Operating Systems Homework #01 Part B

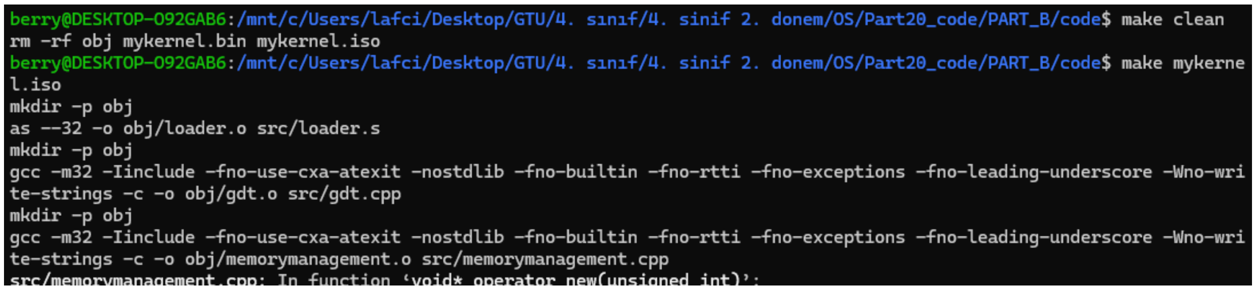
Berru Lafcı - 1901042681

Contents:

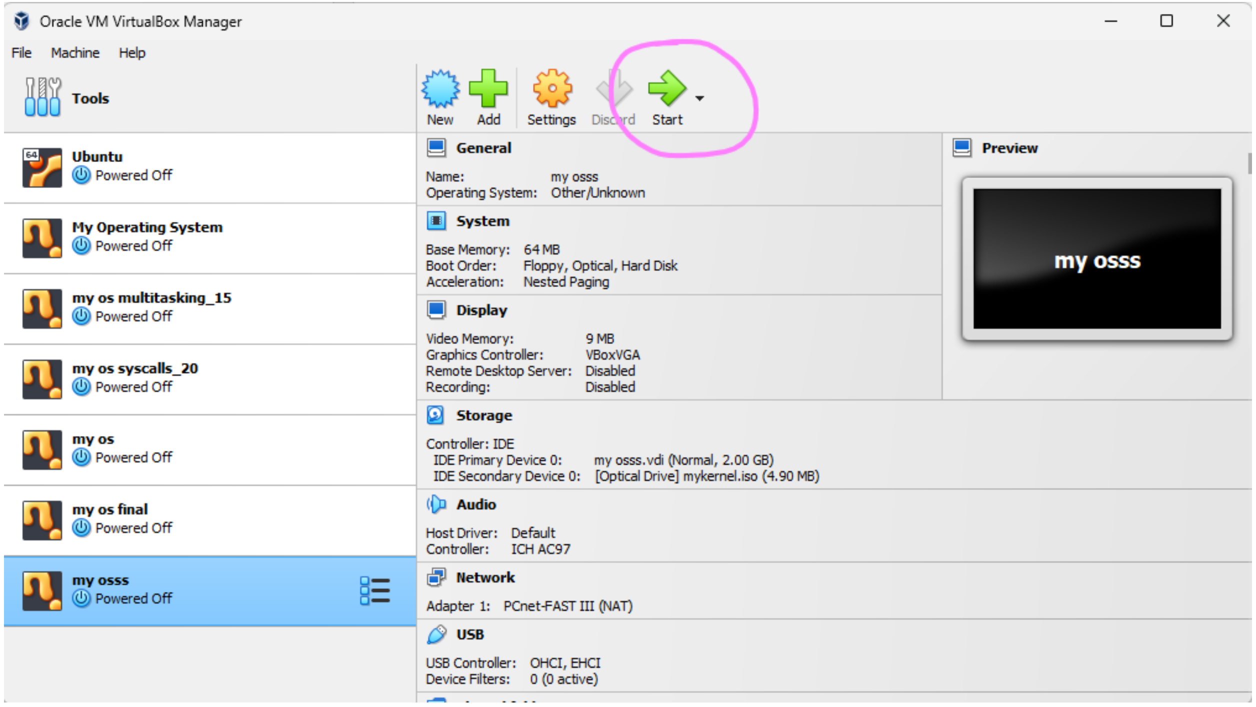
1- Compile and Run
2- Code explanations
3- Test cases and outputs

1- Compile and Run:

I compiled my code with **"make mykernel.iso"** on WSL and run it on Virtual Machine with choosing the corresponding iso file.



```
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/GTU/4. sınıf/4. sinif 2. donem/OS/Part20_code/PART_B/code$ make clean
rm -rf obj mykernel.bin mykernel.iso
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/GTU/4. sınıf/4. sinif 2. donem/OS/Part20_code/PART_B/code$ make mykerne
l.iso
mkdir -p obj
as --32 -o obj/loader.o src/loader.s
mkdir -p obj
gcc -m32 -Iinclude -fno-use-cxa-atexit -nostdlib -fno-builtin -fno-rtti -fno-exceptions -fno-leading-underscore -Wno-wri
te-strings -c -o obj/gdt.o src/gdt.cpp
mkdir -p obj
gcc -m32 -Iinclude -fno-use-cxa-atexit -nostdlib -fno-builtin -fno-rtti -fno-exceptions -fno-leading-underscore -Wno-wri
te-strings -c -o obj/memorymanagement.o src/memorymanagement.cpp
src/memorymanagement.cpp: In function 'void* operator new(unsigned int)':
```

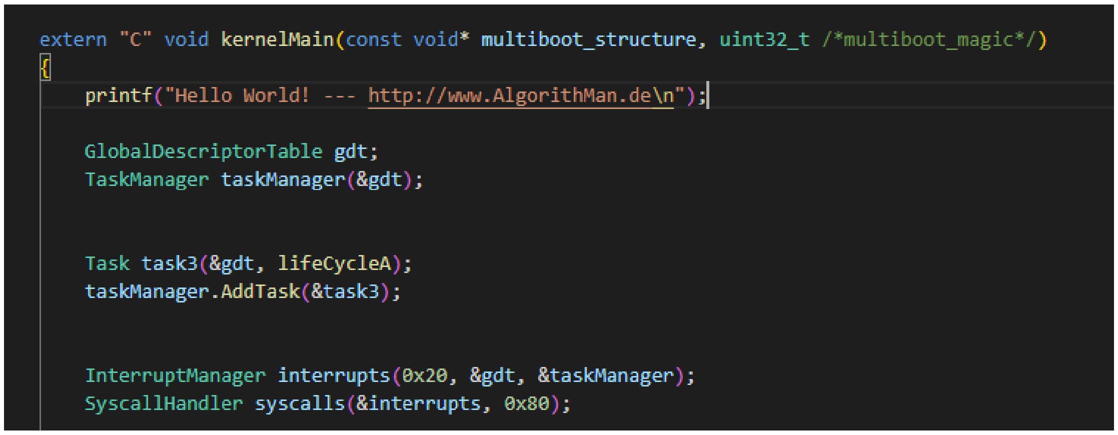


Oracle VM VirtualBox Manager

```
General
Name:              my osss
Operating System:  Other/Unknown

System
Base Memory:   64 MB
Boot Order:    Floppy, Optical, Hard Disk
Acceleration:  Nested Paging

Display
Video Memory:            9 MB
Graphics Controller:     VBoxVGA
Remote Desktop Server:   Disabled
Recording:               Disabled

Storage
Controller: IDE
  IDE Primary Device 0:    my osss.vdi (Normal, 2.00 GB)
  IDE Secondary Device 0:  [Optical Drive] mykernel.iso (4.90 MB)

Audio
Host Driver:  Default
Controller:   ICH AC97

Network
Adapter 1:  PCnet-FAST III (NAT)

USB
USB Controller:  OHCI, EHCI
Device Filters:  0 (0 active)
```

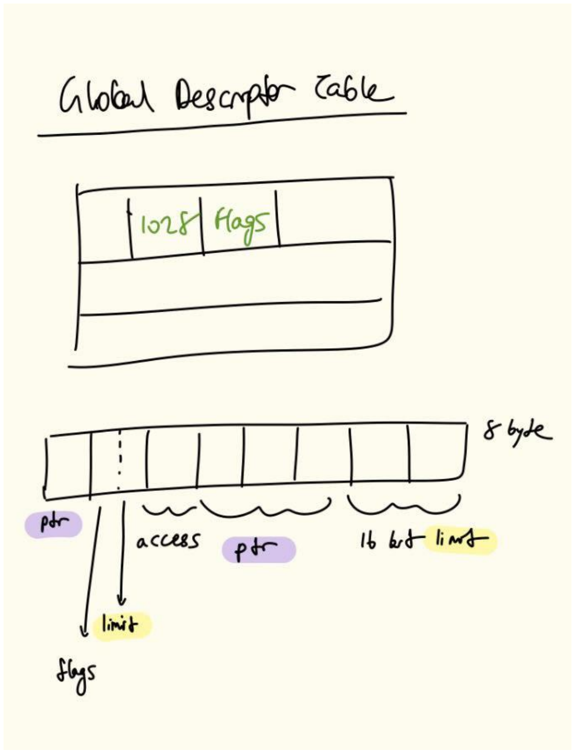
2- Code Explanations:

1- Initialization and creating Init process:



```
extern "C" void kernelMain(const void* multiboot_structure, uint32_t /*multiboot_magic*/)
{
    printf("Hello World! --- http://www.AlgorithMan.de\n");

    GlobalDescriptorTable gdt;
    TaskManager taskManager(&gdt);


    Task task3(&gdt, lifeCycleA);
    taskManager.AddTask(&task3);


    InterruptManager interrupts(0x20, &gdt, &taskManager);
    SyscallHandler syscalls(&interrupts, 0x80);
```

- **GlobalDescriptorTable gdt;**: Sets up global memory segments. It includes details about the base address, the size, and access privileges of memory segments. In Engelman videos, we created the segments sizes and places like this:

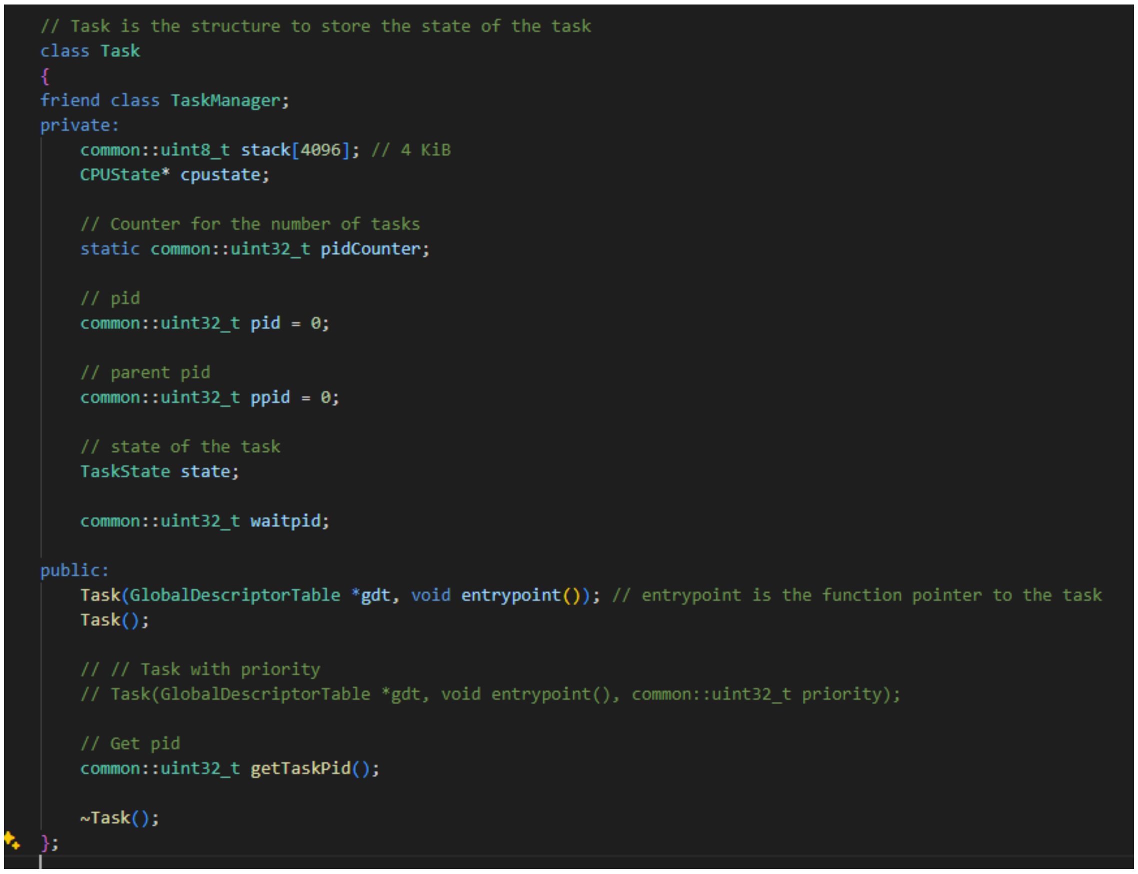


- **TaskManager taskManager(&gdt);**: Manages tasks using the GDT.
- **Task task3(&gdt, lifeCycleA);**: Creates an init process.
- **taskManager.AddTask(&task3);**: Adds the task to the task manager for scheduling.
- **InterruptManager interrupts(0x20, &gdt, &taskManager);**: Manages hardware interrupts and task switching. 0x20 is sent as the timer interrupt.
- **SyscallHandler syscalls(&interrupts, 0x80);**: Handles system calls using 0x80 interrupt number for system calls.

