

Gebze Technical University

Department Of Computer Engineering

CSE 312 /CSE 504 Spring 2023

Operating Systems

Homework #01

Due Date: 10.04.2023

Abdurrahman Bulut

1901042258

## Introduction

I followed the instructions given to set up my environment for this homework. I watched three videos by Viktor Engelmann, which explained the basic design of the operating system we will create in class and showed how to install it in a virtual box. I then downloaded the source code related to 15th video from GitHub and used them to write my own operating system code for the assignment. I started initially after making sure that the source code in the 15th video works correctly. Also I used VirtualBox which is used by the instructor in the youtube series.

### Task 1: “Implementing these POSIX system calls: fork, waitpid, execve, any other POSIX call that you need.”

In this task I follow the youtube series. The video named “Write your own Operating System 20: System calls, POSIX compliance” explains how to call printf as a system call. Then I made similar things for fork(), waitpid() and others. I used the 32bit posix system calls table and for example I gave 2 to eax register for fork() system call. In this video, he told some complaints about Posix but we can use it in the syscall.cpp file.

```
int sysfork(){
    int result;
    asm("int $0x80" : "=a" (result) : "a" (2));
    return result;
}
int sysgetpid(){
    int result;
    asm volatile("int $0x80" : "=a" (result) : "a" (20));
    return result;
}
int syswaitpid(int pid, int* status, int options) {
    int result;
    asm("int $0x80" : "=a" (result) : "a" (7), "b" (pid), "c" (status), "d" (options));
    return result;
}
int sysexecve(const char *name, char *const argv[], char *const envp[]){
    int result;
    asm volatile("int $0x80" : "=a" (result) : "a" (11), "b" (name), "c" (argv), "d" (envp));
    return result;
}
int sysgetrandom(int max_value) {
    int result;
    asm("int $0x80" : "=a" (result) : "a" (7), "b" (max_value));
    return result;
}
/*
*****
*/
```

In the above picture, “a” means “eax”, “b” means “ebx”, “c” means “ecx”, “d” means “edx”. I gave some numbers to eax by looking at the System Calls table for 32 bit Linux. [“Source”](#) .

I implemented sysSYSTEM\_CALL\_NAME to trigger software interrupts. \$0x80 means software interrupt. Then, I catch these numbers from the “eax” register in the syscalls.cpp file.

```
switch(cpu->eax)
{
    case 2:
        cpu->eax = fork();
        break;
    case 4:
        printf((char*)cpu->ebx);
        break;
    case 7:
        waitpid((int)cpu->ebx, (int*)cpu->ecx, cpu->edx);
        break;
    case 11:
        cpu->eax = execve((const char*)cpu->ebx, (char *const *)cpu->ecx, (char *const *)cpu->edx);
        break;
    case 20:
        cpu->eax = getpid();
        break;
    case 318:
        cpu->eax = getrandom((int)cpu->ebx);
        break;
    default:
        break;
}

return esp;
```

Then, in case statements I call each method. These methods are implemented in kernel.cpp. Because I need to access taskManager to manipulate the processes. The methods are shown in the below picture.

```

int fork() {
    Task* currentTask = taskManagerPtr->GetCurrentTask();
    Task* newTask = taskManagerPtr->fork();

    if (currentTask->get_pid() == newTask->get_pid()) {
        return 0; // Child process returns 0
    } else {
        return newTask->get_pid(); // Parent process returns the child's PID
    }
}

int waitpid( int pid, int* status, int options) {
    return taskManagerPtr->WaitForProcess(pid, status, options);
}

int getpid() {
    return taskManagerPtr->GetCurrentTask()->get_pid();
}

int getRandom(int max_value){
    // seed initialized as global like this: uint32_t seed = 1;
    seed = seed * 1103515245 + 12345;
    return (seed / 65536) % max_value;
}

int execve(const char *name, char *const argv[], char *const envp[]) {
    //return load_new_process(name, argv, envp, taskManagerPtr); I don't know to implement image thing.
}

