



**GEBZE TECHNICAL UNIVERSITY**  
**DEPARTMENT OF COMPUTER ENGINEERING**

**CSE 312 /CSE504**  
**Operating Systems**  
**Homework #1 Report**

**171044073**  
**Berkan AKIN**

# 1.0 REQUIREMENTS

## 1.1 Part 1: General Requirements

This section outlines the general requirements necessary to complete the assignment. Follow these steps to successfully complete the assignment:

**System Calls:** Implement POSIX-compliant system calls such as fork, waitpid, execve, and any other necessary POSIX calls.

**Process Creation:** All processes must be created using the fork system call.

**Program Loading:** The kernel must be capable of loading multiple programs into memory, including multiple instances of the same program.

**Multitasking:** Develop a Process Table to hold necessary information about the processes in memory. This table should include the process state, identifiers, and other essential data.

**Interrupt Handling:** The provided source code can handle interrupts, and the kernel must handle and respond to interrupts across multiple processes.

## 1.2 Part 2: Scheduling Strategies

This section details the process scheduling strategies based on timer interrupts:

**Round Robin Scheduling:** On each timer interrupt, a process switch should be made. During every context switch, print all information about the processes in the process table. You can modify the table as needed, provided you document these changes in your report.

**Priority-Based (Preemptive) Scheduling:** When a new process arrives in the ready queue, its priority should be compared with those of the existing processes. The process with the highest priority (lowest number) should be given CPU next.

## 1.3 Part 3: Microkernel Strategies

Microkernel strategies offer different methods for process loading and scheduling:

**Random Loading:** In this strategy, one of the programs is randomly selected and loaded into memory 10 times (creating 10 different processes), started, and the system enters an infinite loop until all processes terminate.

**Dual Program Loading:** In the second strategy, two out of four programs are randomly selected, each loaded three times, started, and the system enters an infinite loop until all processes terminate.

Priority and Dynamic Priority Strategy: In the third strategy, the init process initializes the Process Table and ready queue. The Collatz program starts with the lowest priority and, after a specified number of interrupts (e.g., every 5th interrupt), its priority is dynamically increased. Other programs maintain their initial priorities.

#### **1.4 Part 4: Test Programs**

The programs used to test the assignment are as follows:

Collatz Sequence: Find the Collatz sequence for every number less than 100. The input can be taken from the user or passed as a parameter.

BinarySearch: Binary search algorithm. For example, in the array {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}, if  $x = 110$ , the output should be 6.

LinearSearch: Linear search algorithm. For example, in the array {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}, if  $x = 175$ , the output should be -1.

Long-Running Program: The `long_running_program(n)` function. This function should be run without user interaction and should return the result of its computation.

## 2.0 Completion of Requirements for Part A

In Part A of the homework, the primary focus was on implementing foundational features of a custom operating system. The requirements included the implementation of critical POSIX system calls such as `fork`, `waitpid`, and `execve`. Additionally, the operating system needed to support the loading of multiple programs into memory and manage these processes through a comprehensive Process Table. This table was required to store essential information about each process, such as its state, identifiers, and other critical data. The kernel was also expected to handle interrupts and manage context switching between processes using Round Robin scheduling.

To meet these requirements, we started by implementing the `fork` system call, allowing for the creation of child processes. The `waitpid` and `execve` system calls were then added to handle process synchronization and program execution, respectively. A Process Table was developed to keep track of all processes, storing their states, identifiers, and other necessary information. We ensured that the kernel could load multiple instances of programs into memory, facilitating multitasking. Furthermore, interrupt handling was integrated, enabling the kernel to respond to various interrupts and perform context switches as needed.

### Implementation Steps

The implementation process involved several key steps:

#### System Call Implementation:

- **Fork:** Implemented to create new processes. The `fork` system call duplicates the calling process, creating a new process (child) with a unique process ID.
- **Waitpid:** Added to allow a process to wait for its child processes to change state, ensuring proper synchronization.
- **Execve:** Implemented to execute a new program within a process, replacing the current process image with a new one.

#### Process Table Development:

- Designed a structure to store essential information about each process, including process IDs, states, parent-child relationships, and other relevant data.
- Ensured the table could handle dynamic additions and removals of processes.

#### Memory Management:

- Developed mechanisms to load multiple programs into memory, allowing for the execution of different instances of the same program.
- Implemented memory allocation and deallocation procedures to manage the processes efficiently.