2- MULTITASKING FILES:

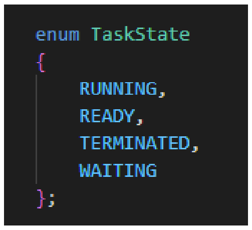
- **Task Class:**

It encapsulates the properties and behavior of a process in an operating system, including its stack, CPU state, PIDs, task state, and entrypoint function.
Tasks are managed by the TaskManager, and each task has a unique PID.



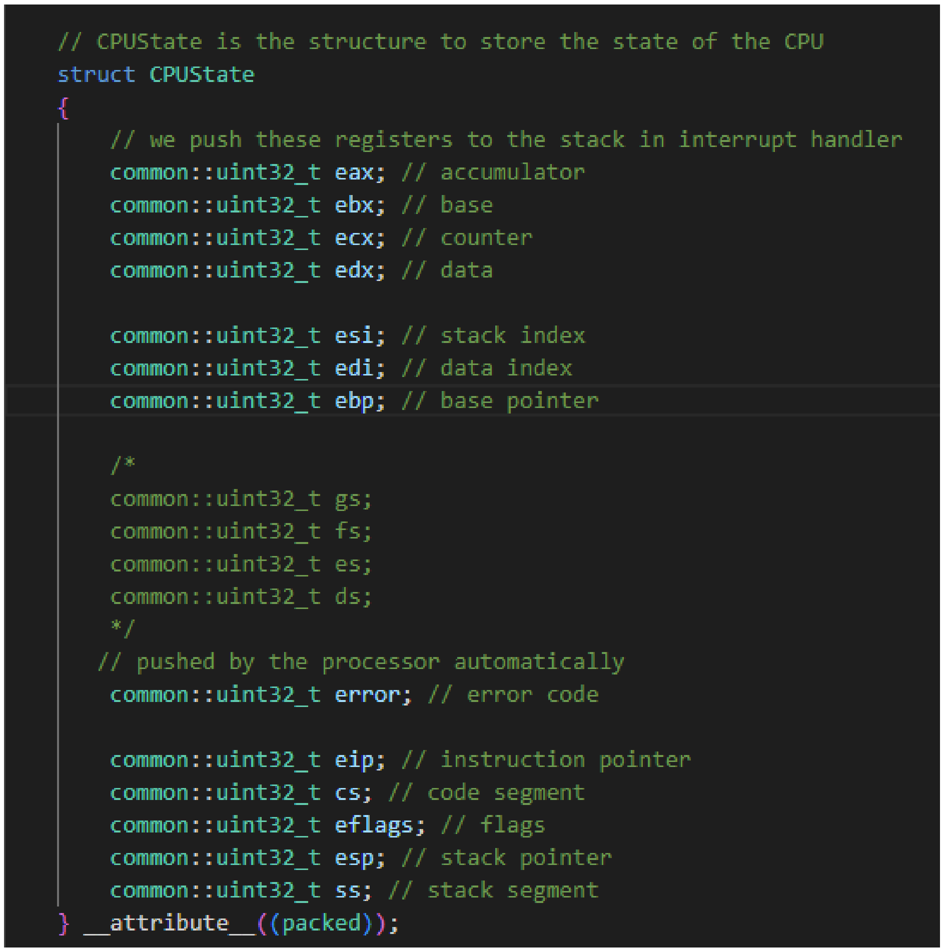
```
// Task is the structure to store the state of the task
class Task
{
friend class TaskManager;
private:
    common::uint8_t stack[4096]; // 4 KiB
    CPUState* cpustate;

    // Counter for the number of tasks
    static common::uint32_t pidCounter;

    // pid
    common::uint32_t pid = 0;

    // parent pid
    common::uint32_t ppid = 0;

    // state of the task
    TaskState state;

    common::uint32_t waitpid;

public:
    Task(GlobalDescriptorTable *gdt, void entrypoint()); // entrypoint is the function pointer to the task
    Task();

    // // Task with priority
    // Task(GlobalDescriptorTable *gdt, void entrypoint(), common::uint32_t priority);

    // Get pid
    common::uint32_t getTaskPid();

    ~Task();
};
```

The TaskState is an enum that showing its states while scheduling:



```
enum TaskState
{
    RUNNING,
    READY,
    TERMINATED,
    WAITING
};
```

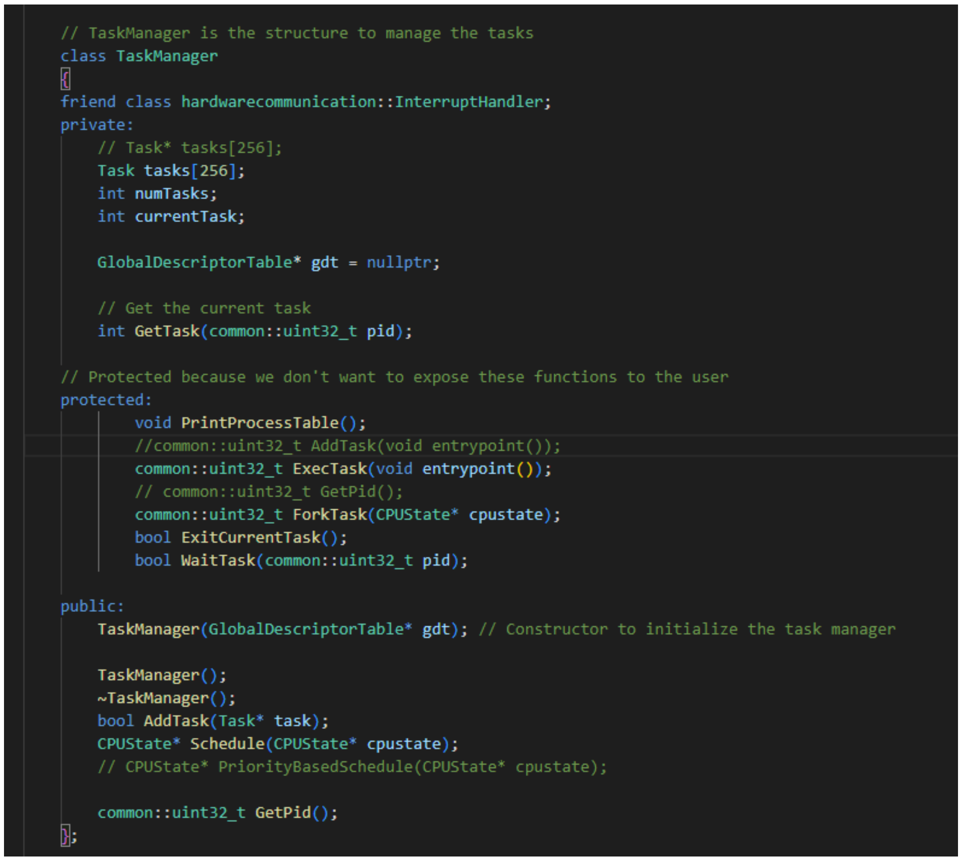
The CPUState is a struct to save the CPU state when an interrupt occurs so that the state can be restored when the interrupt handling is complete:



```
// CPUState is the structure to store the state of the CPU
struct CPUState
{
    // we push these registers to the stack in interrupt handler
    common::uint32_t eax; // accumulator
    common::uint32_t ebx; // base
    common::uint32_t ecx; // counter
    common::uint32_t edx; // data

    common::uint32_t esi; // stack index
    common::uint32_t edi; // data index
    common::uint32_t ebp; // base pointer

    /*
    common::uint32_t gs;
    common::uint32_t fs;
    common::uint32_t es;
    common::uint32_t ds;
    */
   // pushed by the processor automatically
    common::uint32_t error; // error code

    common::uint32_t eip; // instruction pointer
    common::uint32_t cs; // code segment
    common::uint32_t eflags; // flags
    common::uint32_t esp; // stack pointer
    common::uint32_t ss; // stack segment
} __attribute__((packed));
```

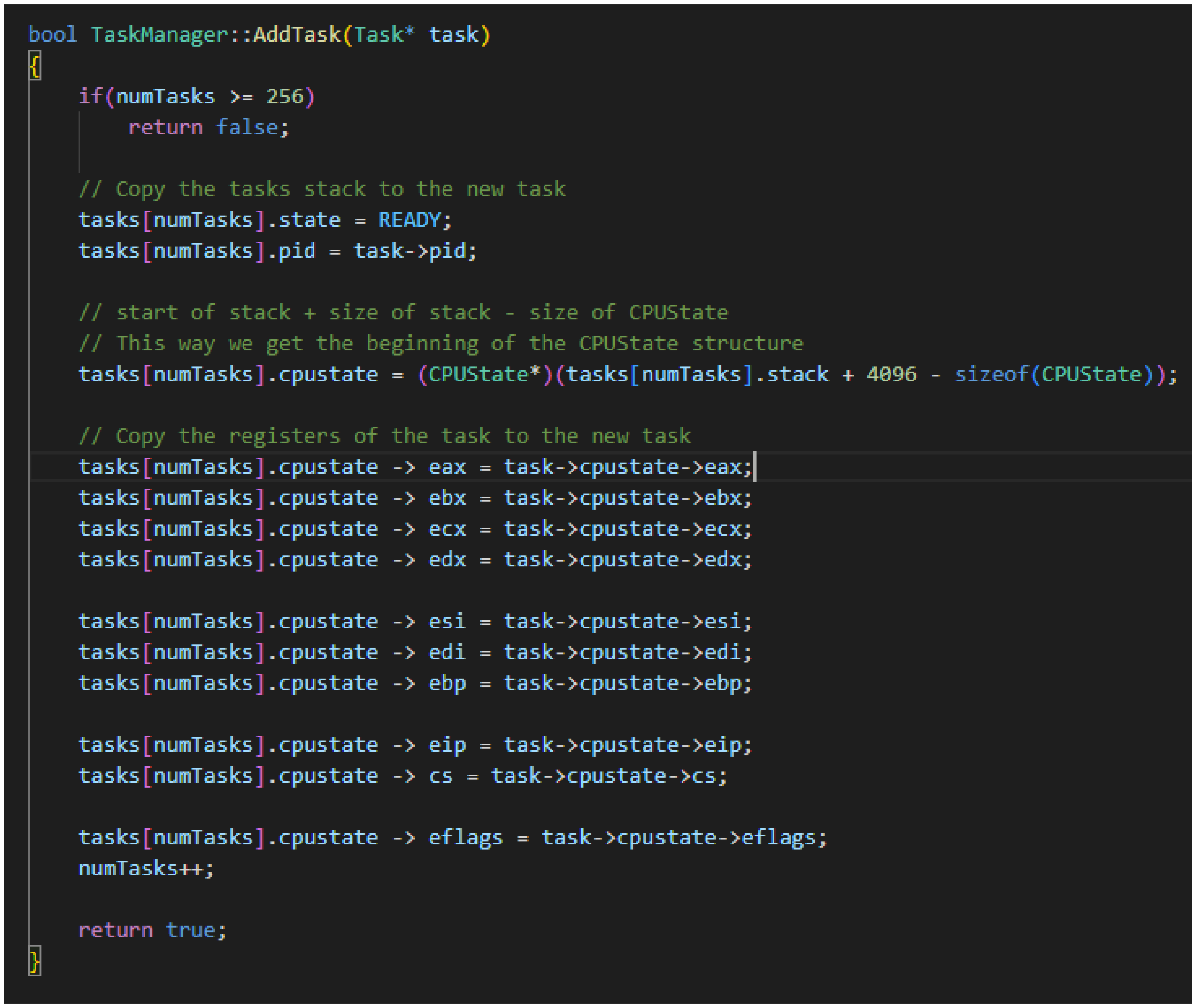
- **TaskManager Class:**

The TaskManager class manages the creation, execution, and scheduling of tasks. It uses an array to keep track of tasks and provides functions for task management such as forking, executing, and waiting for tasks.



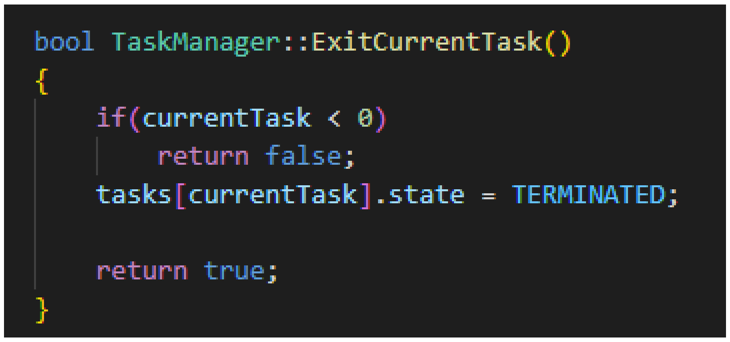
```
// TaskManager is the structure to manage the tasks
class TaskManager
{
friend class hardwarecommunication::InterruptHandler;
private:
    // Task* tasks[256];
    Task tasks[256];
    int numTasks;
    int currentTask;

    GlobalDescriptorTable* gdt = nullptr;

    // Get the current task
    int GetTask(common::uint32_t pid);

// Protected because we don't want to expose these functions to the user
protected:
        void PrintProcessTable();
        //common::uint32_t AddTask(void entrypoint());
        common::uint32_t ExecTask(void entrypoint());
        // common::uint32_t GetPid();
        common::uint32_t ForkTask(CPUState* cpustate);
        bool ExitCurrentTask();
        bool WaitTask(common::uint32_t pid);

public:
    TaskManager(GlobalDescriptorTable* gdt); // Constructor to initialize the task manager

    TaskManager();
    ~TaskManager();
    bool AddTask(Task* task);
    CPUState* Schedule(CPUState* cpustate);
    // CPUState* PriorityBasedSchedule(CPUState* cpustate);

    common::uint32_t GetPid();
};
```

- **bool TaskManager::AddTask(Task* task):** This function calls from kernelMain and creates the init process with copying registers, cpustate and stack of the task to the new task.



