Gebze Technical University

Department of Computer Engineering

CSE312 Operating Systems

Homework 1 Report

Burak Yıldırım
1901042609

Contents

1 Introduction

In this homework, we're expected to implement the `fork`, `waitpid`, `execve` system calls and the other system call that we might need, and handle interrupts using the operating system written by Victor Engelmann.

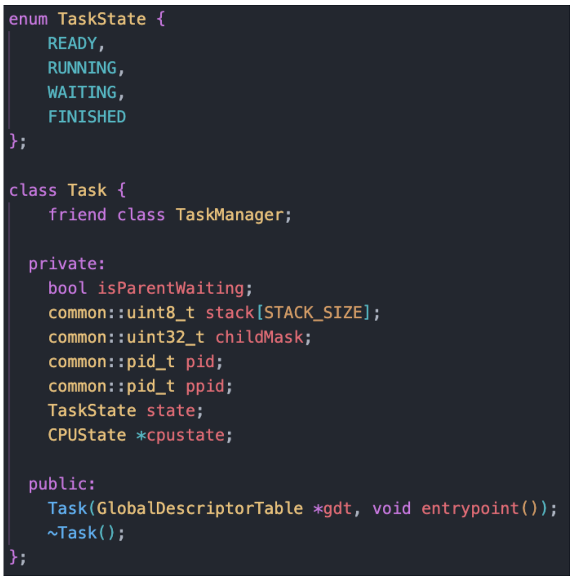
2 Implementation

2.1 TaskManager and Task

To comply with the requirements of this homework, I added some members to both the Task class and the TaskManager class.

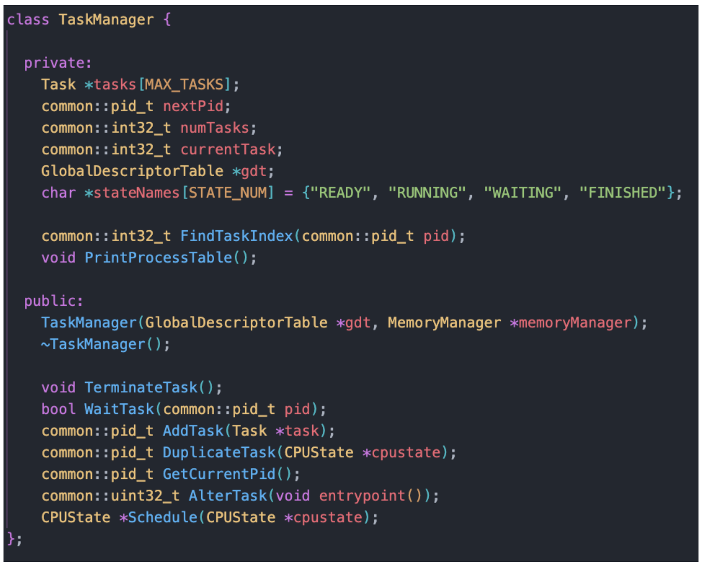
For the Task class, I added the process ID, parent process ID, state of the task, `childMask`, which consists of 1s equivalent to the number of tasks that the task is waiting for, and an `isParentWaiting` variable to check if this task is being waited on by the parent process.

For the TaskManager class, I added a counter for the next process ID, a `GlobalDescriptorTable` pointer to get a new code segment for `execve` system call, and an array of string representing the names of the task states. Since the initial code Victor Engelmann wrote had Task pointer array for tasks, I continued with that approach. I allocated dynamic memory for each Task pointer in the array so that tasks wouldn't point to the same location and override each other. For the dynamic allocation I used Victor Engelmann's MemoryManager class. Also all the system calls are implemented in this class.



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

class Task {
    friend class TaskManager;

  private:
    bool isParentWaiting;
    common::uint8_t stack[STACK_SIZE];
    common::uint32_t childMask;
    common::pid_t pid;
    common::pid_t ppid;
    TaskState state;
    CPUState *cpustate;

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

Figure 1: Task class



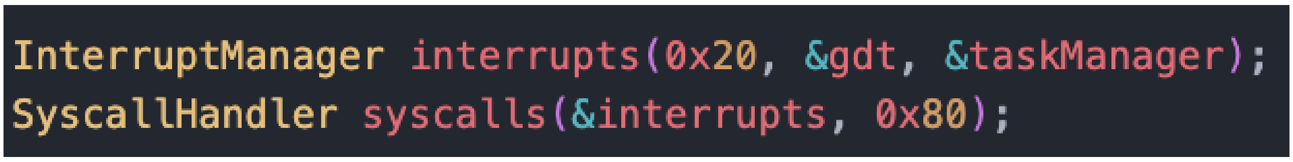
```
class TaskManager {

  private:
    Task *tasks[MAX_TASKS];
    common::pid_t nextPid;
    common::int32_t numTasks;
    common::int32_t currentTask;
    GlobalDescriptorTable *gdt;
    char *stateNames[STATE_NUM] = {"READY", "RUNNING", "WAITING", "FINISHED"};

    common::int32_t FindTaskIndex(common::pid_t pid);
    void PrintProcessTable();

  public:
    TaskManager(GlobalDescriptorTable *gdt, MemoryManager *memoryManager);
    ~TaskManager();

    void TerminateTask();
    bool WaitTask(common::pid_t pid);
    common::pid_t AddTask(Task *task);
    common::pid_t DuplicateTask(CPUState *cpustate);
    common::pid_t GetCurrentPid();
    common::uint32_t AlterTask(void entrypoint());
    CPUState *Schedule(CPUState *cpustate);
};
```