#### Interrupt Handling:

- Integrated interrupt handling capabilities to respond to various hardware and software interrupts.

- Implemented context switching mechanisms to switch between processes during interrupts, adhering to Round Robin scheduling.

### Testing and Debugging:

- Conducted extensive testing with programs such as Collatz sequence calculations, binary search, and long-running computational tasks.
- Debugged and optimized the implementation to ensure stability and efficiency.

With these steps completed, Part A of the homework successfully established a robust foundation for the custom operating system, setting the stage for further enhancements in subsequent parts. The next sections of the report will provide detailed explanations of each implementation step, including design decisions, code snippets, and screenshots demonstrating the system's functionality.

## 3.0 Part A Implementation

First, the **getpid()** system call was implemented, followed by **execve()**, **waitpid()**, **exit()**, **fork()**, and **setpriority()** system calls. Each system call was utilized by generating an interrupt and handling it through an interrupt handler. The implementation of these system calls will be explained step by step.

- Programs operate by giving a program to the TaskManager and running it through there.

```
//Task task1(&gdt, initProcess2);
Task task2(&gdt, initProcess);
//Task task3(&gdt, firstStrategy);
//Task task4(&gdt, secondStrategy);
//Task task5(&gdt, initProcessPartC);

//taskManager.AddTask(&task1);
taskManager.AddTask(&task2);
//taskManager.AddTask(&task3);
//taskManager.AddTask(&task4);
//taskManager.AddTask(&task5);
```

### 3.1 getpid() Implementation

- At the assembly code level, we generate an interrupt and load the enum value into the eax register. After the interrupt is generated, we use this value to identify which system call created the interrupt. We use the ecx register to carry the return value.

- 

```
21 int myos::getPid()
22 {
23     int pid=-1;
24     asm("int $0x80" : "=c" (pid) : "a" (SYSCALLS::GETPID));
25     return pid;
26 }
27
```

- We catch the interrupt here. We try to access the interrupt manager from within the InterruptManager. We write the returned value to the ecx register.

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

    switch(cpu->eax)
    {
        case GETPID:
            cpu->ecx = interruptManager->getPidInterrupt();
            break;
    }
}
```

- We can access the TaskManager from the InterruptManager. This way, we can access the PID value of the current task

```
uint32_t InterruptManager::getPidInterrupt(){
    return taskManager->getCurrentTaskPid();
}
```

```
common::uint32_t TaskManager::getCurrentTaskPid(){
    return tasks[currentTask]->pid;
}
```

### 3.1.1 Test And Result

- As seen in the code below, fork is performed. The output printed on the screen shows that the child PID and parent PID are different.

```
void initProcess(){
    uint32_t num = getPid();
    printf("Init Process PID:");
    printfInt(num);

    int pid = fork();
    printf("    Fork PID:");
    printfInt(pid);
    printf("\n");
    if(pid == 0){
        int getPidChild = getPid();
        printf("Child pid:");
        printfInt(getPidChild);
        printf("\n");
        fork();
        setPriority(getPid(), 5);
        fork();
        execve(CollatzSequence);
    }
    else{
        int getPidParent = getPid();
        printf("Parent PID:");
        printfInt(getPidParent);
        printf("\n");
        //waitpid(2);
        fork();
        fork();
        execve(long_running_program);
    }
    while(true);
}
```

PID	Parent PID	Priority	EIP	State
1	0	1	1088147	Running

Init Process PID:1      Fork PID:1  
Parent PID:1  
Number of tasks: 4      Current tasks: 2

PID	Parent PID	Priority	EIP	State
1	0	1	1088429	Ready
2	1	1	1055432	Running
3	1	1	1055432	Ready
4	1	1	1055432	Ready

Fork PID:0  
Child pid:2  
KEYBOARD 0x1DKEYBOARD 0x38

### 3.2 execve() Implementation

- When an interrupt is generated, the eax register is used to indicate which system call it is, the ebx register carries the new task to be assigned, and the ecx register carries the return value.

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

- This is where the interrupt is caught. From within the InterruptManager, we try to access the TaskManager. This allows us to access the task class and assign a new value to the EIP register. The new value is carried into the TaskManager along with the CPU state.