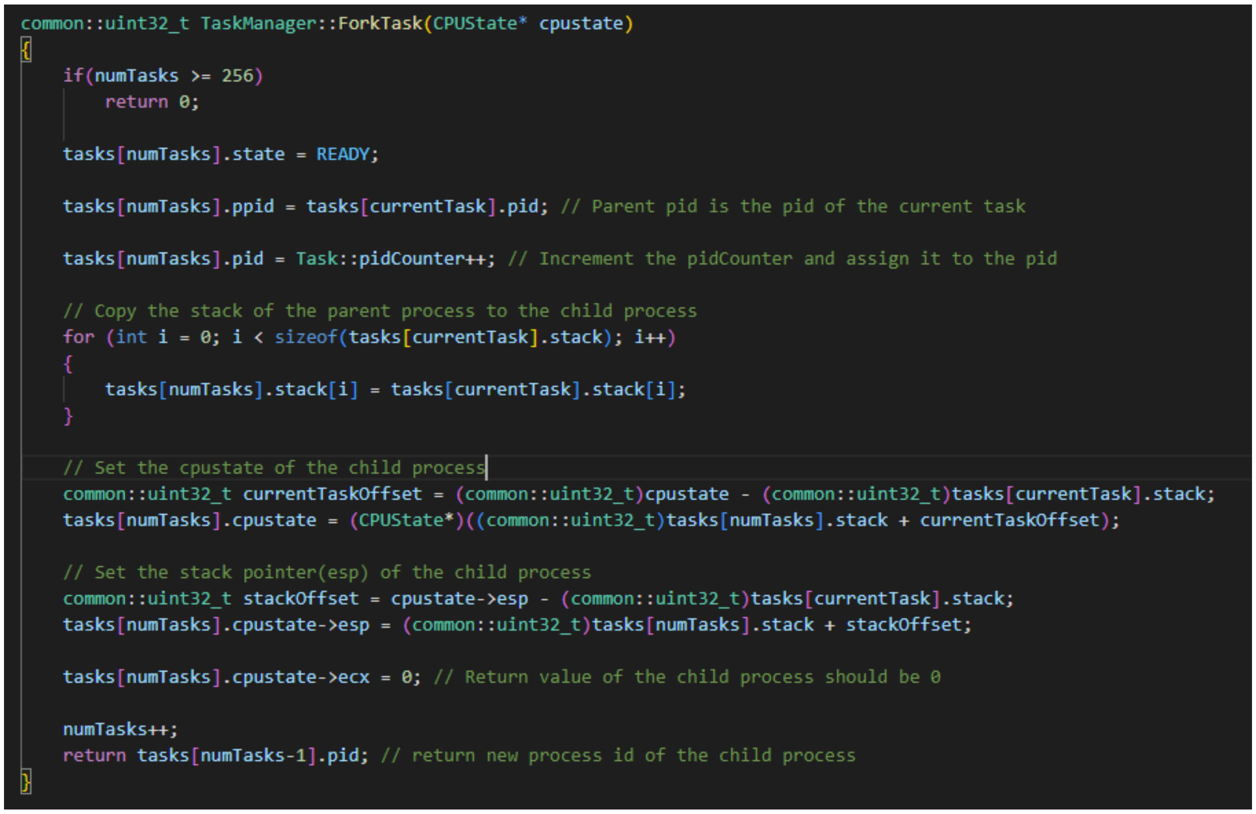
```
bool TaskManager::AddTask(Task* task)
{
    if(numTasks >= 256)
        return false;

    // Copy the tasks stack to the new task
    tasks[numTasks].state = READY;
    tasks[numTasks].pid = task->pid;

    // start of stack + size of stack - size of CPUState
    // This way we get the beginning of the CPUState structure
    tasks[numTasks].cpustate = (CPUState*)(tasks[numTasks].stack + 4096 - sizeof(CPUState));

    // Copy the registers of the task to the new task
    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;
}
```

- **bool TaskManager::ExitCurrentTask():** When we exit from the current task, we make the state as TERMINATED.



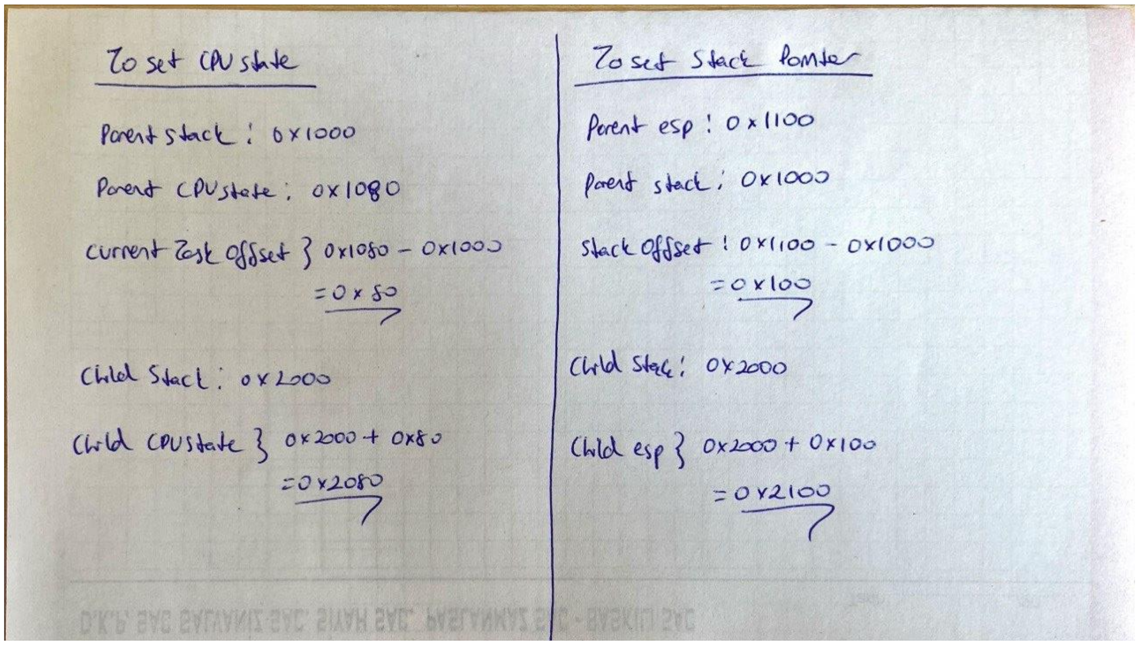
```
bool TaskManager::ExitCurrentTask()
{
    if(currentTask < 0)
        return false;
    tasks[currentTask].state = TERMINATED;

    return true;
}
```

- **common::uint32_t TaskManager::ForkTask(CPUState* cpustate):**
    - The function first checks if the maximum number of tasks (256) has been reached. If it has, the function returns 0, indicating that no new task can be created.
    - The new task's state is set to READY.
    - The parent process ID (ppid) of the new task is set to the process ID (pid) of the current task.
    - The pidCounter incremented value is assigned to the new task.
    - The entire stack of the current task is copied to the new task. This ensures that the child has the same stack contents as the parent task at the time of forking.
    - The offset of the CPUState within the current task's stack is calculated.
    - The CPU state of the new task is set to the corresponding location within the new task's stack. This effectively copies the CPU state from the parent task to the child.
    - The stack pointer (esp) of the new task is set based on the stack pointer of the current task. This ensures that the new task's stack pointer points to the true location within its own stack.
    - The ecx register of the new task is set to 0. This is the behavior of fork(), where the child process receives a return value of 0.
    - The total number of tasks (numTasks) has increased as we created a new process with fork.
    - The function returns the process ID of the child now.



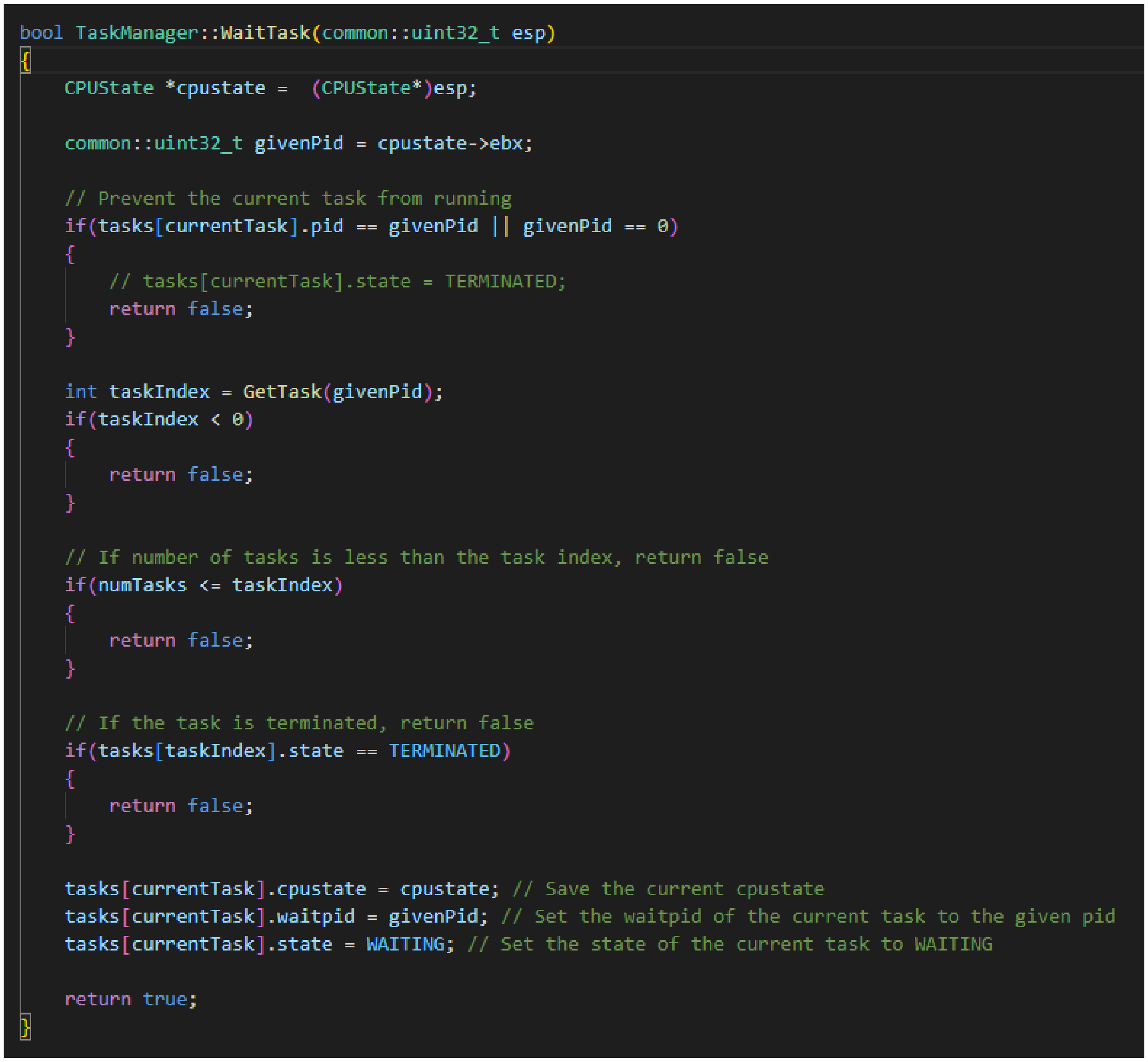
```
common::uint32_t TaskManager::ForkTask(CPUState* cpustate)
{
    if(numTasks >= 256)
        return 0;

    tasks[numTasks].state = READY;

    tasks[numTasks].ppid = tasks[currentTask].pid; // Parent pid is the pid of the current task

    tasks[numTasks].pid = Task::pidCounter++; // Increment the pidCounter and assign it to the pid

    // Copy the stack of the parent process to the child process
    for (int i = 0; i < sizeof(tasks[currentTask].stack); i++)
    {
        tasks[numTasks].stack[i] = tasks[currentTask].stack[i];
    }

    // Set the cpustate of the child process
    common::uint32_t currentTaskOffset = (common::uint32_t)cpustate - (common::uint32_t)tasks[currentTask].stack;
    tasks[numTasks].cpustate = (CPUState*)((common::uint32_t)tasks[numTasks].stack + currentTaskOffset);

    // Set the stack pointer(esp) of the child process
    common::uint32_t stackOffset = cpustate->esp - (common::uint32_t)tasks[currentTask].stack;
    tasks[numTasks].cpustate->esp = (common::uint32_t)tasks[numTasks].stack + stackOffset;

    tasks[numTasks].cpustate->ecx = 0; // Return value of the child process should be 0

    numTasks++;
    return tasks[numTasks-1].pid; // return new process id of the child process
}
```

To describe better here is an example for setting the cpustate and esp of the child. This way their places in memory calculated smoothly:



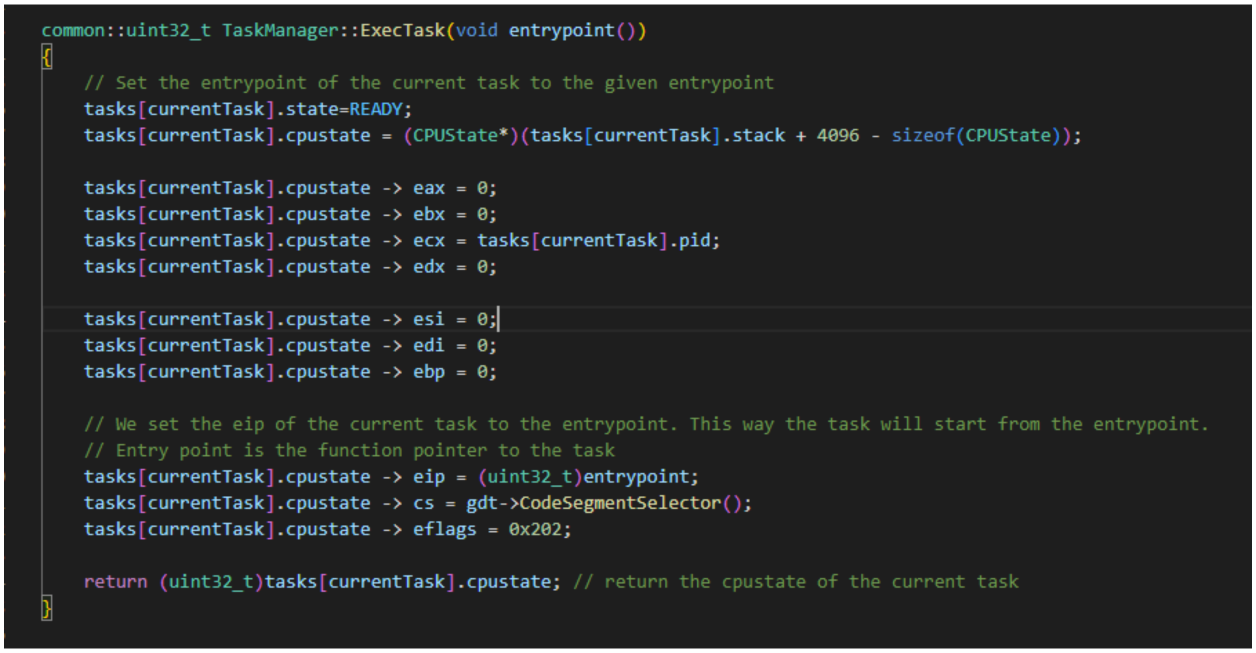
To set CPU state	To set Stack Pointer
Parent stack : 0x1000	Parent esp : 0x1100
Parent CPUstate : 0x1080	Parent stack : 0x1000
current Task offset } 0x1080 − 0x1000 = 0x80	Stack Offset : 0x1100 − 0x1000 = 0x100
Child Stack : 0x2000	Child Stack : 0x2000
Child CPUState } 0x2000 + 0x80 = 0x2080	Child esp } 0x2000 + 0x100 = 0x2100

- **bool TaskManager::WaitTask(common::uint32_t esp):**
    - The ebx register of the current task's CPU state contains the process ID (givenPid) of the task that the current task wants to wait for.
    - The function checks if the current task is trying to wait on itself or if the givenPid is 0. If either condition is true, the function returns false, indicating that the wait operation is not valid.
    - The function saves the current CPU state of the task.
    - It sets the waitpid of the current task to givenPid, indicating which task it is waiting for.
    - The state of the current task is set to WAITING, meaning it will not be scheduled for execution until the specified task terminates.



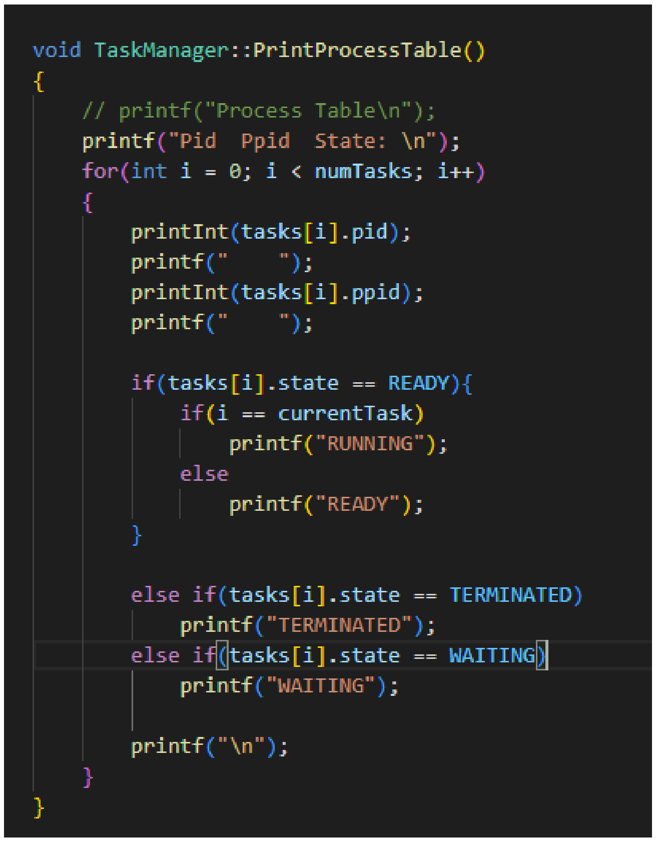
```
bool TaskManager::WaitTask(common::uint32_t esp)
{
    CPUState *cpustate =  (CPUState*)esp;

    common::uint32_t givenPid = cpustate->ebx;

    // Prevent the current task from running
    if(tasks[currentTask].pid == givenPid || givenPid == 0)
    {
        // tasks[currentTask].state = TERMINATED;
        return false;
    }

    int taskIndex = GetTask(givenPid);
    if(taskIndex < 0)
    {
        return false;
    }

    // If number of tasks is less than the task index, return false
    if(numTasks <= taskIndex)
    {
        return false;
    }

    // If the task is terminated, return false
    if(tasks[taskIndex].state == TERMINATED)
    {
        return false;
    }

    tasks[currentTask].cpustate = cpustate; // Save the current cpustate
    tasks[currentTask].waitpid = givenPid; // Set the waitpid of the current task to the given pid
    tasks[currentTask].state = WAITING; // Set the state of the current task to WAITING

    return true;
}
```

- **common::uint32_t TaskManager::ExecTask(void entrypoint()):**
    - The function sets the state of the current task to READY, ensuring that the task is marked as ready to run.
    - The CPU registers are initialized as we are creating a new process with exec.
    - The eip register (instruction pointer) is set to the address of the entry point function. This ensures that the task starts executing from this given function as entry point.
    - The cs register (code segment) is set to the code segment selector from the GDT (Global Descriptor Table).
    - The eflags register is set to 0x202, which represents the default flags for kernel mode. This setup includes enabling interrupts.



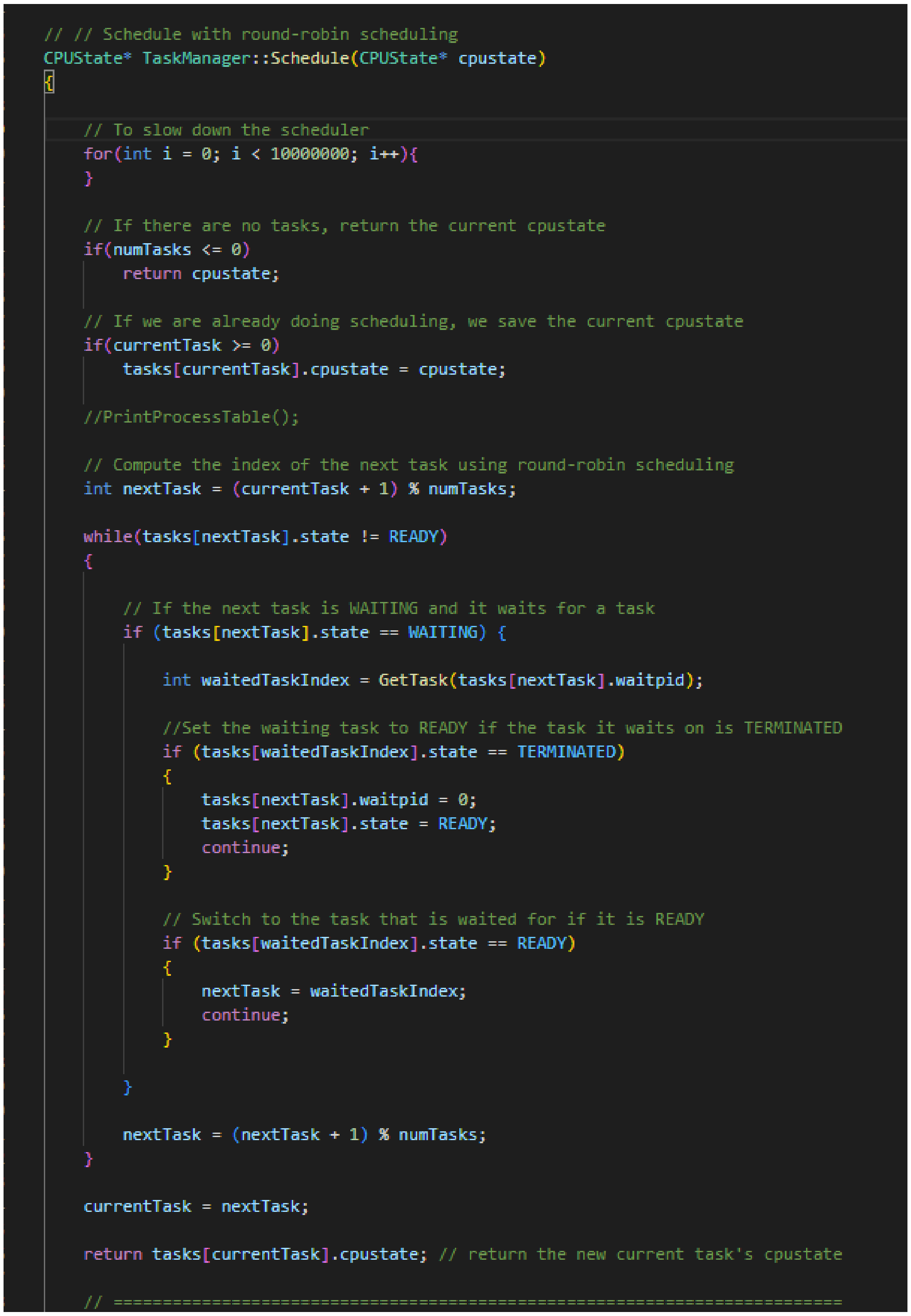
```
common::uint32_t TaskManager::ExecTask(void entrypoint())
{
    // Set the entrypoint of the current task to the given entrypoint
    tasks[currentTask].state=READY;
    tasks[currentTask].cpustate = (CPUState*)(tasks[currentTask].stack + 4096 - sizeof(CPUState));

    tasks[currentTask].cpustate -> eax = 0;
    tasks[currentTask].cpustate -> ebx = 0;
    tasks[currentTask].cpustate -> ecx = tasks[currentTask].pid;
    tasks[currentTask].cpustate -> edx = 0;

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

    // We set the eip of the current task to the entrypoint. This way the task will start from the entrypoint.
    // Entry point is the function pointer to the task
    tasks[currentTask].cpustate -> eip = (uint32_t)entrypoint;
    tasks[currentTask].cpustate -> cs = gdt->CodeSegmentSelector();
    tasks[currentTask].cpustate -> eflags = 0x202;

    return (uint32_t)tasks[currentTask].cpustate; // return the cpustate of the current task
}
```

- **void TaskManager::PrintProcessTable():** Process table prints the pid, ppid and task state of the tasks.



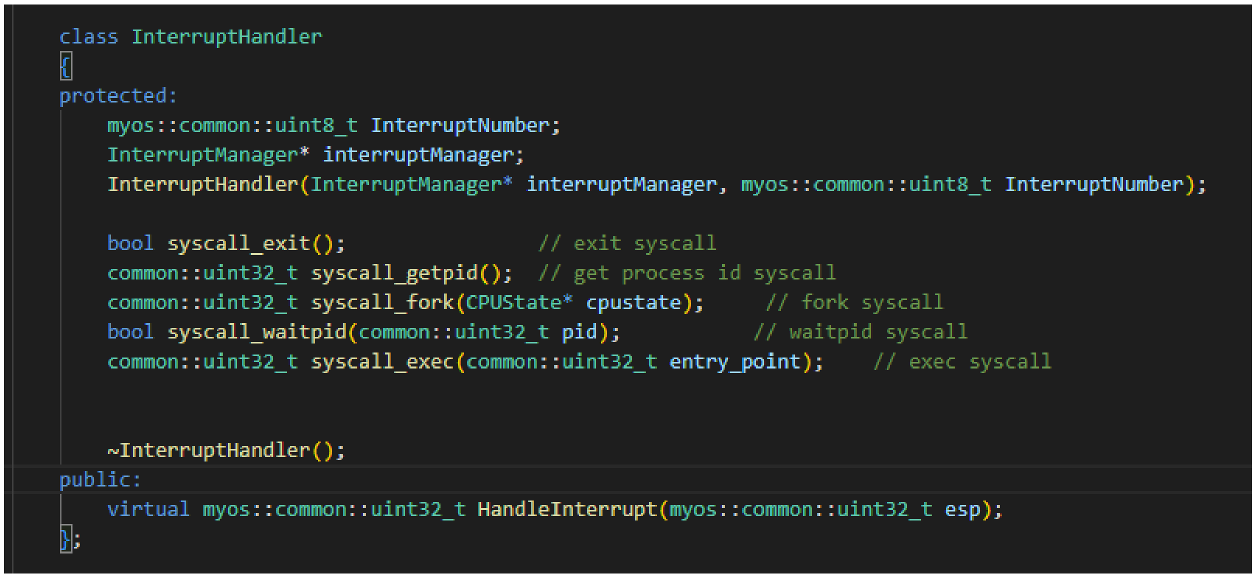
```
void TaskManager::PrintProcessTable()
{
    // printf("Process Table\n");
    printf("Pid  Ppid  State: \n");
    for(int i = 0; i < numTasks; i++)
    {
        printInt(tasks[i].pid);
        printf("    ");
        printInt(tasks[i].ppid);
        printf("    ");

        if(tasks[i].state == READY){
            if(i == currentTask)
                printf("RUNNING");
            else
                printf("READY");
        }

        else if(tasks[i].state == TERMINATED)
            printf("TERMINATED");
        else if(tasks[i].state == WAITING)
            printf("WAITING");

        printf("\n");
    }
}
```

- **CPUState* TaskManager::Schedule(CPUState* cpustate):**
    - This loop slows down the scheduler. This way we can observe the scheduling with timer interrupt.
    - The function enters a loop to find the next task that is in the READY state.
    - If the next task is in the WAITING state, it checks the state of the task it is waiting for (tasks[nextTask].waitpid).
    - If the task being waited on has terminated, the waiting task's waitpid is reset to 0, and its state is set to READY.
    - If the task being waited on is READY, the scheduler switches to that task.
    - If neither condition is met, the scheduler moves to the next task in the list.
    - Once a task in the READY state is found, the currentTask index is updated to nextTask. Then it returns the CPU state of the new current task to restore the task contents after.