Figure 2: TaskManager class

2.2 System Call

2.2.1 Handling

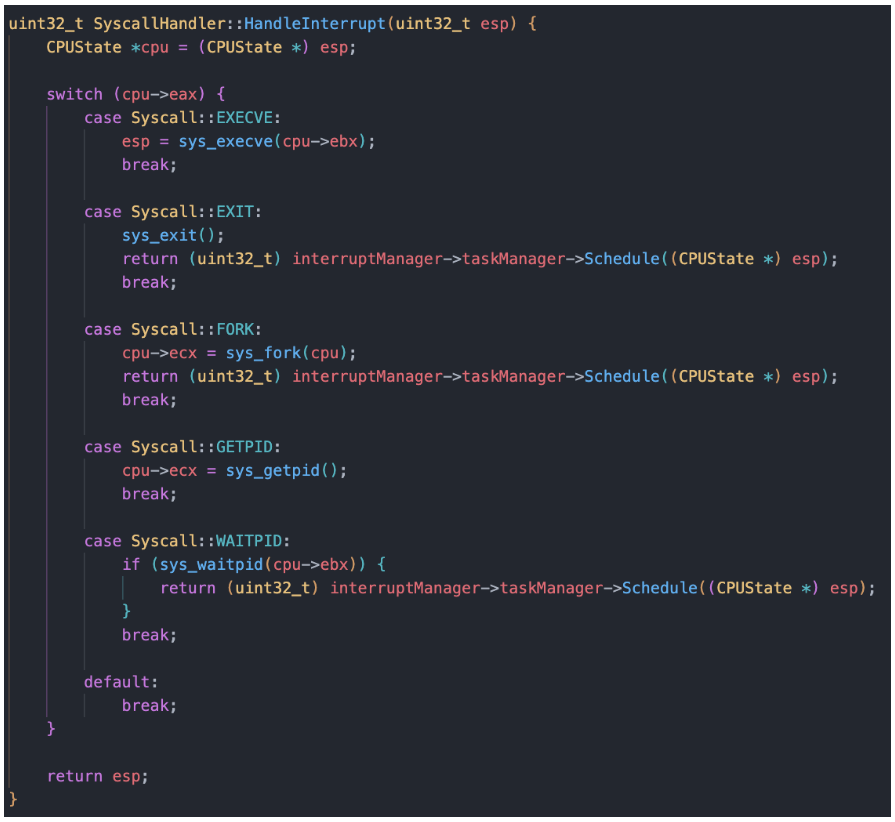
To handle the system calls, I used the same pattern as Victor Engelmann did. For any system call, I triggered an `0x80` interrupt which is then catched by the `SyscallHandler` since it was registered to catch interrupts with this number on here:



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

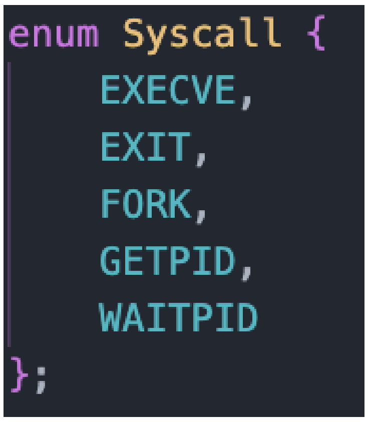
Figure 3: System call interrupt number registery

When the interrupt is catched, the `HandleInterrupt` function of the `SyscallHandler` is fired. In the handler, system calls are handled like the Victor Engelmann's `sysprintf`. To differentiate the system calls, I created the `Syscall` enum, so whenever a system call is issued, appropriate enum value is written to the `eax` register of the `cpustate`.