```
case EXECVE:
    esp = (uint32_t) interruptManager->interruptExec(cpu);
    //return InterruptHandler::HandleInterrupt(esp);
    break;
```

```
CPUState* InterruptManager::interruptExec(CPUState* cpustate){
    return taskManager->makeExec(cpustate);
}
```

- Here, we reset the task's registers and assign the new task value to the EIP register.

```
CPUState* TaskManager::makeExec(CPUState* cpustate){
    //yeni yer açmadanda çalışıyor
    tasks[currentTask]->cpustate = (CPUState*)(tasks[currentTask]->stack + 4096 - sizeof(CPUState));

    //entry point is in ebx
    tasks[currentTask]->cpustate -> eip = cpustate -> ebx;
    tasks[currentTask]->cpustate -> ecx = taskManager[currentTask].getCurrentTaskPid();
    tasks[currentTask]->cpustate -> eax = 0;
    tasks[currentTask]->cpustate -> ebx = 0;
    tasks[currentTask]->cpustate -> edx = 0;

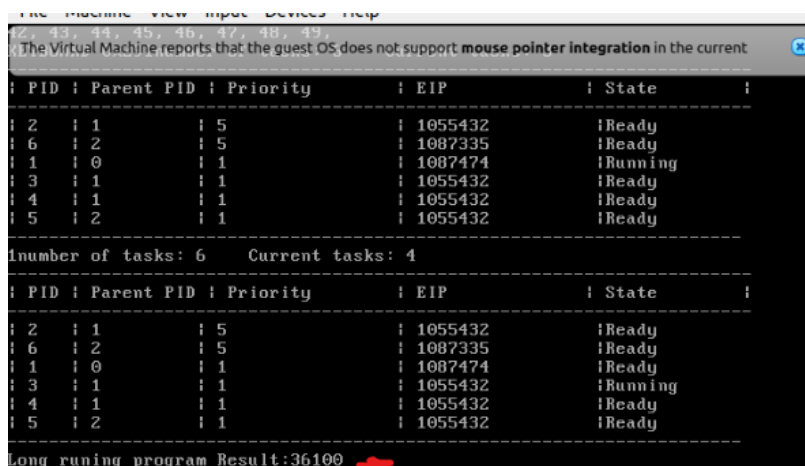
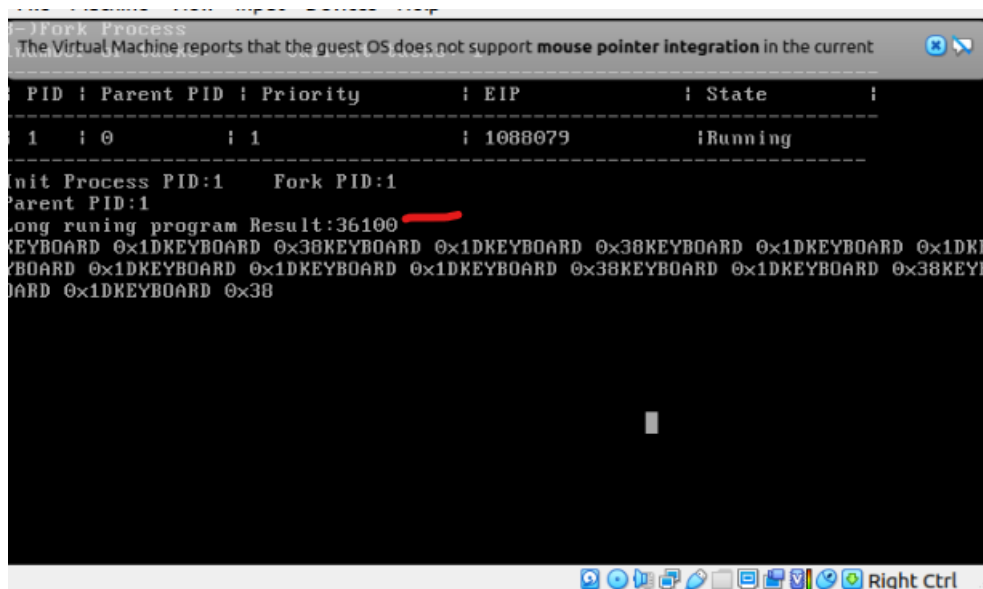
    tasks[currentTask]->cpustate -> esi = 0;
    tasks[currentTask]->cpustate -> edi = 0;
    tasks[currentTask]->cpustate -> ebp = 0;
    tasks[currentTask]->processState = Ready;
    tasks[currentTask]->cpustate->cs = gdt->CodeSegmentSelector();
    tasks[currentTask]->cpustate->eflags = 0x202;