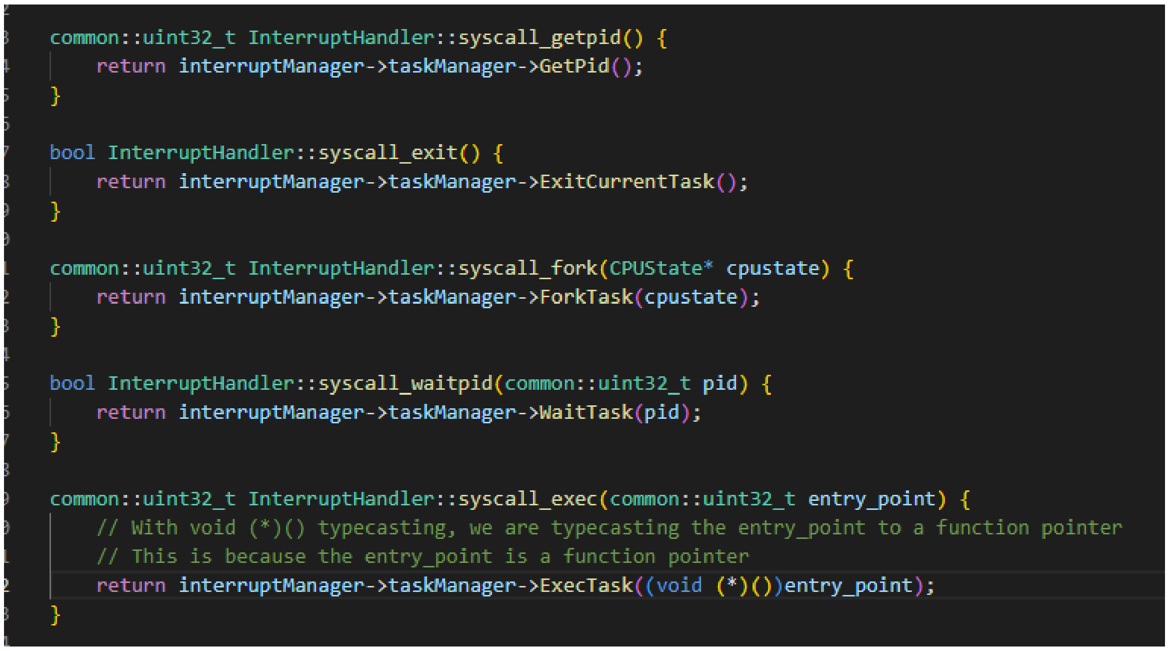


```
// // Schedule with round-robin scheduling
CPUState* TaskManager::Schedule(CPUState* cpustate)
{

    // To slow down the scheduler
    for(int i = 0; i < 10000000; i++){
    }

    // If there are no tasks, return the current cpustate
    if(numTasks <= 0)
        return cpustate;

    // If we are already doing scheduling, we save the current cpustate
    if(currentTask >= 0)
        tasks[currentTask].cpustate = cpustate;

    //PrintProcessTable();

    // Compute the index of the next task using round-robin scheduling
    int nextTask = (currentTask + 1) % numTasks;

    while(tasks[nextTask].state != READY)
    {

        // If the next task is WAITING and it waits for a task
        if (tasks[nextTask].state == WAITING) {

            int waitedTaskIndex = GetTask(tasks[nextTask].waitpid);

            //Set the waiting task to READY if the task it waits on is TERMINATED
            if (tasks[waitedTaskIndex].state == TERMINATED)
            {
                tasks[nextTask].waitpid = 0;
                tasks[nextTask].state = READY;
                continue;
            }

            // Switch to the task that is waited for if it is READY
            if (tasks[waitedTaskIndex].state == READY)
            {
                nextTask = waitedTaskIndex;
                continue;
            }

        }

        nextTask = (nextTask + 1) % numTasks;
    }

    currentTask = nextTask;

    return tasks[currentTask].cpustate; // return the new current task's cpustate

    // ==========================================================================
```

3- INTERRUPTS FILES



```cpp
class InterruptHandler
{
protected:
    myos::common::uint8_t InterruptNumber;
    InterruptManager* interruptManager;
    InterruptHandler(InterruptManager* interruptManager, myos::common::uint8_t InterruptNumber);

    bool syscall_exit();                // exit syscall
    common::uint32_t syscall_getpid();  // get process id syscall
    common::uint32_t syscall_fork(CPUState* cpustate);     // fork syscall
    bool syscall_waitpid(common::uint32_t pid);            // waitpid syscall
    common::uint32_t syscall_exec(common::uint32_t entry_point);    // exec syscall


    ~InterruptHandler();
public:
    virtual myos::common::uint32_t HandleInterrupt(myos::common::uint32_t esp);
};
```



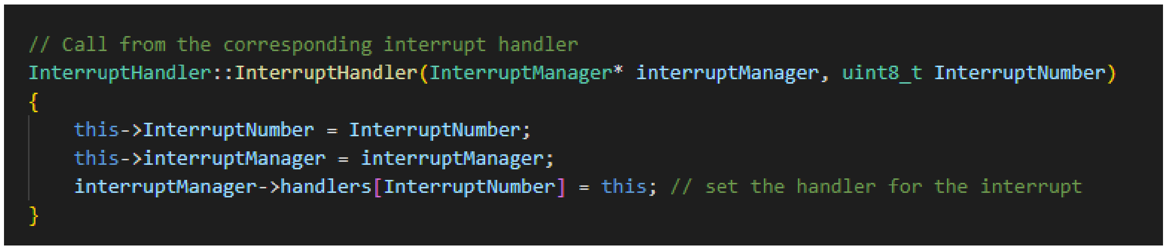
```cpp
common::uint32_t InterruptHandler::syscall_getpid() {
    return interruptManager->taskManager->GetPid();
}

bool InterruptHandler::syscall_exit() {
    return interruptManager->taskManager->ExitCurrentTask();
}

common::uint32_t InterruptHandler::syscall_fork(CPUState* cpustate) {
    return interruptManager->taskManager->ForkTask(cpustate);
}

bool InterruptHandler::syscall_waitpid(common::uint32_t pid) {
    return interruptManager->taskManager->WaitTask(pid);
}

common::uint32_t InterruptHandler::syscall_exec(common::uint32_t entry_point) {
    // With void (*)() typecasting, we are typecasting the entry_point to a function pointer
    // This is because the entry_point is a function pointer
    return interruptManager->taskManager->ExecTask((void (*)())entry_point);
}
```

1. A system call or hardware interrupt occurs, triggering the **InterruptManager**.

2. The **InterruptManager** identifies the specific interrupt and delegates it to the appropriate **InterruptHandler**.

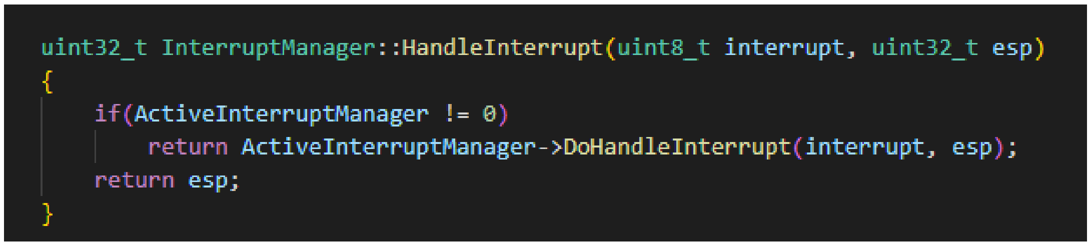
3. The **InterruptHandler** processes the system call by invoking the corresponding method in the **TaskManager**.
4. The **TaskManager** executes the requested task management operation, such as getpid, exit, fork etc.

This part of the code sets the corresponding interrupt handler:

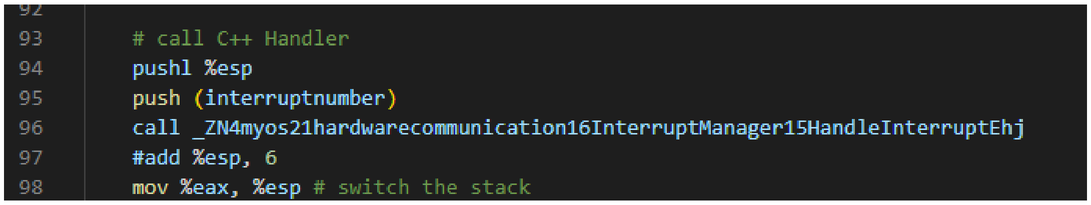


```
// Call from the corresponding interrupt handler
InterruptHandler::InterruptHandler(InterruptManager* interruptManager, uint8_t InterruptNumber)
{
    this->InterruptNumber = InterruptNumber;
    this->interruptManager = interruptManager;
    interruptManager->handlers[InterruptNumber] = this; // set the handler for the interrupt
}
```

This HandleInterrupt calls from interruptstubs.s. The interrupt transfers from C++ to ASM world. We must write the handler in assembly because we shouldn't jump a function in C++. Because C++ compiler may put the RAM somewhere irrelevant, and we can't guarantee that in C++.



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

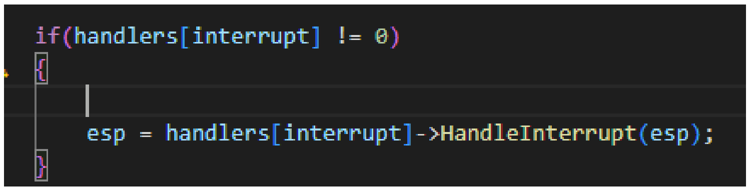


```
    # call C++ Handler
    pushl %esp
    push (interruptnumber)
    call _ZN4myos21hardwarecommunication16InterruptManager15HandleInterruptEhj
    #add %esp, 6
    mov %eax, %esp # switch the stack
```

To answer the interrupt, we must send data back through the command and data ports (members of interrupt manager). This way we can send an answer back to the interrupt to not halt the system.

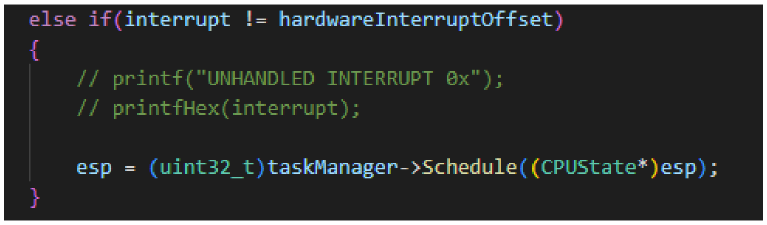
uint32_t InterruptManager::DoHandleInterrupt(uint8_t interrupt, uint32_t esp):

- If the interrupt number is 0x80, then it is a syscall. So, we need to call the syscall handler. Esp comes from the CPUState struct. If not, then it is a hardware interrupt. So, we need to call the hardware interrupt handler.



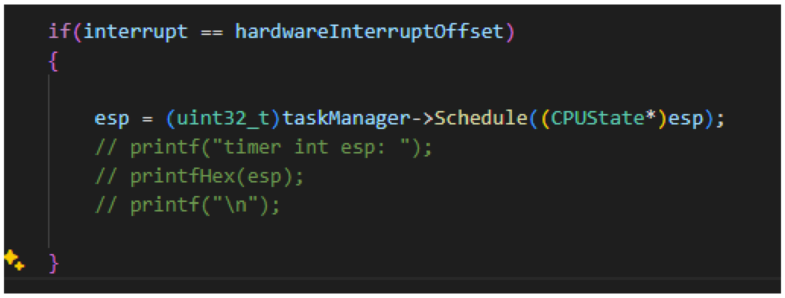
```
if(handlers[interrupt] != 0)
{

    esp = handlers[interrupt]->HandleInterrupt(esp);
}
```

- If it is not timer interrupt, we should keep going the scheduling.



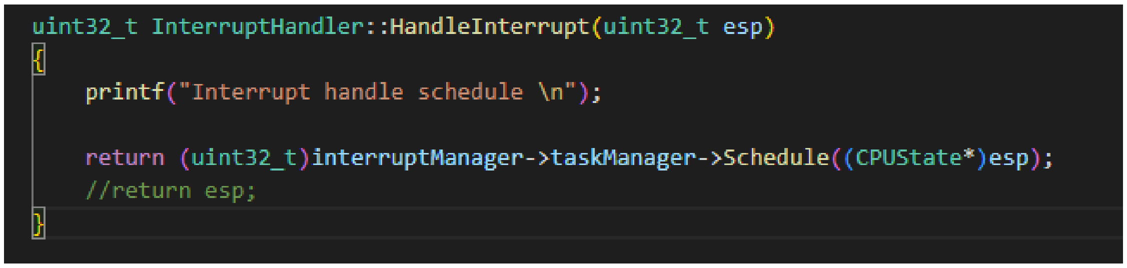
```
else if(interrupt != hardwareInterruptOffset)
{
    // printf("UNHANDLED INTERRUPT 0x");
    // printfHex(interrupt);

    esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
}
```

- If it is timer interrupt, we should make scheduling.