```

Task 2 - 3: “Loading multiple programs into memory: Kernel will be able to load multiple programs into memory.” and “Handling multi-programming: you need to develop a Process Table that will hold the necessary information about the processes in the memory. You should study what Process Tables hold. You can read carefully throughout chapter 2 of the course book or any other online resource). ”

An example of loading multiple tasks (programs) with predefined entry points into my TaskManager. I am adding multiple programs to my TaskManager as shown in the below picture.

```

GlobalDescriptorTable gdt;

TaskManager taskManager;
taskManagerPtr = &taskManager;

Task task1(&gdt, taskA);
Task task2(&gdt, taskB);
Task task3(&gdt, taskC);
Task task4(&gdt, taskD);
Task task5(&gdt, taskE);
Task task6(&gdt, taskF);
taskManager.AddTask(&task1);
taskManager.AddTask(&task2);
taskManager.AddTask(&task3);
taskManager.AddTask(&task4);
taskManager.AddTask(&task5);
taskManager.AddTask(&task6);

```

taskManagerPtr is used to access from everywhere on my kernel.cpp. I needed it for system calls implementations. I defined this above of the code;

```
myos::TaskManager* taskManagerPtr;
```

I will explain some methods in TaskManager class;

```
bool TaskManager::AddTask(Task *task)
{
    if (numTasks < 256) {
        tasks[numTasks] = task;
        tasks[numTasks]->pid = numTasks+1;
        tasks[numTasks]->state = READY;
        tasks[numTasks]->memory_alloc = 4096;

        for (int j = 0; j < 10; j++) {
            tasks[numTasks]->open_files[j] = 0;
        }
        numTasks++;
        return true;
    }
    return false;
}
```

This adds a new task to the TaskManager. Initialize some properties of tasks and it returns true if the task was successfully added, false otherwise. I used numTasks property as process id. Because it is unique for each process.

```

Task* TaskManager::copyTask(Task* source) {

    Task* newTask = source;

    // Copy the source task's stack and CPU state to the new task
    for (unsigned int i = 0; i < sizeof(source->stack); i++) {
        newTask->stack[i] = source->stack[i];
    }

    newTask->cpustate = source->cpustate;

    newTask->set_pid(next_pid());

    newTask->set_ppid(source->get_pid());

    for (unsigned int i = 0; i < sizeof(source->name); i++) {
        newTask->name[i] = source->name[i];
    }

    newTask->set_priority(source->get_priority());
    newTask->set_memory_alloc(source->get_memory_alloc());
    for (unsigned int i = 0; i < sizeof(source->open_files) / sizeof(source->open_files[0]); i++) {
        newTask->open_files[i] = source->open_files[i];
    }

    return newTask;
}

```

This method creates a new Task object that is a copy of the provided source process. This is used when forking a process.

```

Task* TaskManager::fork() {
    Task* currentTask = tasks[currentTask];
    Task* newTask = copyTask(currentTask);
    AddTask(newTask);
    return newTask;
}

```

And the fork method. Forks the currently running task, creating a new task with the same state as the current task, but with a different process ID (PID).

```

int TaskManager::WaitForProcess(int pid, int* status, int options) {

    Task* target = get_process_by_pid(pid);
    if (!target) {
        return -1;
    }

    Task* currentTask = GetCurrentTask();

    // Suspend the current task until the target process has completed
    while (target->getState() != ProcessState::TERMINATED) {
        asm("int $0x20"); // This will trigger a timer interrupt, which we will call TaskManager::Schedule
    }

    if (status) {
        *status = 0; // successful completion
    }

    // Return the PID of the terminated process
    return pid;
}

```

Waits for a process with the specified pid to change its state. It blocks the calling process until the specified process terminates or changes state based on the provided options. It is used for waitpid system calls.

```

Task* TaskManager::get_process_by_pid(int pid) {
    for (int i = 0; i < numTasks; i++) {
        if (tasks[i]->pid == pid) {
            return tasks[i];
        }
    }
    return nullptr;
}

Task *TaskManager::GetCurrentTask()
{
    if (currentTask >= 0 && currentTask < numTasks)
        return tasks[currentTask];
    else
        return nullptr;
}