    return tasks[currentTask]->cpustate;
}
```

### 3.2.1 Test And Result

```
// For First Part
void initProcess(){
    uint32_t num = getpid();
    printf("Init Process PID:");
    printfInt(num);

    int pid = fork();
    printf("    Fork PID:");
    printfInt(pid);
    printf("\n");
    if(pid == 0){
        int getPidChild =getPid();
        printf("Child pid:");
        printfInt(getPidChild);
        printf("\n");
        fork();
        setPriority(getPid(),5);
        fork();
        execve(CollatzSequence);
    }
    else{
        int getPidParent =getPid();
        printf("Parent PID:");
        printfInt(getPidParent);
        printf("\n");
        //waitpid(2);
        fork();
        fork();
        execve(long_running_program);
    }
    while(true);
}
```





### 3.3 waitpid() Implementation

- When the waitpid function is called, it makes a syscall and generates an interrupt. If the input value is -1, it attempts to implement the ability to wait for all child processes, but this **was not accomplished**. The ability to wait for the process with the given PID was implemented. The PID value is carried to the InterruptManager using the ebx register.

```
int myos::waitpid(int pid){
    int rValue = 0;
    int tmpPid = pid;
    if(pid == -1){
        asm("int $0x80" : "=c" (rValue) : "a" (SYSCALLS::WAITPIDALL));
    }
    else{
        asm("int $0x80"
            : "=c" (rValue)          // çıktı: ecx kaydedicisine sonuç değeri
            : "a" (SYSCALLS::WAITPIDONE), "b" (pid) // girdi: eax'e syscall numarası, ebx'e pid değeri
            :);
    }
    return rValue;
}
```

- After the process is put into a wait position, a context switch is performed.

```
case WAITPIDONE:
    esp = (uint32_t) interruptManager->setOneProcess(cpu);
    esp = interruptManager->scheduleTask((CPUState*)esp);

    break;
```

```
CPUState* InterruptManager::setOneProcess(CPUState* cpu){
    return taskManager->waitOneProcess(cpu);
}
```

- Which process to wait for is assigned to the waitProcessPid variable using the ebx register. The process state is set to Wait. It prevents a process from waiting for itself and from waiting if the process has already finished.

```

CPUState* TaskManager::waitOneProcess(CPUState* cpustate){
    int pid = cpustate->ebx;

    if(numTasks<pid){
        return cpustate;
    }

    int index = getProcessIndex(pid);
    if(index == -1){
        return cpustate;
    }

    if(tasks[index]->processState == Finish){
        return cpustate;
    }

    if(tasks[currentTask]->pid == pid ){
        return cpustate;
    }

    tasks[currentTask]->cpustate = cpustate;
    tasks[currentTask]->waitpPidProcess = cpustate->ebx;
    tasks[currentTask]->processState = Wait;

    return tasks[currentTask]->cpustate;
}

```

### 3.3.1 Test And Results

As seen in the code, the parent process is instructed to wait for process number 2. This is done by calling waitpid with the argument set to 2. First, the Collatz programs are run, and then the long-running program is executed because the parent process is in a waiting state. This is shown in the screen output.

```

// For First Part
void initProcess(){
    uint32_t num = getPid();
    printf("Init Process PID:");
    printfInt(num);

    int pid = fork();
    printf("    Fork PID:");
    printfInt(pid);
    printf("\n");
    if(pid == 0){
        int getPidChild =getPid();
        printf("Child pid:");
        printfInt(getPidChild);
        printf("\n");
        fork();
        setPriority(getPid(),5);
        fork();
        execve(CollatzSequence);
    }
    else{
        int getPidParent =getPid();
        printf("Parent PID:");
        printfInt(getPidParent);
        printf("\n");
        waitpid(2);
        fork();
        fork();
        execve(long_running_program);
    }
    while(true);
}

```

```

4. deneme [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
number of tasks: 1    Current tasks: 1
The Virtual Machine reports that the guest OS does not support mouse pointer integration in the current
PID | Parent PID | Priority | EIP | State |
-----
1 | 0 | 1 | 1088084 | Running
Init Process PID:1    Fork PID:1
Parent PID:1
number of tasks: 2    Current tasks: 2
PID | Parent PID | Priority | EIP | State |
-----
1 | 0 | 1 | 1055349 | Wait
2 | 1 | 1 | 1055432 | Running
Fork PID:0
Child pid:2
Collatz:2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 49,
KEYBOARD 0x1DKEYBOARD 0x38

```