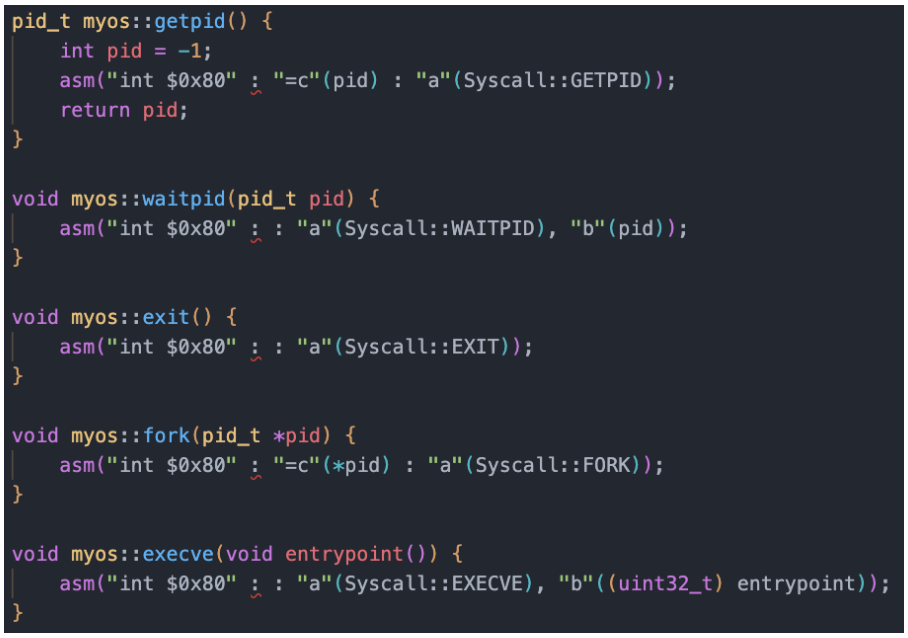
```
uint32_t SyscallHandler::HandleInterrupt(uint32_t esp) {
    CPUState *cpu = (CPUState *) esp;

    switch (cpu->eax) {
        case Syscall::EXECVE:
            esp = sys_execve(cpu->ebx);
            break;

        case Syscall::EXIT:
            sys_exit();
            return (uint32_t) interruptManager->taskManager->Schedule((CPUState *) esp);
            break;

        case Syscall::FORK:
            cpu->ecx = sys_fork(cpu);
            return (uint32_t) interruptManager->taskManager->Schedule((CPUState *) esp);
            break;

        case Syscall::GETPID:
            cpu->ecx = sys_getpid();
            break;

        case Syscall::WAITPID:
            if (sys_waitpid(cpu->ebx)) {
                return (uint32_t) interruptManager->taskManager->Schedule((CPUState *) esp);
            }
            break;

        default:
            break;
    }

    return esp;
}
```

Figure 4: System call handling



```
enum Syscall {
    EXECVE,
    EXIT,
    FORK,
    GETPID,
    WAITPID
};
```

Figure 5: Syscall enum

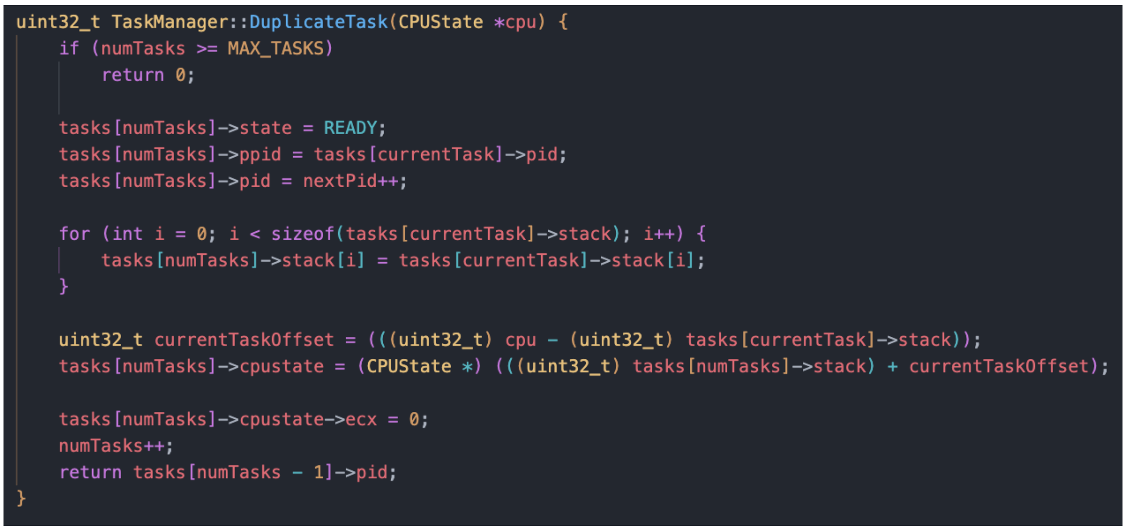


```
pid_t myos::getpid() {
    int pid = -1;
    asm("int $0x80" : "=c"(pid) : "a"(Syscall::GETPID));
    return pid;
}

void myos::waitpid(pid_t pid) {
    asm("int $0x80" : : "a"(Syscall::WAITPID), "b"(pid));
}

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

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

void myos::execve(void entrypoint()) {
    asm("int $0x80" : : "a"(Syscall::EXECVE), "b"((uint32_t) entrypoint));
}
```

Figure 6: System call functions

2.2.2 fork

First the state of the child process is set to `READY`. Then the process ID and the parent process ID of the child process are set. After that the whole stack of the parent process is copied to the child process. After the copying is done, the current offset of the parent's CPU state is calculated and added to the beginning position of the child's stack. This is done to ensure that the parent and the child continues from the same position. After that the `ecx` register of the child's CPU state is set to 0 to differentiate the parent and the child. Then the number of tasks is incremented by 1 and the pid of the child process returned. Finally the returned value is written to the `ecx` register of the parent's CPU state. After fork, I call the scheduler. If I don't call the scheduler, the child doesn't continue right after where the fork was in the code.



```
uint32_t TaskManager::DuplicateTask(CPUState *cpu) {
    if (numTasks >= MAX_TASKS)
        return 0;

    tasks[numTasks]->state = READY;
    tasks[numTasks]->ppid = tasks[currentTask]->pid;
    tasks[numTasks]->pid = nextPid++;

    for (int i = 0; i < sizeof(tasks[currentTask]->stack); i++) {
        tasks[numTasks]->stack[i] = tasks[currentTask]->stack[i];
    }

    uint32_t currentTaskOffset = (((uint32_t) cpu - (uint32_t) tasks[currentTask]->stack));
    tasks[numTasks]->cpustate = (CPUState *) (((uint32_t) tasks[numTasks]->stack) + currentTaskOffset);

    tasks[numTasks]->cpustate->ecx = 0;
    numTasks++;
    return tasks[numTasks - 1]->pid;
}
```

