: 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1,
8- Sequence: 4, 2, 1,
9- Sequence: 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2,
1,
10- Sequence: 5, 16, 8, 4, 2, 1,
11- Sequence: 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1,
12- Sequence: 6, 3, 10, 5, 16, 8, 4, 2, 1,
13- Sequence: 40, 20, 10, 5, 16, 8, 4, 2, 1,
14- Sequence: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1,
******* Collatz program is finished *******
Parent process is waiting for children to finish...
******* Long running program is running ********
Result: 392146832
******* Long running program is finished *******

Init process continues...
```

Last execution of loop:


aydOS [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

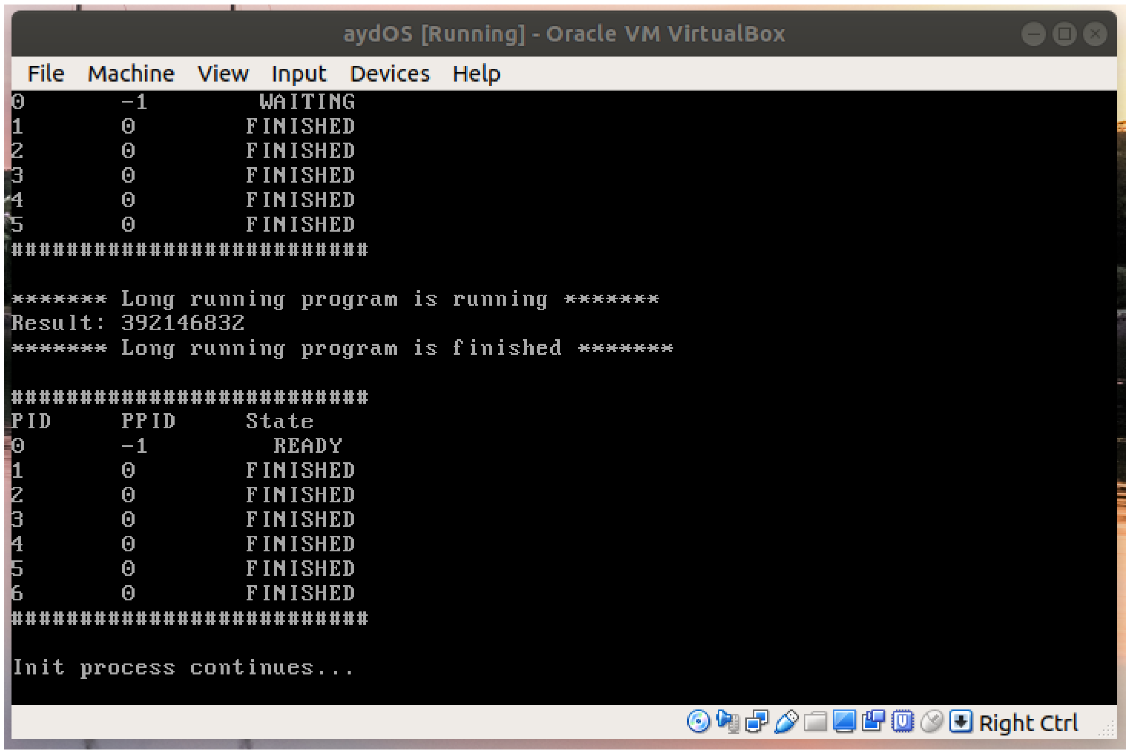
```
Init process continues...
1- Sequence:
2- Sequence: 1,
3- Sequence: 10, 5, 16, 8, 4, 2, 1,
4- Sequence: 2, 1,
arent process is waiting for children to finish...

5- Sequence: 16, 8, 4, 2, 1,
6- Sequence: 3, 10, 5, 16, 8, 4, 2, 1,
7- Sequence: 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1,
8- Sequence: 4, 2, 1,
9- Sequence: 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2,
1,
10- Sequence: 5, 16, 8, 4, 2, 1,
11- Sequence: 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1,
12- Sequence: 6, 3, 10, 5, 16, 8, 4, 2, 1,
13- Sequence: 40, 20, 10, 5, 16, 8, 4, 2, 1,
14- Sequence: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1,
******* Collatz program is finished *******
******* Long running program is running *******
Result: 392146832
******* Long running program is finished *******

Init process continues...
```

I tested with these parameters:

collatz -> 15

long_running_program -> 1000

Last screenshot of my OS with process table:


aydOS [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

```
0        -1         WAITING
1        0         FINISHED
2        0         FINISHED
3        0         FINISHED
4        0         FINISHED
5        0         FINISHED
##########################

******* Long running program is running *******
Result: 392146832
******* Long running program is finished *******

##########################
PID      PPID      State
0        -1          READY
1        0         FINISHED
2        0         FINISHED
3        0         FINISHED
4        0         FINISHED
5        0         FINISHED
6        0         FINISHED
##########################

Init process continues...
```