```

First method simply returns Process itself with the specified pid. Second one returns the current process.

```

}
bool TaskManager::update_process_state(int pid, ProcessState state){
    for (int i = 0; i < numTasks; i++) {
        if (tasks[i]->pid == pid) {
            tasks[i]->state = state;
            return true;
        }
    }
    return false;
}
void TaskManager::printProcessTable()

```

This method updates the process state. Available States are READY, RUNNING, BLOCKED, TERMINATED.

```

void TaskManager::printProcessTable()
{
    printf("Process table:\n");
    for (int i = 0; i < numTasks; ++i) {
        Task *task = tasks[i];

        char task_number[2];
        task_number[0] = i + '0';
        task_number[1] = '\0';

        printf("Task ");
        printf(task_number);
        printf(": [State: ");

        if (i == currentTask) {
            printf("Running");
        } else {
            printf("Ready");
        }

        printf(", EIP: 0x");
        printfHex((task->cpustate->eip >> 4) & 0xFF);
        printfHex(task->cpustate->eip & 0xFF);
        printf(", ESP: 0x");
        printfHex((task->cpustate->esp >> 4) & 0xFF);
        printfHex(task->cpustate->esp & 0xFF);

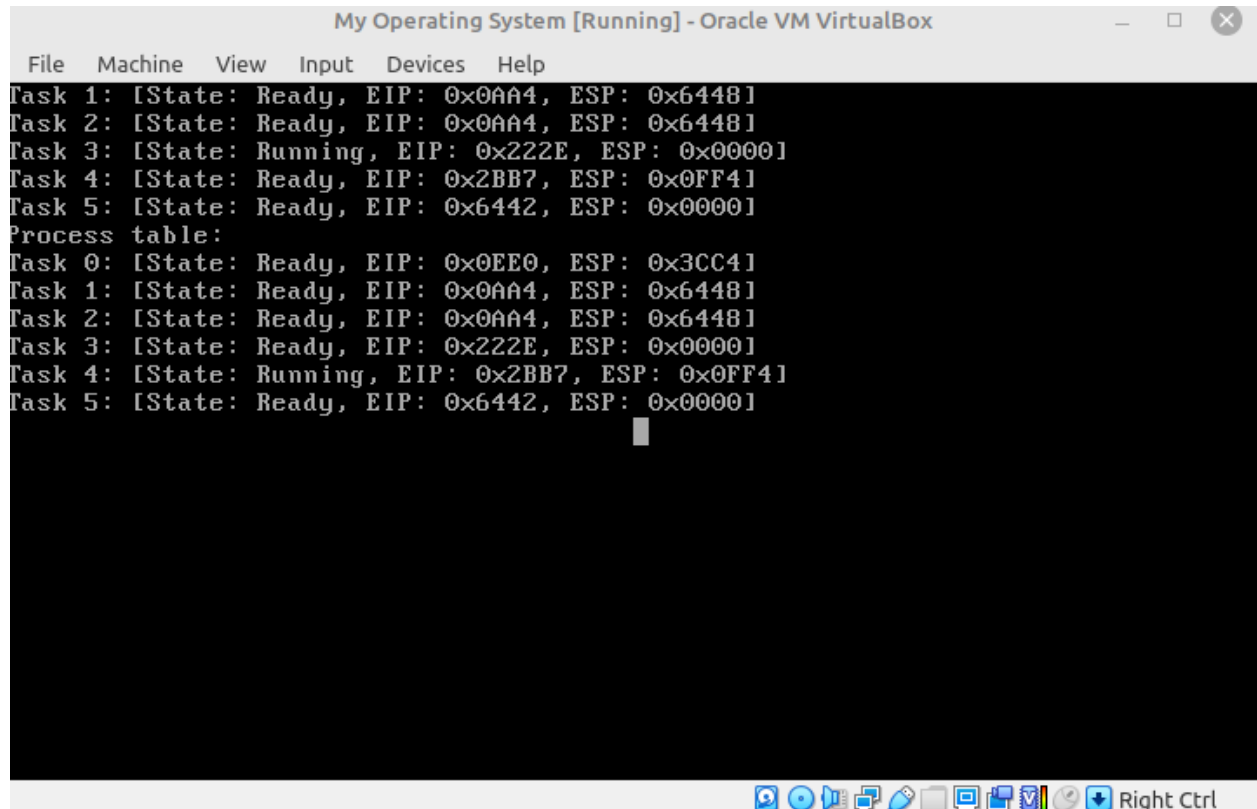
        printf("]\n");
    }
}