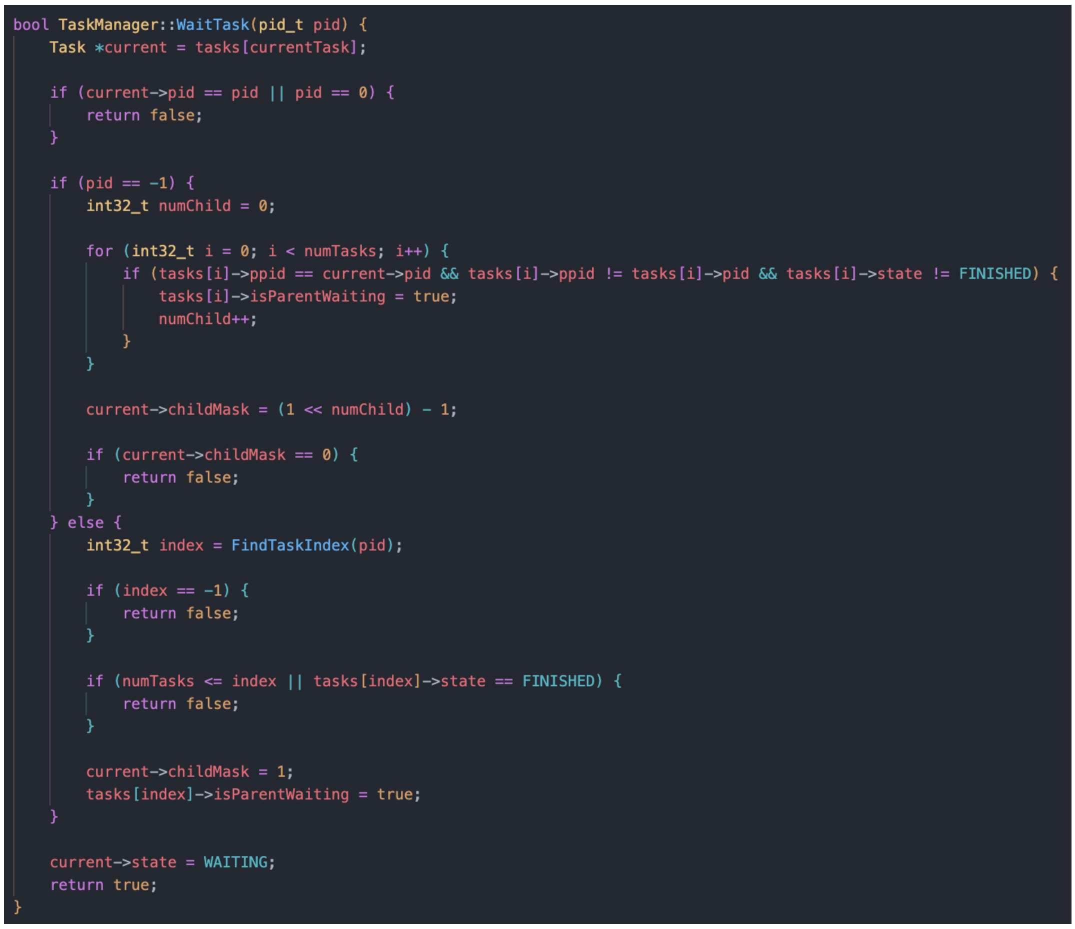
Figure 7: fork implementation

2.2.3 waitpid

First the pointer to the current process is set. Then it's checked if the given pid is 0 or if it's same as the pid of the current process. If these checks are passed then it's checked if the pid is -1. -1 means wait all the child processes, not just one. If pid is -1 the process table is searched with the given criteria:

- the ppid of the process should be the pid of the current process.
- the ppid and the pid of the process should not be the same (to prevent the init process from waiting itself).
- the state of the process should no be `FINISHED`.

`isParentWaiting` variable of each process found this way is set to `true`, and the number of processes found is stored in `numChild`. Then the `childMask` of the current process is set to $(1 \ll$ `numChild`$) - 1$, which is as many 1s as `numChild`, e.g. if the `numChild` is 3 then the `childMask` is 111. After this if the `childMask` is 0, it means there is no child to wait so it returns `false`. On the other hand if the pid is not -1 after the initial checks, then the index of the task which the pid belongs is found. If the task exists and is not finished, the `childMask` is set to 1 and the `isParentWaiting` of the task to wait is set to `true`. Finally, for both cases, state of the current task is set to `WAITING` and `true` is returned.