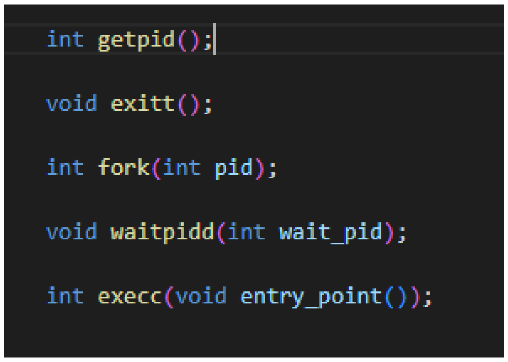
```
if(interrupt == hardwareInterruptOffset)
{

    esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
    // printf("timer int esp: ");
    // printfHex(esp);
    // printf("\n");

}
```

So when there is a timer interrupt, we call this code block to schedule again:



```
uint32_t InterruptHandler::HandleInterrupt(uint32_t esp)
{
    printf("Interrupt handle schedule \n");

    return (uint32_t)interruptManager->taskManager->Schedule((CPUState*)esp);
    //return esp;
}
```

4- SYSCALL FILES:



```
int getpid();

void exitt();

int fork(int pid);

void waitpidd(int wait_pid);

int execc(void entry_point());
```

int $0x80 triggers a software interrupt, invoking the syscall handler. In kernel side, we attach an interrupt handler to handle 0x80 interrupt. Then interrupt handler looks at eax, ebx, ecx etc. registers and requests the corresponding syscall.

Getpid():

- The value 1 in register eax indicates the getpid syscall.
- The value of the ecx register will be stored in the variable pid after the interrupt call is completed.

Exit():

- The value 0 in eax indicates the exitt syscall.

Fork(int pid):

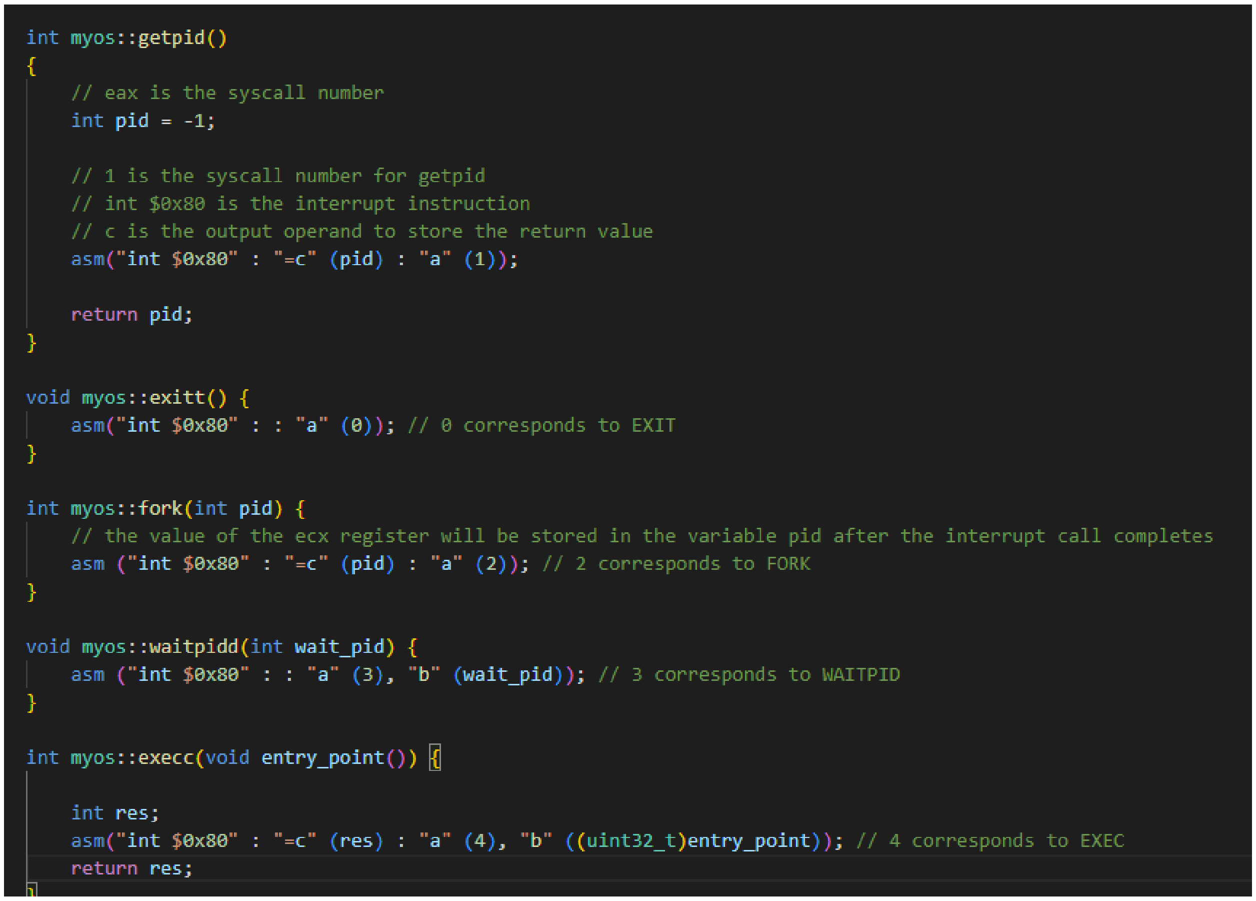
- The value 2 in eax indicates the fork syscall.
- The value of the ecx register will be stored in the variable pid after the interrupt call is completed.

Waitpidd(int wait_pid):

- The value 3 in eax indicates the waitpidd syscall.
- The wait_pid parameter is passed in the ebx register.

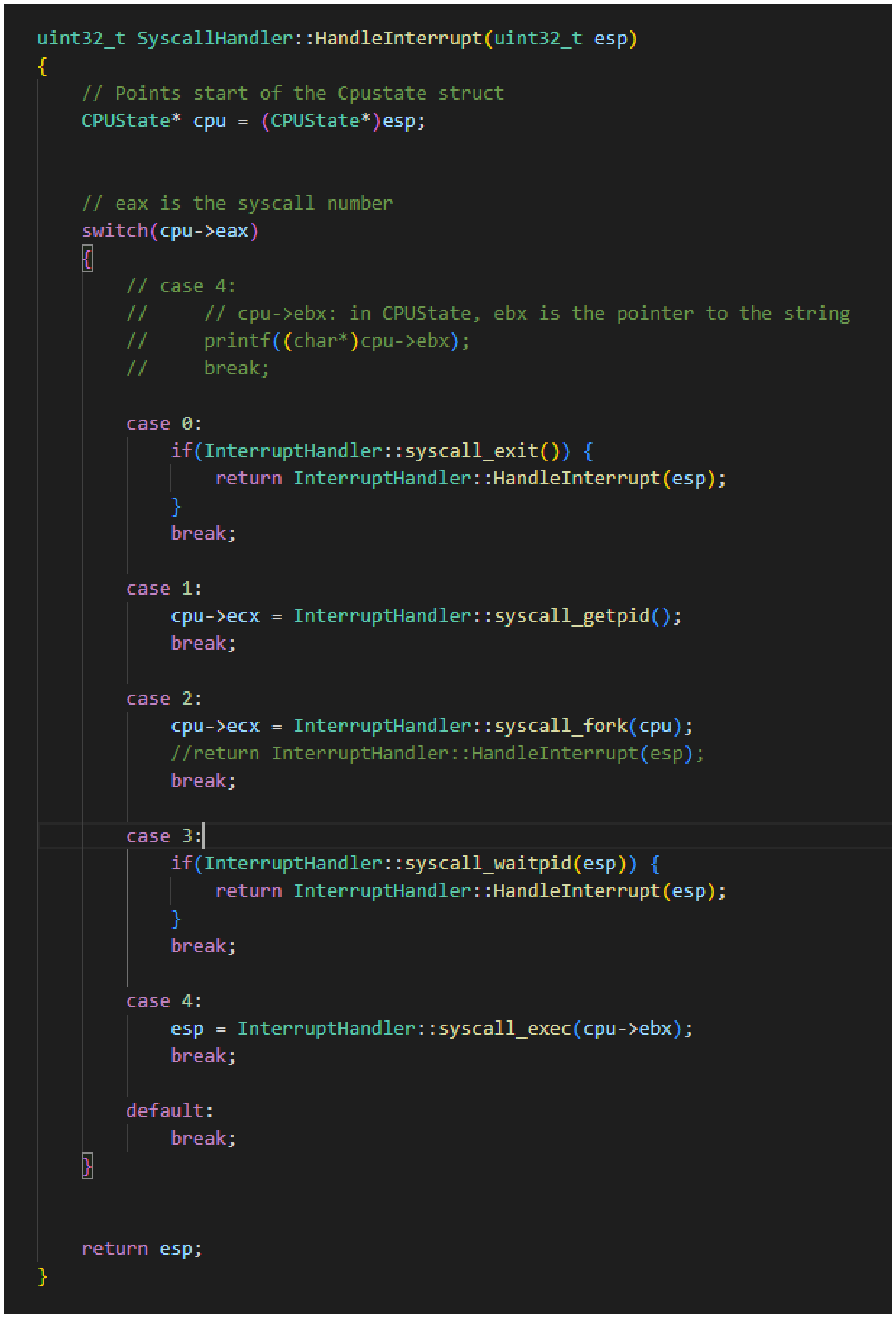
Exec(void entry_point):

- The value 4 in eax indicates the execc syscall.
- The entry point of the new program is passed in ebx.
- The value of the ecx register will be stored in the variable pid after the interrupt call is completed.



```
int myos::getpid()
{
    // eax is the syscall number
    int pid = -1;

    // 1 is the syscall number for getpid
    // int $0x80 is the interrupt instruction
    // c is the output operand to store the return value
    asm("int $0x80" : "=c" (pid) : "a" (1));

    return pid;
}

void myos::exitt() {
    asm("int $0x80" : : "a" (0)); // 0 corresponds to EXIT
}

int myos::fork(int pid) {
    // the value of the ecx register will be stored in the variable pid after the interrupt call completes
    asm ("int $0x80" : "=c" (pid) : "a" (2)); // 2 corresponds to FORK
}

void myos::waitpidd(int wait_pid) {
    asm ("int $0x80" : : "a" (3), "b" (wait_pid)); // 3 corresponds to WAITPID
}

int myos::execc(void entry_point()) {

    int res;
    asm("int $0x80" : "=c" (res) : "a" (4), "b" ((uint32_t)entry_point)); // 4 corresponds to EXEC
    return res;
}
```

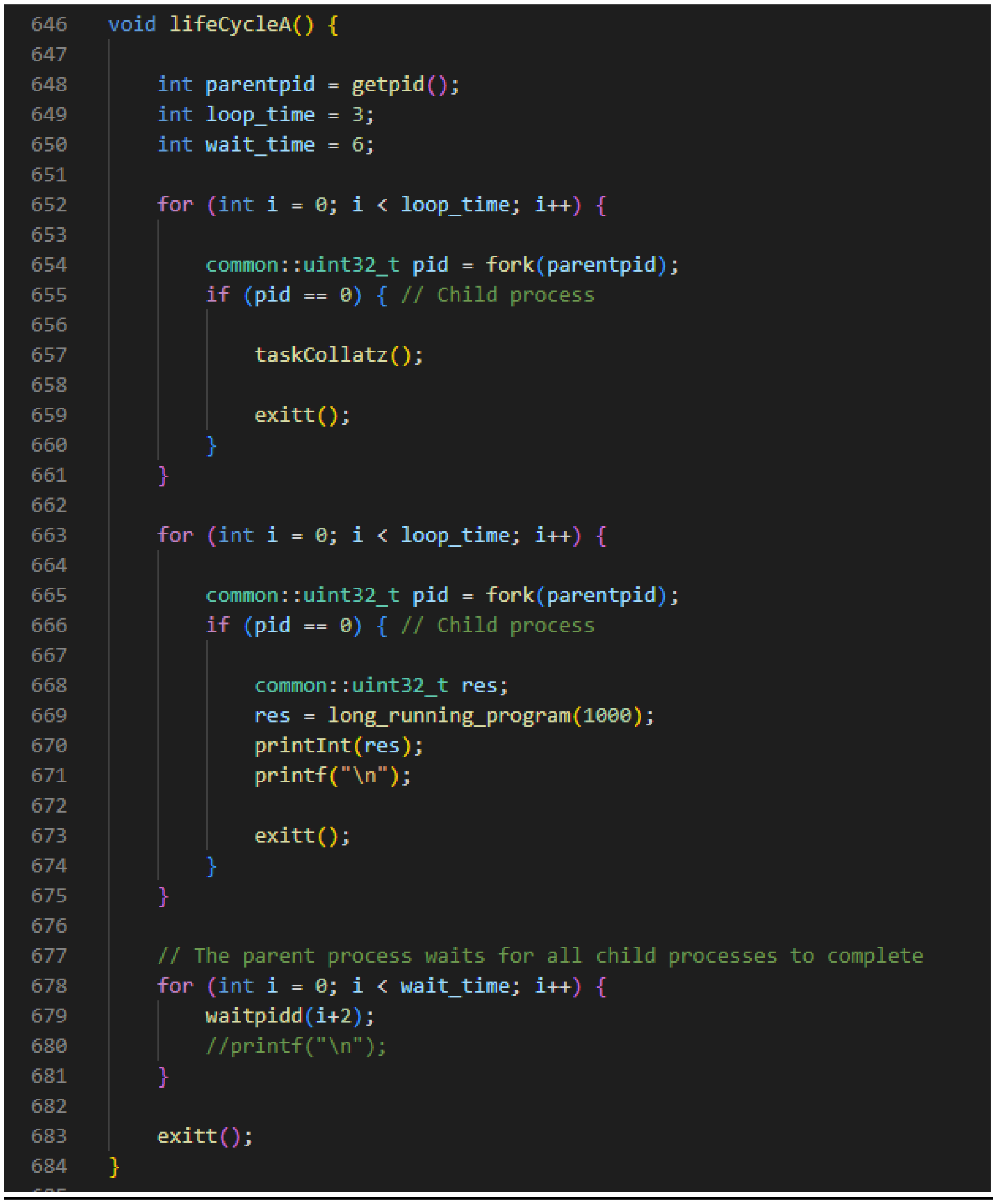
Handles system calls based on the value in eax. Then they go their corresponding functions in interrupts.cpp as I shown in above.



```
uint32_t SyscallHandler::HandleInterrupt(uint32_t esp)
{
    // Points start of the Cpustate struct
    CPUState* cpu = (CPUState*)esp;


    // eax is the syscall number
    switch(cpu->eax)
    {
        // case 4:
        //     // cpu->ebx: in CPUState, ebx is the pointer to the string
        //     printf((char*)cpu->ebx);
        //     break;

        case 0:
            if(InterruptHandler::syscall_exit()) {
                return InterruptHandler::HandleInterrupt(esp);
            }
            break;

        case 1:
            cpu->ecx = InterruptHandler::syscall_getpid();
            break;

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

        case 3:
            if(InterruptHandler::syscall_waitpid(esp)) {
                return InterruptHandler::HandleInterrupt(esp);
            }
            break;

        case 4:
            esp = InterruptHandler::syscall_exec(cpu->ebx);
            break;

        default:
            break;
    }


    return esp;
}
```

3- Test Cases and Outputs

I loaded the taskCollatz 3 times and long running program 3 times with fork. This way all the processes created by fork system call as written in the pdf.
Then I wait for all of them to terminate. I also exit from the parent process because of to see the final process table. If I don't put an exit, the parent process stuck on the RUNNING (as it should be) and go to infinite.