```



Above method prints a Process table with some properties. I will print all properties of a process in another method. This shows the process state and other things. “esp” (Extended Stack Pointer): points to the top of the stack in the memory. “eip” (Extended Instruction Pointer) : contains the memory address of the next instruction to be executed.

When I run the same 6 tasks, this table will be printed to the screens. As we can see 6 Process are initialized and fourth process is in Running state.



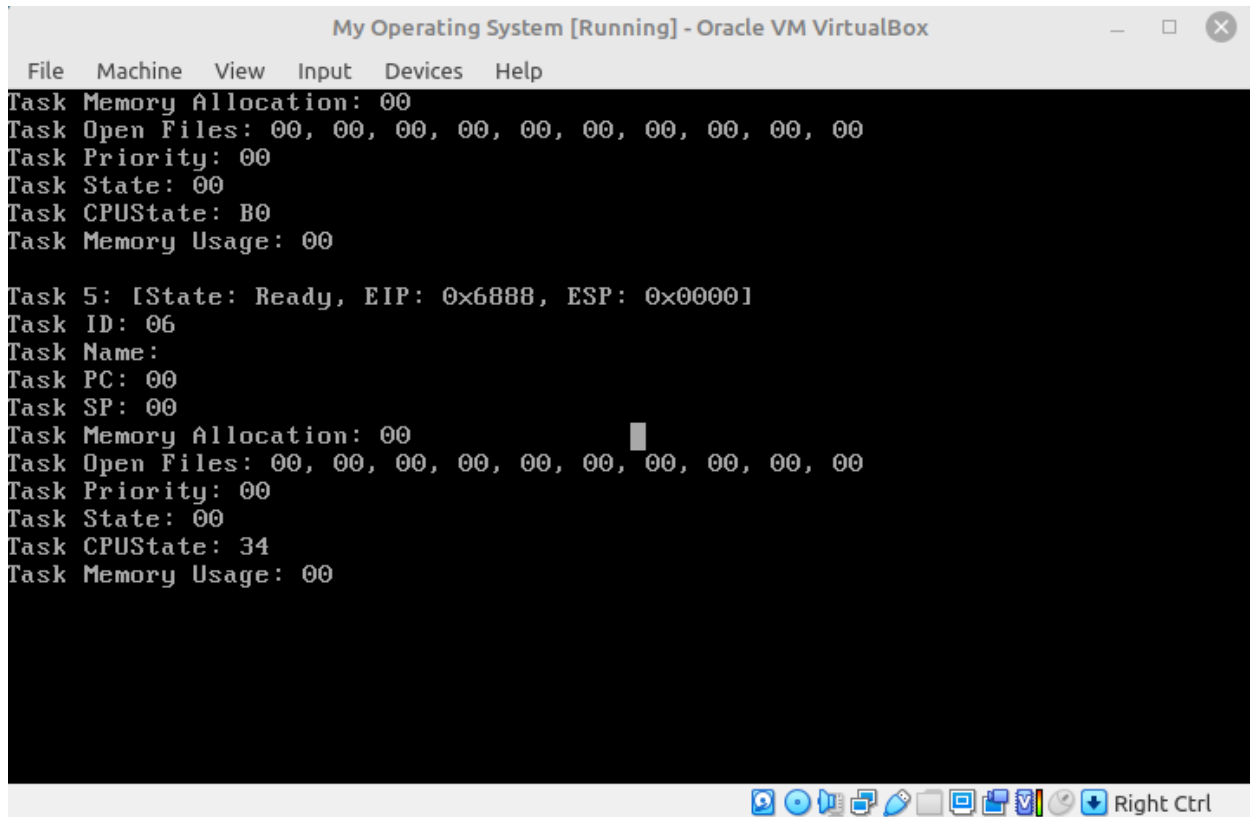
```
File  Machine  View  Input  Devices  Help
Task 1: [State: Ready, EIP: 0x0AA4, ESP: 0x6448]
Task 2: [State: Ready, EIP: 0x0AA4, ESP: 0x6448]
Task 3: [State: Running, EIP: 0x222E, ESP: 0x0000]
Task 4: [State: Ready, EIP: 0x2BB7, ESP: 0x0FF4]
Task 5: [State: Ready, EIP: 0x6442, ESP: 0x0000]
Process table:
Task 0: [State: Ready, EIP: 0x0EE0, ESP: 0x3CC4]
Task 1: [State: Ready, EIP: 0x0AA4, ESP: 0x6448]
Task 2: [State: Ready, EIP: 0x0AA4, ESP: 0x6448]
Task 3: [State: Ready, EIP: 0x222E, ESP: 0x0000]
Task 4: [State: Running, EIP: 0x2BB7, ESP: 0x0FF4]
Task 5: [State: Ready, EIP: 0x6442, ESP: 0x0000]
```

```

void TaskManager::printTaskProperties(Task* task) {
    printf("Task ID: ");
    printfHex(task->pid);
    printf("\nTask Name: ");
    printf(task->name);
    printf("\nTask PC: ");
    printfHex(task->pc);
    printf("\nTask SP: ");
    printfHex(task->sp);
    printf("\nTask Memory Allocation: ");
    printfHex(task->memory_alloc);
    printf("\nTask Open Files: ");
    for (int i = 0; i < 10; i++) {
        printfHex(task->open_files[i]);
        if (i != 9) printf(", ");
    }
    printf("\nTask Priority: ");
    printfHex(task->priority);
    printf("\nTask State: ");
    printfHex((uint8_t)task->state);
    printf("\nTask CPUState: ");
    printfHex((uint32_t)task->cpustate);
    printf("\nTask Memory Usage: ");
    printfHex(task->memoryUsage);
    printf("\n\n");
}