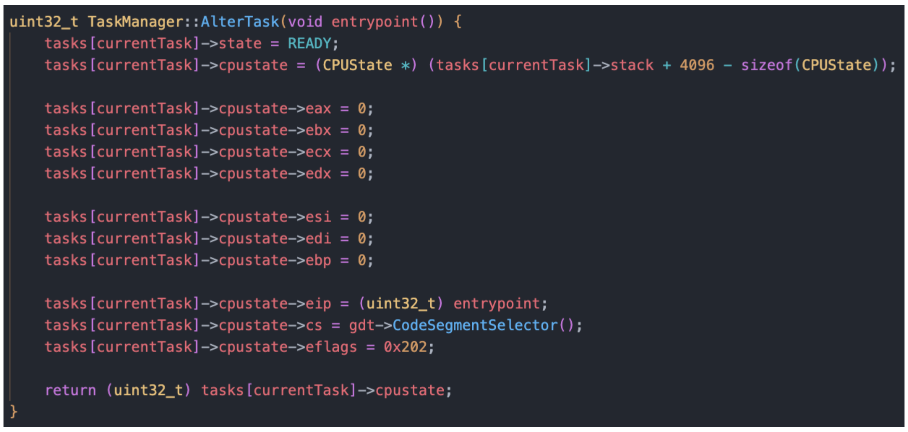


```
bool TaskManager::WaitTask(pid_t pid) {
    Task *current = tasks[currentTask];

    if (current->pid == pid || pid == 0) {
        return false;
    }

    if (pid == -1) {
        int32_t numChild = 0;

        for (int32_t i = 0; i < numTasks; i++) {
            if (tasks[i]->ppid == current->pid && tasks[i]->ppid != tasks[i]->pid && tasks[i]->state != FINISHED) {
                tasks[i]->isParentWaiting = true;
                numChild++;
            }
        }

        current->childMask = (1 << numChild) - 1;

        if (current->childMask == 0) {
            return false;
        }
    } else {
        int32_t index = FindTaskIndex(pid);

        if (index == -1) {
            return false;
        }

        if (numTasks <= index || tasks[index]->state == FINISHED) {
            return false;
        }

        current->childMask = 1;
        tasks[index]->isParentWaiting = true;
    }

    current->state = WAITING;
    return true;
}
```

Figure 8: waitpid implementation

2.2.4 execve

This one is fairly simple. First the CPU state of the current process is set to the beginning of the stack, and all the registers of the CPU state is set to 0. After that the instruction pointer `eip` is set to the given entrypoint and a new code segment is assigned. Finally the CPU state of the current process is returned.

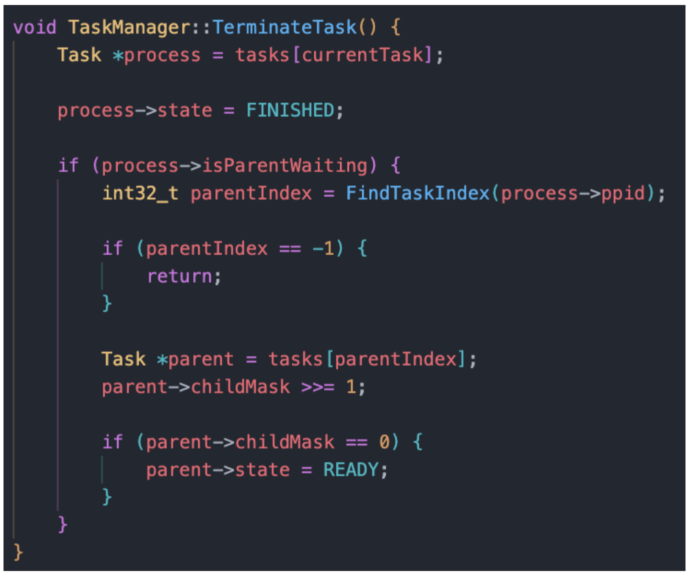


```
uint32_t TaskManager::AlterTask(void entrypoint()) {
    tasks[currentTask]->state = READY;
    tasks[currentTask]->cpustate = (CPUState *) (tasks[currentTask]->stack + 4096 - sizeof(CPUState));

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

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

    tasks[currentTask]->cpustate->eip = (uint32_t) entrypoint;
    tasks[currentTask]->cpustate->cs = gdt->CodeSegmentSelector();
    tasks[currentTask]->cpustate->eflags = 0x202;

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

Figure 9: execve implementation

2.2.5 exit

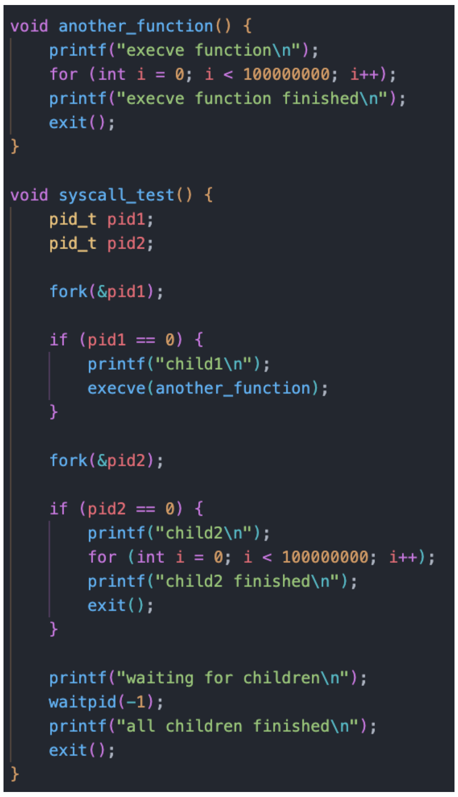
First the state of the current process is set to `FINISHED`. Then if the `isParentWaiting` is `true`, the index of the parent process is found. If index is -1, function returns. If not, the `childMask` of the parent process is shifted to the right by 1. If the `childMask` becomes 0 after the shift it means that the current process was the last process the parent was waiting, so the state of the parent is changed to `READY`.