```

-----
Collatz:2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 49,
number of tasks: 5    Current tasks: 3
PID | Parent PID | Priority | EIP | State |
-----
1 | 0 | 1 | 1055349 | Wait
2 | 1 | 5 | 1087339 | Ready
3 | 2 | 5 | 50 | Running
4 | 2 | 5 | 1055432 | Ready
5 | 3 | 5 | 1055432 | Ready
KEYBOARD 0x1DKEYBOARD 0x1DKEYBOARD 0x38

```

```

5 | 3 | 5 | 1055432 | Ready
The Virtual Machine reports that the guest OS does not support mouse pointer integration in the current
Collatz:2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 49,
number of tasks: 5    Current tasks: 4
PID | Parent PID | Priority | EIP | State |
-----
1 | 0 | 1 | 1055349 | Wait
2 | 1 | 5 | 1087339 | Ready
3 | 2 | 5 | 1087339 | Ready
4 | 2 | 5 | 50 | Running
5 | 3 | 5 | 1055432 | Ready
KEYBOARD 0x1DKEYBOARD 0x38

```

- After process 2 finishes executing, it can be seen that the process forked by the parent process has also finished.

```

4. deneme [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
1 0 1 1055392 iFinish
The Virtual Machine reports that the guest OS does not support mouse pointer integration in the current
3 2 5 1055392 iFinish
4 2 5 1055392 iFinish
5 3 5 1055392 iFinish
6 1 1 1055392 iFinish
7 1 1 1055392 iFinish
8 6 1 1055432 iRunning
-----
Long runing program Result:36100
number of tasks: 8 Current tasks: 1
-----
PID Parent PID Priority EIP State
1 0 1 1055392 iFinish
2 1 5 1055392 iFinish
3 2 5 1055392 iFinish
4 2 5 1055392 iFinish
5 3 5 1055392 iFinish
6 1 1 1055392 iFinish
7 1 1 1055392 iFinish
8 6 1 1055392 iFinish
-----
All Process Finished

```

- **Note:** To take screenshots, the exit function was commented out and then uncommented. Therefore, in the initial screenshots, the processes did not finish.

### 3.4 exit() Implementation

- When the exit function makes a syscall, an interrupt is generated. It sets itself to the finished state. Therefore, scheduling does not run that task again.

```

int myos::exit(){
    int status =-1;
    asm("int $0x80" : "=c" (status) : "a" (SYSCALLS::EXIT));
    return status;
}

```

```

case EXIT:
    if(interruptManager->exitInterrupt(cpu) == true){
        cpu->ecx = 0;
        esp =interruptManager->scheduleTask(cpu);
        //return InterruptHandler::HandleInterrupt(esp);
    }
    break;

```

- The generated interrupt is caught, and with the eax register indicating that the syscall is exit, it enters the appropriate switch case. Here, the TaskManager is accessed, the currentTask's processState is marked as Finish, and finally, scheduling is done to run the next process.

```

bool InterruptManager::exitInterrupt(CPUState* cpustate){
    return taskManager->setCurrentTaskFinished(cpustate);
}

```

```

bool TaskManager::setCurrentTaskFinished(CPUState* cpustate){
    tasks[currentTask]->cpustate = cpustate;
    tasks[currentTask]->processState = Finish;

    int i=0;

    return true;
}

```

### 3.4.1 Test and Result

- The Collatz and long-running programs, shown in the screenshots below, will transition to the exit state upon completion, allowing the next program to be scheduled. The provided screenshot of the process table shows the final state.