```

This is to print all properties of a process.



Task 4: "Handling Interrupts: Our given source code can handle interrupts, and your kernel will handle and respond between multiple processes."

To handler interrupts between multiple processes and also to make Task 5, I added Timer interrupt to my OS. I edited the HandleInterrupt method.

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

    if (interrupt == 0x20) {
        esp = ActiveInterruptManager->TimerInterrupt(esp);
    }

    return esp;
}

void InterruptManager::HandleInterruptRequest0x20()
```

If interrupt will be 0x20, then TimerInterrupt will work.

```
SetInterruptDescriptorTableEntry(0x80, CodeSegment, &HandleInterruptRequest0x80, 0, IDT_INTERRUPT_GATE);
SetInterruptDescriptorTableEntry(0x20, CodeSegment, &HandleInterruptRequest0x20, 0, IDT_INTERRUPT_GATE);
```

I added 0x20 to InterruptDescriptorTable in interrupts.cpp file like other interrupts which are 0x80.

```
static void HandleInterruptRequest0x80();
static void HandleInterruptRequest0x20();
```

I made same thing in the interrupts.h file.

```
void InterruptManager::HandleInterruptRequest0x20()
{
    if (ActiveInterruptManager != 0)
    {
        myos::common::uint32_t esp;

        asm("mov %%esp, %0" : "=r"(esp));
        ActiveInterruptManager->TimerInterrupt(esp);
    }
    asm("add $0x20, %esp");
    asm("iret");
}
```

In this method, Actually I made similar things with hardware interrupt. This function handles the timer interrupt by saving the stack pointer, calling the TimerInterrupt method to perform necessary actions, cleaning up the stack, and returning to the interrupted code. First assembly line retrieves the value of the ESP register, which is the stack pointer, and stores it in the esp variable. This is done using the mov instruction in inline assembly. The ESP register is important because it points to the top of the stack, which contains the CPU state that needs to be restored after handling the interrupt. After handling the interrupt, it adds a value of 0x20 to the stack pointer. This operation is necessary to clean up the stack from the interrupt handler's data before returning to the interrupted code. it returns from the interrupt using the “iret” instruction in inline assembly. This instruction restores the CPU state from the stack and resumes the execution of the interrupted code.

```

myos::common::uint32_t InterruptManager::TimerInterrupt(myos::common::uint32_t esp)
{
    timer_interrupt_count++;
    if (timer_interrupt_count % TIME_SLICE_THRESHOLD == 0)
    {
        taskManager->Schedule((CPUState *)esp);
    }
    return esp;
}

```

This function makes the timer interrupt indeed. It does the same thing with the “DoHandleInterrupt” method.

Task 5: “Perform Round Robin scheduling: Every time a timer interrupt occurs, there is a chance to make a process switch. Whenever a context scheduling occurs, you will print all the information about the processes in process table”

Actually our code already performs Round Robin scheduling.

```

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

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