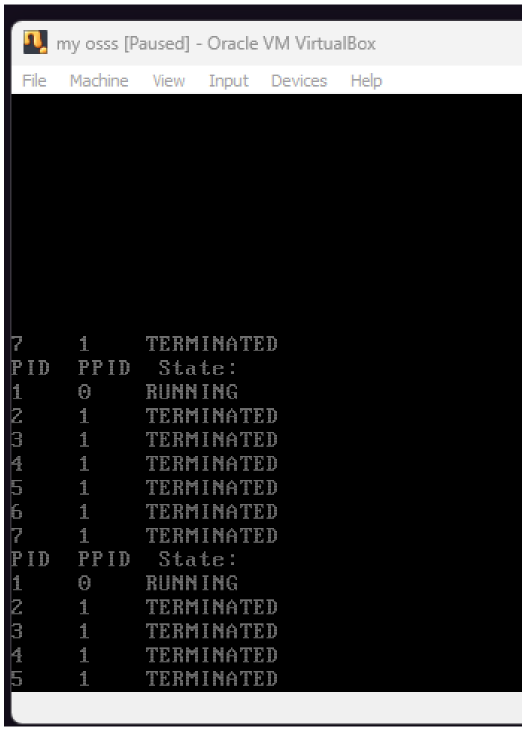


```
void lifeCycleA() {

    int parentpid = getpid();
    int loop_time = 3;
    int wait_time = 6;

    for (int i = 0; i < loop_time; i++) {

        common::uint32_t pid = fork(parentpid);
        if (pid == 0) { // Child process

            taskCollatz();

            exitt();
        }
    }

    for (int i = 0; i < loop_time; i++) {

        common::uint32_t pid = fork(parentpid);
        if (pid == 0) { // Child process

            common::uint32_t res;
            res = long_running_program(1000);
            printInt(res);
            printf("\n");

            exitt();
        }
    }

    // The parent process waits for all child processes to complete
    for (int i = 0; i < wait_time; i++) {
        waitpidd(i+2);
        //printf("\n");
    }

    exitt();
}
```

Outputs:

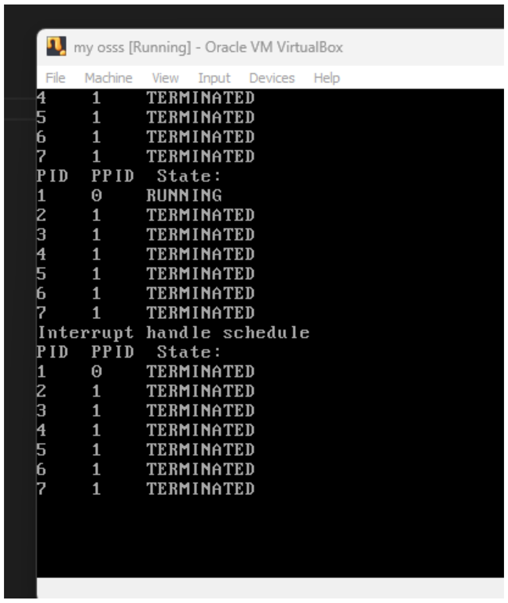
My parent process id is 1 and the init process id is 0. The rest of them are child processes.

Parent without exit:


my osss [Paused] - Oracle VM VirtualBox
File Machine View Input Devices Help

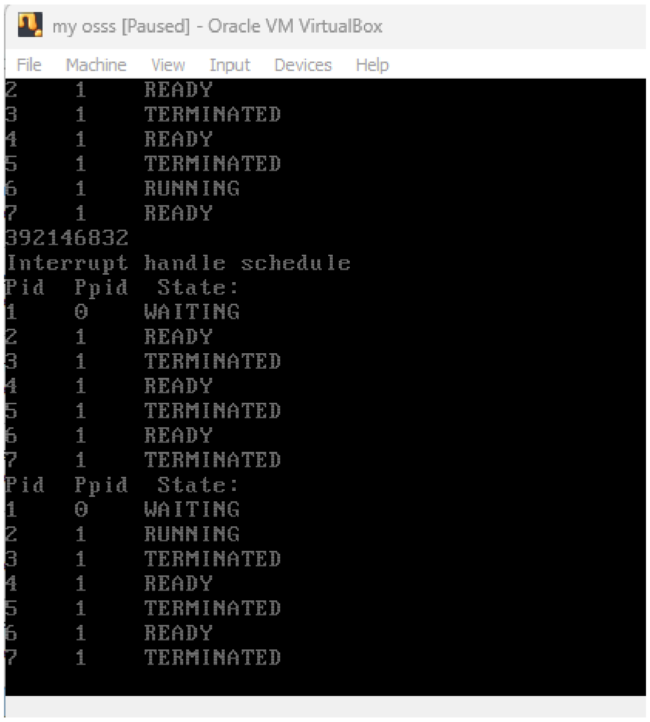
```
7     1     TERMINATED
PID   PPID   State:
1     0     RUNNING
2     1     TERMINATED
3     1     TERMINATED
4     1     TERMINATED
5     1     TERMINATED
6     1     TERMINATED
7     1     TERMINATED
PID   PPID   State:
1     0     RUNNING
2     1     TERMINATED
3     1     TERMINATED
4     1     TERMINATED
5     1     TERMINATED
```

Parent with exit:


my osss [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

```
4     1     TERMINATED
5     1     TERMINATED
6     1     TERMINATED
7     1     TERMINATED
PID   PPID   State:
1     0     RUNNING
2     1     TERMINATED
3     1     TERMINATED
4     1     TERMINATED
5     1     TERMINATED
6     1     TERMINATED
7     1     TERMINATED
Interrupt handle schedule
PID   PPID   State:
1     0     TERMINATED
2     1     TERMINATED
3     1     TERMINATED
4     1     TERMINATED
5     1     TERMINATED
6     1     TERMINATED
7     1     TERMINATED
```

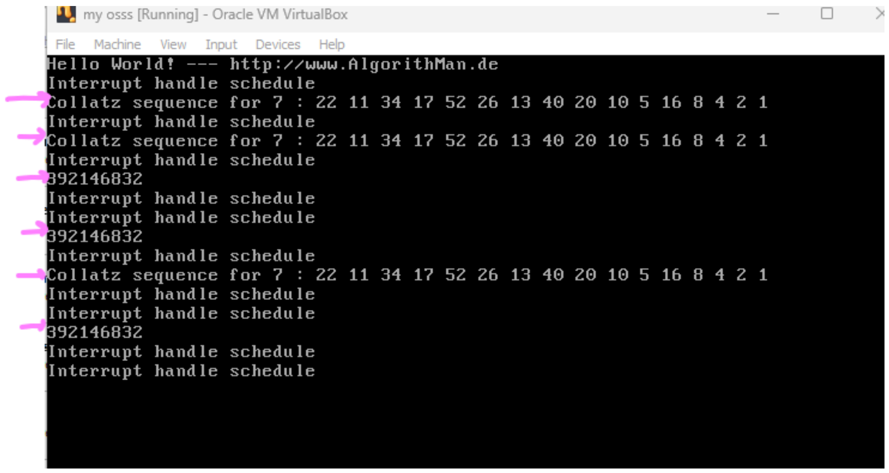
At the middle of execution:


my osss [Paused] - Oracle VM VirtualBox

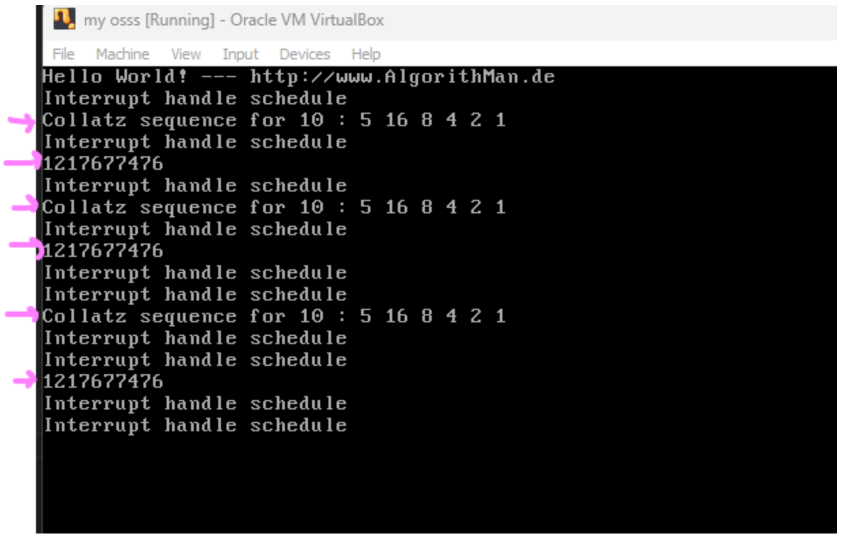
```
2     1     READY
3     1     TERMINATED
4     1     READY
5     1     TERMINATED
6     1     RUNNING
7     1     READY
392146832
Interrupt handle schedule
Pid  Ppid  State:
1     0     WAITING
2     1     READY
3     1     TERMINATED
4     1     READY
5     1     TERMINATED
6     1     READY
7     1     TERMINATED
Pid  Ppid  State:
1     0     WAITING
2     1     RUNNING
3     1     TERMINATED
4     1     READY
5     1     TERMINATED
6     1     READY
7     1     TERMINATED
```

Without process table:

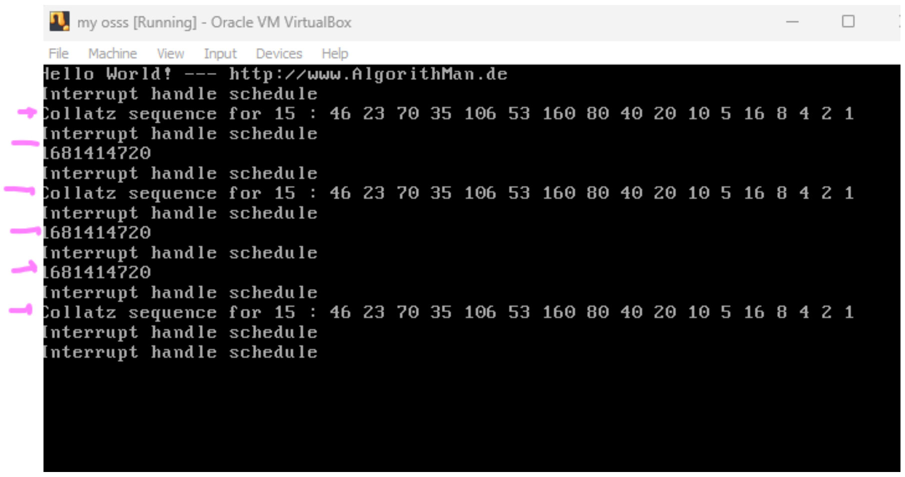
I tried the functions with different values. As you can see their order is different from each other. So that we can observe the timer interrupt with the output of “Interrupt handle schedule”. And we can observe the round robin scheduling.