```
void TaskManager::TerminateTask() {
    Task *process = tasks[currentTask];

    process->state = FINISHED;

    if (process->isParentWaiting) {
        int32_t parentIndex = FindTaskIndex(process->ppid);

        if (parentIndex == -1) {
            return;
        }

        Task *parent = tasks[parentIndex];
        parent->childMask >>= 1;

        if (parent->childMask == 0) {
            parent->state = READY;
        }
    }
}
```

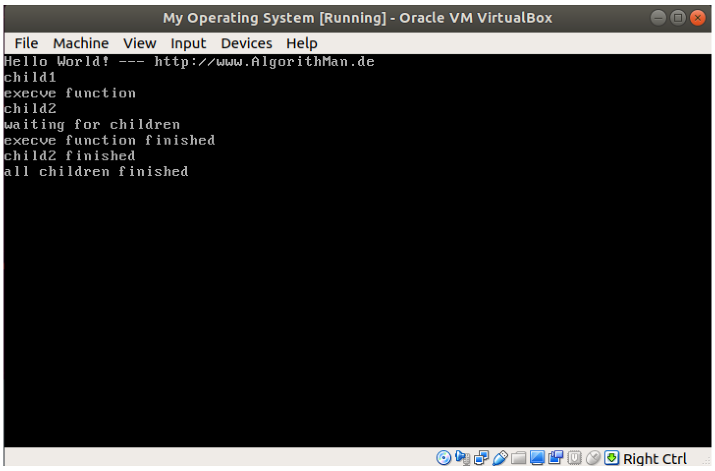
Figure 10: exit implementation

2.2.6 Testing



```
void another_function() {
    printf("execve function\n");
    for (int i = 0; i < 100000000; i++);
    printf("execve function finished\n");
    exit();
}

void syscall_test() {
    pid_t pid1;
    pid_t pid2;

    fork(&pid1);

    if (pid1 == 0) {
        printf("child1\n");
        execve(another_function);
    }

    fork(&pid2);

    if (pid2 == 0) {
        printf("child2\n");
        for (int i = 0; i < 100000000; i++);
        printf("child2 finished\n");
        exit();
    }

    printf("waiting for children\n");
    waitpid(-1);
    printf("all children finished\n");
    exit();
}
```

Figure 11: Test functions

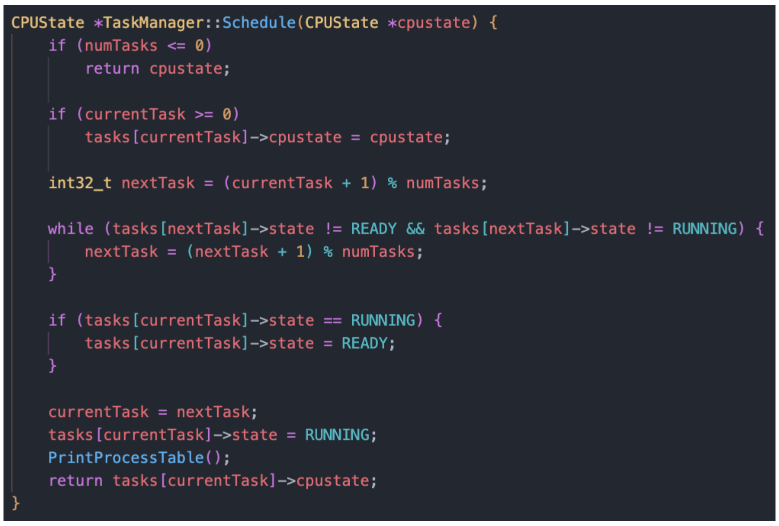

My Operating System [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