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

    // Print process table information when a context switch occurs
    printProcessTable();

    return tasks[currentTask]->cpustate;
}

```

But I added state property to it. Below round-robin scheduling algorithm that takes into account the state of the tasks.

```

// Round Robin
CPUState *TaskManager::Schedule(CPUState *oldstate)
{
    if (numTasks <= 1) {
        return oldstate;
    }

    tasks[currentTask]->cpustate = oldstate;

    currentTask = (currentTask + 1) % numTasks;
    while (tasks[currentTask]->state != READY) {
        currentTask = (currentTask + 1) % numTasks;
    }
    // Print process table information when a context switch occurs
    printProcessTable();

    return tasks[currentTask]->cpustate;
}

```

It increments the current task index and wrap it around if it exceeds the number of tasks. If the next task is not in the READY state, continue incrementing the index and wrapping it around until a READY task is found. This step ensures that only tasks that are ready to run are considered for scheduling.

```

void TaskManager::printProcessTable()
{
    printf("Process table:\n");
    for (int i = 0; i < numTasks; ++i) {
        Task *task = tasks[i];

        char task_number[2];
        task_number[0] = i + '0';
        task_number[1] = '\0';

        printf("Task ");
        printf(task_number);
        printf(": [State: ");

        if (i == currentTask) {
            printf("Running");
        } else {
            printf("Ready");
        }

        printf(", EIP: 0x");
        printfHex((task->cpustate->eip >> 4) & 0xFF);
        printfHex(task->cpustate->eip & 0xFF);
        printf(", ESP: 0x");
        printfHex((task->cpustate->esp >> 4) & 0xFF);
        printfHex(task->cpustate->esp & 0xFF);

        printf("]\n");
        printTaskProperties(tasks[i]);
    }
}

```

This method prints all properties of a process.

```

void TaskManager::printTaskProperties(Task* task) {
    printf("Task ID: ");
    printfHex(task->pid);
    printf("\nTask Name: ");
    printf(task->name);
    printf("\nTask PC: ");
    printfHex(task->pc);
    printf("\nTask SP: ");
    printfHex(task->sp);
    printf("\nTask Memory Allocation: ");
    printfHex(task->memory_alloc);
    printf("\nTask Open Files: ");
    for (int i = 0; i < 10; i++) {
        printfHex(task->open_files[i]);
        if (i != 9) printf(", ");
    }
    printf("\nTask Priority: ");
    printfHex(task->priority);
    printf("\nTask State: ");
    printfHex((uint8_t)task->state);
    printf("\nTask CPUState: ");
    printfHex((uint32_t)task->cpustate);
    printf("\nTask Memory Usage: ");
    printfHex(task->memoryUsage);
    printf("\n\n");
}

```

And this method is used in the previous method. This prints for a process.

## Task 6: "Lifecycle"

First Strategy: It initializes the process table with loading 3 different tasks. the code should enter an infinite loop until all the processes terminate. This is achieved by having the main kernel loop continuously, and the scheduler selects and runs the processes in the background. When all processes have terminated, the main loop will break, and the kernel will exit.