my osss [Running] - Oracle VM VirtualBox

```
Hello World! --- http://www.AlgorithMan.de
Interrupt handle schedule
Collatz sequence for 7 : 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
Interrupt handle schedule
Collatz sequence for 7 : 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
Interrupt handle schedule
392146832
Interrupt handle schedule
Interrupt handle schedule
392146832
Interrupt handle schedule
Collatz sequence for 7 : 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
Interrupt handle schedule
Interrupt handle schedule
392146832
Interrupt handle schedule
Interrupt handle schedule
```


my osss [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

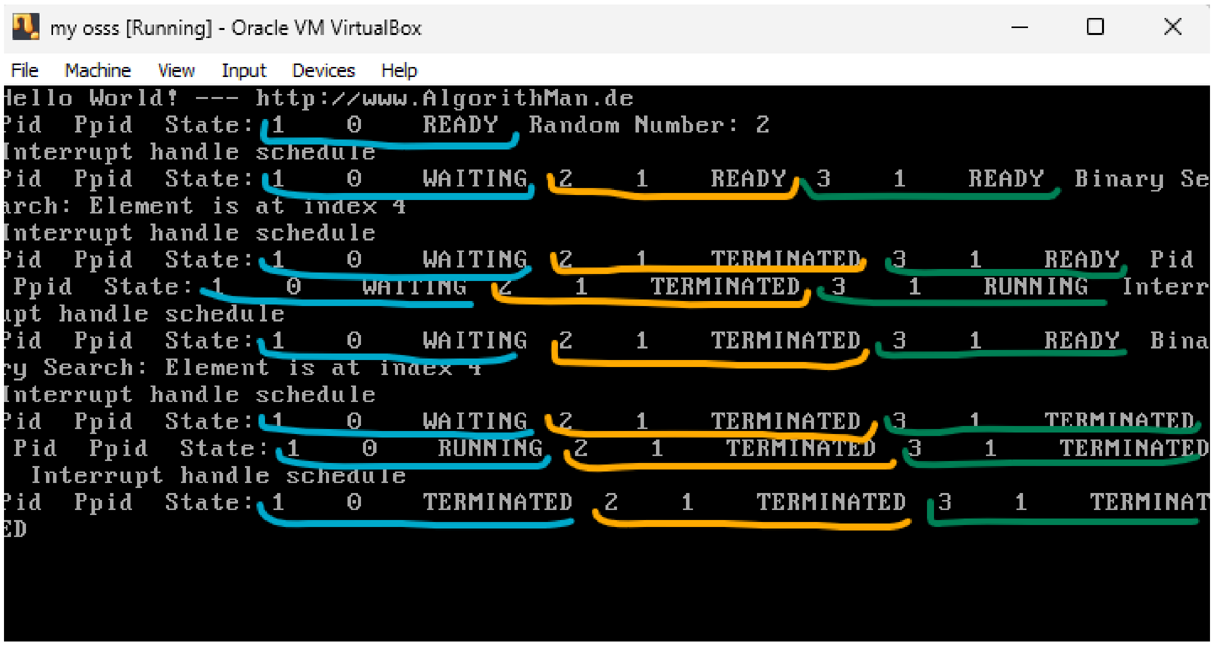
```
Hello World! --- http://www.AlgorithMan.de
Interrupt handle schedule
Collatz sequence for 10 : 5 16 8 4 2 1
Interrupt handle schedule
1217677476
Interrupt handle schedule
Collatz sequence for 10 : 5 16 8 4 2 1
Interrupt handle schedule
1217677476
Interrupt handle schedule
Interrupt handle schedule
Collatz sequence for 10 : 5 16 8 4 2 1
Interrupt handle schedule
Interrupt handle schedule
1217677476
Interrupt handle schedule
Interrupt handle schedule
```


my osss [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

```
Hello World! --- http://www.AlgorithMan.de
Interrupt handle schedule
Collatz sequence for 15 : 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
Interrupt handle schedule
1681414720
Interrupt handle schedule
Collatz sequence for 15 : 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
Interrupt handle schedule
1681414720
Interrupt handle schedule
1681414720
Interrupt handle schedule
Collatz sequence for 15 : 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
Interrupt handle schedule
Interrupt handle schedule
```

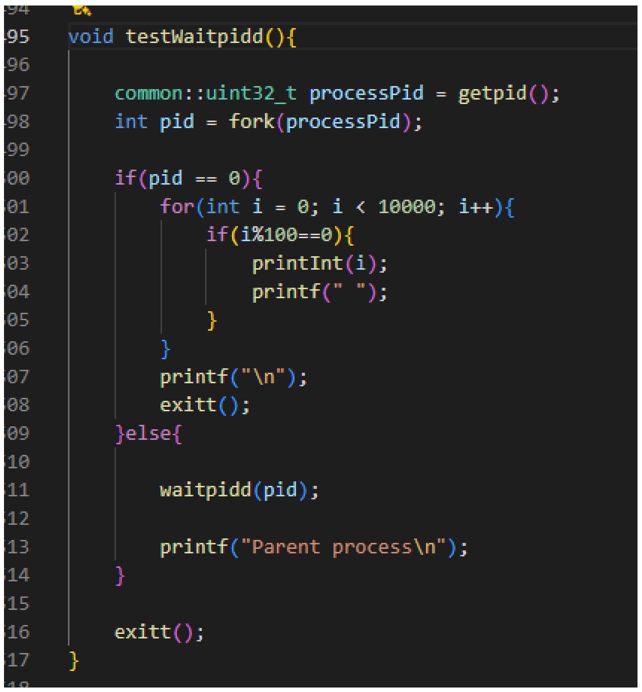
Syscall tests separately:

Fork test: To see the forks clearly on the process table, I tried 2 forks. You can see there are 2 child and 1 parent. And you can see their states.

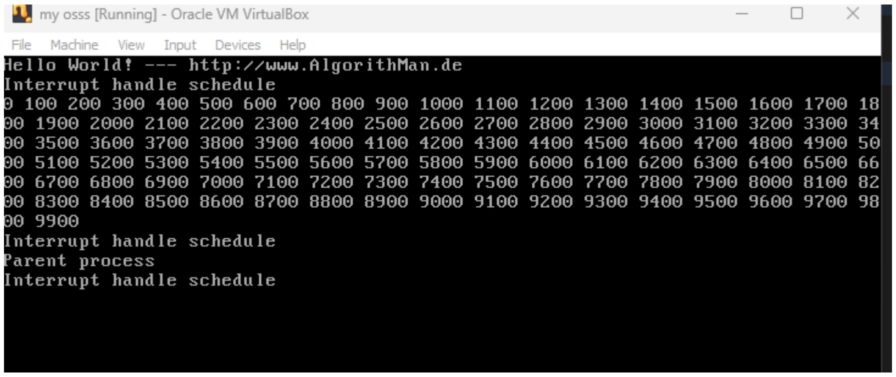

my osss [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

```
Hello World! --- http://www.AlgorithMan.de
Pid  Ppid  State: 1    0    READY  Random Number: 2
Interrupt handle schedule
Pid  Ppid  State: 1    0    WAITING  2    1    READY  3    1    READY  Binary Se
arch: Element is at index 4
Interrupt handle schedule
Pid  Ppid  State: 1    0    WAITING  2    1    TERMINATED  3    1    READY  Pid
 Ppid  State: 1    0    WAITING  2    1    TERMINATED  3    1    RUNNING  Interr
upt handle schedule
Pid  Ppid  State: 1    0    WAITING  2    1    TERMINATED  3    1    READY  Bina
ry Search: Element is at index 4
Interrupt handle schedule
Pid  Ppid  State: 1    0    WAITING  2    1    TERMINATED  3    1    TERMINATED
 Pid  Ppid  State: 1    0    RUNNING  2    1    TERMINATED  3    1    TERMINATED
  Interrupt handle schedule
Pid  Ppid  State: 1    0    TERMINATED  2    1    TERMINATED  3    1    TERMINAT
ED
```

Waitpid test:



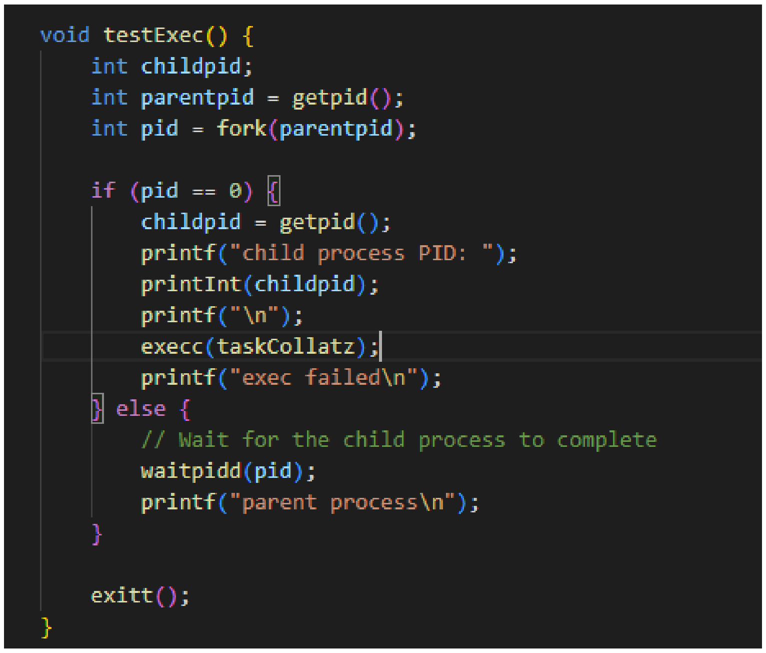
```
void testWaitpidd(){

    common::uint32_t processPid = getpid();
    int pid = fork(processPid);

    if(pid == 0){
        for(int i = 0; i < 10000; i++){
            if(i%100==0){
                printInt(i);
                printf(" ");
            }
        }
        printf("\n");
        exitt();
    }else{

        waitpidd(pid);

        printf("Parent process\n");
    }

    exitt();
}
```



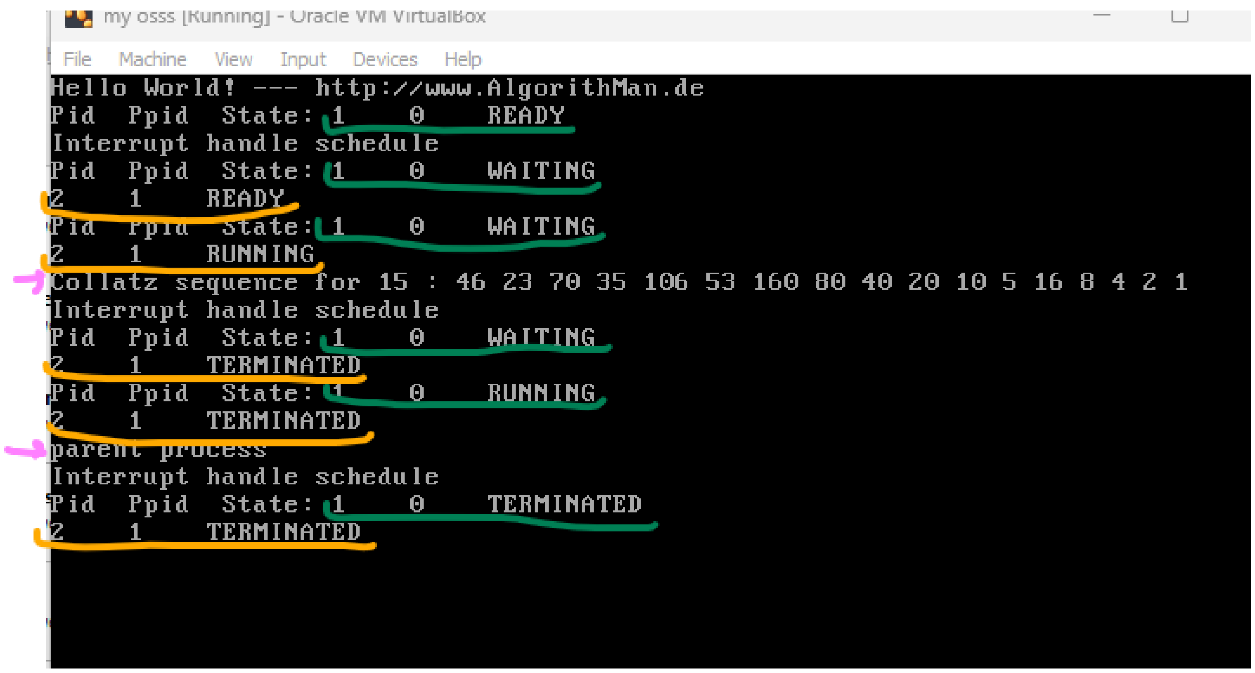
my osss [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

```
Hello World! --- http://www.AlgorithMan.de
Interrupt handle schedule
0 100 200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400 1500 1600 1700 18
00 1900 2000 2100 2200 2300 2400 2500 2600 2700 2800 2900 3000 3100 3200 3300 34
00 3500 3600 3700 3800 3900 4000 4100 4200 4300 4400 4500 4600 4700 4800 4900 50
00 5100 5200 5300 5400 5500 5600 5700 5800 5900 6000 6100 6200 6300 6400 6500 66
00 6700 6800 6900 7000 7100 7200 7300 7400 7500 7600 7700 7800 7900 8000 8100 82
00 8300 8400 8500 8600 8700 8800 8900 9000 9100 9200 9300 9400 9500 9600 9700 98
00 9900
Interrupt handle schedule
Parent process
Interrupt handle schedule
```

Exec test:



```
void testExec() {
    int childpid;
    int parentpid = getpid();
    int pid = fork(parentpid);

    if (pid == 0) {
        childpid = getpid();
        printf("child process PID: ");
        printInt(childpid);
        printf("\n");
        execc(taskCollatz);
        printf("exec failed\n");
    } else {
        // Wait for the child process to complete
        waitpidd(pid);
        printf("parent process\n");
    }

    exitt();
}
```



my osss [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

```
Hello World! --- http://www.AlgorithMan.de
Pid  Ppid  State: 1    0    READY
Interrupt handle schedule
Pid  Ppid  State: 1    0    WAITING
2     1    READY
Pid  Ppid  State: 1    0    WAITING
2     1    RUNNING
Collatz sequence for 15 : 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
Interrupt handle schedule
Pid  Ppid  State: 1    0    WAITING
2     1    TERMINATED
Pid  Ppid  State: 1    0    RUNNING
2     1    TERMINATED
parent process
Interrupt handle schedule
Pid  Ppid  State: 1    0    TERMINATED
2     1    TERMINATED
```