```
Hello World! --- http://www.AlgorithMan.de
child1
execve function
child2
waiting for children
execve function finished
child2 finished
all children finished
```

Figure 12: Test results

2.3 Scheduler

The scheduler is implemented as in the Figure 13. It works like regular Round Robin scheduler. If the state of a process is `FINISHED` or `WAITING`, it's not executed.



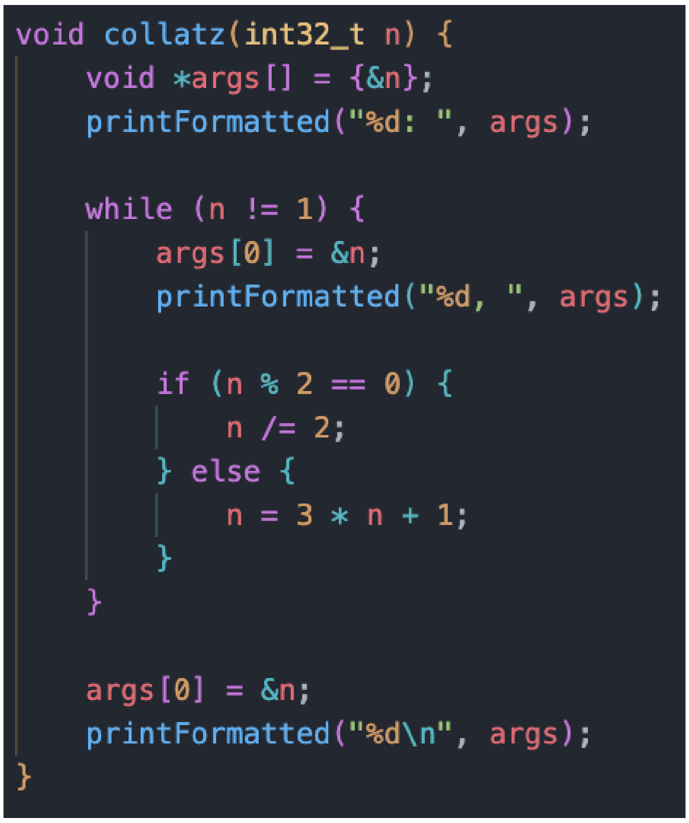
```
CPUState *TaskManager::Schedule(CPUState *cpustate) {
    if (numTasks <= 0)
        return cpustate;

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

    int32_t nextTask = (currentTask + 1) % numTasks;

    while (tasks[nextTask]->state != READY && tasks[nextTask]->state != RUNNING) {
        nextTask = (nextTask + 1) % numTasks;
    }

    if (tasks[currentTask]->state == RUNNING) {
        tasks[currentTask]->state = READY;
    }

    currentTask = nextTask;
    tasks[currentTask]->state = RUNNING;
    PrintProcessTable();
    return tasks[currentTask]->cpustate;
}
```

Figure 13: Scheduler implementation

3 Lifecycles

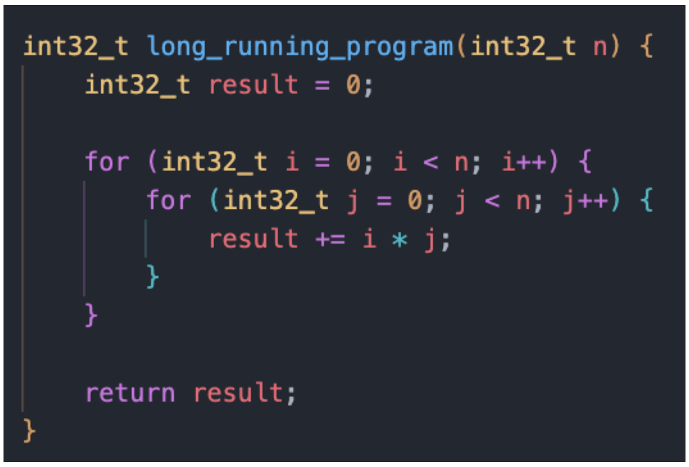
3.1 Functions

All the functions here are written by ChatGPT4. I just changed the `int` types to `int32_t` type.



```c
void collatz(int32_t n) {
    void *args[] = {&n};
    printFormatted("%d: ", args);

    while (n != 1) {
        args[0] = &n;
        printFormatted("%d, ", args);

        if (n % 2 == 0) {
            n /= 2;
        } else {
            n = 3 * n + 1;
        }
    }

    args[0] = &n;
    printFormatted("%d\n", args);
}
```

Figure 14: Collatz

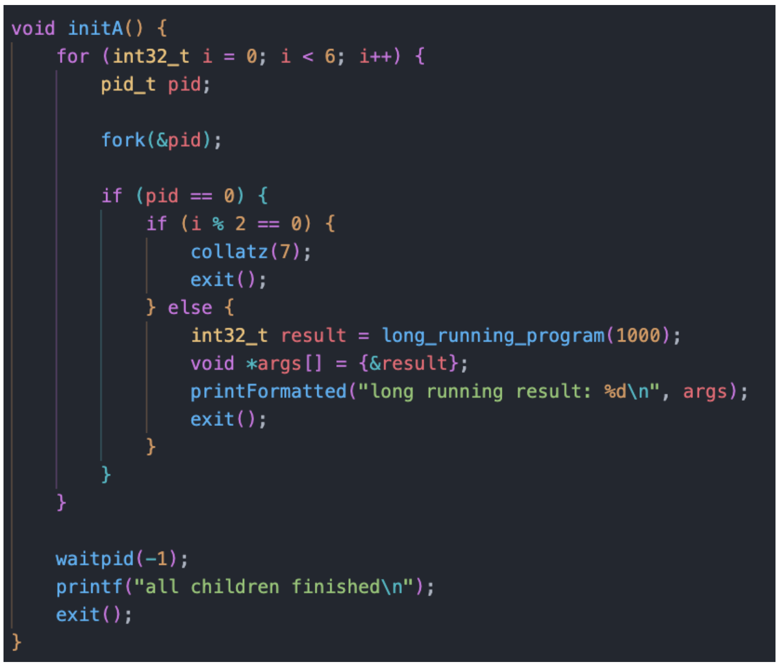