```

void strategy1(GlobalDescriptorTable &gdt, TaskManager &taskManager) {
    Task task1(&gdt, BinarySearch);
    Task task2(&gdt, LinearSearch);
    Task task3(&gdt, Collatz);
    taskManager.AddTask(&task1);
    taskManager.AddTask(&task2);
    taskManager.AddTask(&task3);
}

```

Second Strategy: This part loads it into memory ten times, starting each process.

```

void strategy2(GlobalDescriptorTable &gdt, TaskManager &taskManager) {
    int chosenProgram = sysgetrandom(3) % 3;
    for (int i = 0; i < 10; ++i) {
        if (chosenProgram == 0) {
            Task task(&gdt, BinarySearch);
            taskManager.AddTask(&task);
        } else if (chosenProgram == 1) {
            Task task(&gdt, LinearSearch);
            taskManager.AddTask(&task);
        } else {
            Task task(&gdt, Collatz);
            taskManager.AddTask(&task);
        }
    }
}

```

Third strategy: It randomly chooses two out of three programs and loads each program three times.

```
void strategy3(GlobalDescriptorTable &gdt, TaskManager &taskManager) {
    int program1 = sysgetrandom(3) % 3;
    int program2;
    do {
        program2 = sysgetrandom(3) % 3;
    } while (program2 == program1);

    for (int i = 0; i < 3; ++i) {
        if (program1 == 0 || program2 == 0) {
            Task task(&gdt, BinarySearch);
            taskManager.AddTask(&task);
        }
        if (program1 == 1 || program2 == 1) {
            Task task(&gdt, LinearSearch);
            taskManager.AddTask(&task);
        }
        if (program1 == 2 || program2 == 2) {
            Task task(&gdt, Collatz);
            taskManager.AddTask(&task);
        }
    }
}
```

I choose one of the strategies with this part of code:

```
int chosenStrategy = sysgetrandom(3) % 3;

if (chosenStrategy == 0) {
    strategy1(gdt, taskManager);
} else if (chosenStrategy == 1) {
    strategy2(gdt, taskManager);
} else {
    strategy3(gdt, taskManager);
}
```

And my Binary Search, Linear Search and Collatz algorithms:

```
int binarySearch(int arr[], int size, int x) {
    int left = 0;
    int right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == x)
            return mid;

        if (arr[mid] < x)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}

void BinarySearch() {
    while(true){
        int arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170};
        int n = sizeof(arr) / sizeof(arr[0]);
        int x = 110;
        int result = binarySearch(arr, n, x);
        sysprintf(int_to_str(result));
    }
}
```

```
int linearSearch(int arr[], int size, int x) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == x)
            return i;
    }
    return -1;
}

void LinearSearch() {
    while(true){
        int arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170};
        int n = sizeof(arr) / sizeof(arr[0]);
        int x = 175;
        int result = linearSearch(arr, n, x);
        sysprintf(int_to_str(result));
    }
}
```

```

void Collatz() {
while(true){
    for (int n = 1; n <= 25; n++) {

        while (n != 1) {
            sysprintf(int_to_str(n));
            sysprintf(": ");
            if (n % 2 == 0)
                n = n / 2;
            else
                n = 3 * n + 1;
        }
        sysprintf(int_to_str(n));
        sysprintf("\n");
    }
}
}

```

## Conclusion

I made all the parts but I couldn't test them. The big problem was to implement a Timer interrupt. I was getting some errors and I couldn't solve it. I had to use it in Round Robin but I couldn't. I can print process tables etc. I made system calls except for getrandom. It always gives number 7. I didn't understand why. And I couldn't make an execve system call. I don't know how to replace the core image of a process.