```c
int32_t long_running_program(int32_t n) {
    int32_t result = 0;

    for (int32_t i = 0; i < n; i++) {
        for (int32_t j = 0; j < n; j++) {
            result += i * j;
        }
    }

    return result;
}
```

Figure 15: Long running program

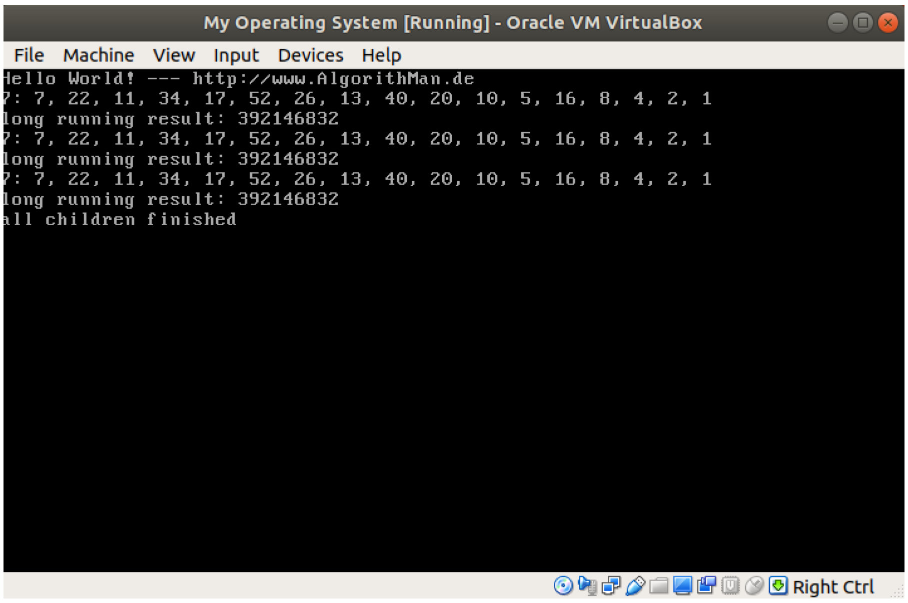
3.2 Part A

Strategy of part A is loading each program 3 times and performing round robin scheduling. Unfortunatelly, due to the collatz ending so fast and timer interrupts being so frequent I couldn't see or catch any process table printing except the last one. The final process table print shows that all the processes have executed successfully and the parent waited all of them.



```
void initA() {
    for (int32_t i = 0; i < 6; i++) {
        pid_t pid;

        fork(&pid);

        if (pid == 0) {
            if (i % 2 == 0) {
                collatz(7);
                exit();
            } else {
                int32_t result = long_running_program(1000);
                void *args[] = {&result};
                printFormatted("long running result: %d\n", args);
                exit();
            }
        }
    }

    waitpid(-1);
    printf("all children finished\n");
    exit();
}
```

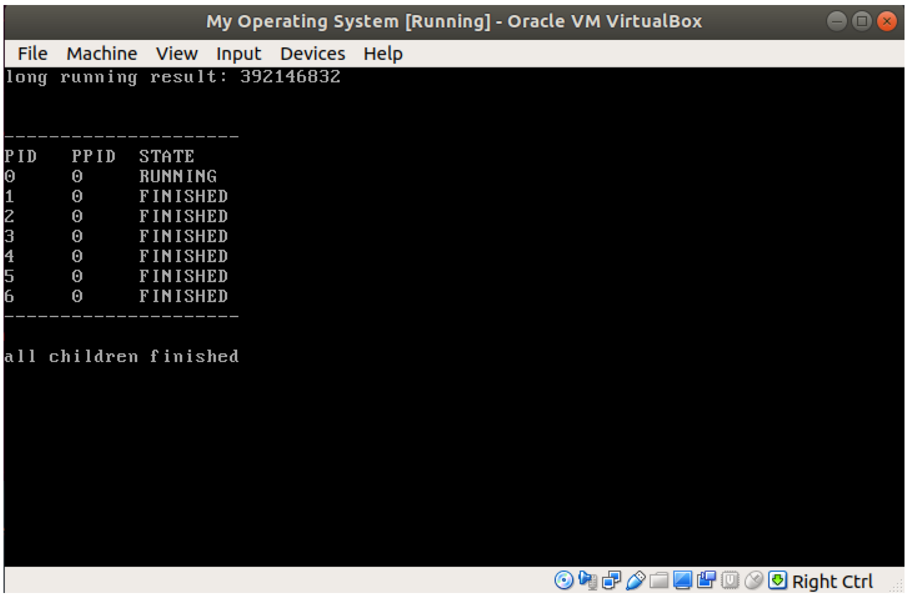
Figure 16: Part A



My Operating System [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

```
Hello World! --- http://www.AlgorithMan.de
7: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
long running result: 392146832
7: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
long running result: 392146832
7: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
long running result: 392146832
all children finished
```

Figure 17: Part A results


My Operating System [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

```
long running result: 392146832


---------------------
PID   PPID  STATE
0     0     RUNNING
1     0     FINISHED
2     0     FINISHED
3     0     FINISHED
4     0     FINISHED
5     0     FINISHED
6     0     FINISHED
---------------------

all children finished
```

Figure 18: Part A results with process table