```

// For First Part
void initProcess(){
    uint32_t num = getpid();
    printf("Init Process PID:");
    printfInt(num);

    int pid = fork();
    printf(" Fork PID:");
    printfInt(pid);
    printf("\n");
    if(pid == 0){
        int getPidChild =getPid();
        printf("Child pid:");
        printfInt(getPidChild);
        printf("\n");
        fork();
        setPriority(getPid(),5);
        fork();
        execve(CollatzSequence);
    }
    else{
        int getPidParent =getPid();
        printf("Parent PID:");
        printfInt(getPidParent);
        printf("\n");
        //waitpid(2);
        fork();
        fork();
        execve(long_running_program);
    }
    while(true);
}

```

```

void CollatzSequence() {
    int i=2;
    int number = 2;
    printf("Collatz:");
    while(i<50){
        number = i;
        printfInt(number);
        printf(", ");
        while (number != 1) {
            if (number % 2 == 0) {
                number /= 2;
            } else {
                number = 3 * number + 1;
            }
            //printfInt(number);
            //printf(", ");
        }
        i++;
        //printfInt(number);
    }
    printf("\n");

    exit();
}

```

```

void long_running_program() {
    int n=20;
    long long result = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            result += i * j;
        }
    }
    printf("Long runing program Result:");
    printfInt(result);
    printf("\n");
    exit();
}

```

```

4. deneme [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
! 4 ! 1 ! 1 ! 1055392 !Finish
! 5 ! 2 ! 5 ! 1055392 !Finish
! 6 ! 2 ! 5 ! 1055392 !Finish
! 7 ! 3 ! 1 ! 1055392 !Finish
! 8 ! 5 ! 5 ! 1055432 !Running
-----
Collatz:2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 49,
number of tasks: 8 Current tasks: 1
-----
! PID ! Parent PID ! Priority ! EIP ! State !
-----
! 1 ! 0 ! 1 ! 1055392 !Finish
! 2 ! 1 ! 5 ! 1055392 !Finish
! 3 ! 1 ! 1 ! 1055392 !Finish
! 4 ! 1 ! 1 ! 1055392 !Finish
! 5 ! 2 ! 5 ! 1055392 !Finish
! 6 ! 2 ! 5 ! 1055392 !Finish
! 7 ! 3 ! 1 ! 1055392 !Finish
! 8 ! 5 ! 5 ! 1055392 !Finish
-----
All Process Finished

```

**Note:** Due to the program running quickly, this is the only screenshot I was able to capture.

### 3.5 fork() Implementation

- When a fork call is made, a process is duplicated, creating an additional process. After the fork call, the return values of the fork function are different for the parent and child processes. Following the fork syscall, both processes continue to execute from the same point.
- An interrupt is generated and handled. The eax register is used to determine which syscall was made. The ecx register is set to 0 for the child process and the parent's own PID value for the parent process.

```

int myos::fork(){
    int pid = -1;
    asm("int $0x80" : "=c" (pid) : "a" (SYSCALLS::FORK));
    return pid;
}

```

```

78     case FORK:
79         cpu->ecx=interruptManager->createProcess(cpu);
80         //return InterruptHandler::HandleInterrupt(esp);
81         break;

```

```

int InterruptManager::createProcess(CPUState* cpustate){
    return taskManager->makeFork(cpustate);
}

```

1. `tasks[numTasks] = new Task(gdt, nullptr);`
  - A new task (**Task**) is created and added to the **tasks** array at the index **numTasks**.

2. **for (int i = 0; i < sizeof(tasks[currentTask]->stack); i++) tasks[numTasks]->stack[i] = tasks[currentTask]->stack[i];**
  - The stack of the current task is copied to the new task. This ensures that the new task shares the same state as the current task.
3. **common::uint32\_t currentTaskOffset = (((common::uint32\_t)cpustate - (common::uint32\_t)tasks[currentTask]->stack));**
  - The offset of the current task's CPU state within the stack is calculated. This helps in setting the same stack state for the new task.
4. **tasks[numTasks]->cpustate = (CPUState)((common::uint32\_t)tasks[numTasks]->stack) + currentTaskOffset);\***
  - The CPU state of the new task is set using the calculated offset.
5. **tasks[numTasks]->cpustate->ecx = 0;**
  - The **ecx** register of the new task is set to 0. This is the return value of **fork** for the child process.
6. **tasks[numTasks]->cpustate->ebp = tasks[numTasks]->stack + cpustate->ebp - tasks[currentTask]->stack;**
  - The base pointer (**ebp**) of the new task's stack is set.
7. **tasks[numTasks]->processState = Ready;**
  - The state of the new task is marked as 'Ready'. This indicates that the new task is ready to be executed.
8. **tasks[numTasks]->Ppid = tasks[currentTask]->pid;**
  - The parent PID of the new task is set to the PID of the current task.
9. **tasks[numTasks]->pid = numTasks + 1;**
  - The PID of the new task is set based on the current number of tasks.
10. **tasks[numTasks]->setPriority(tasks[currentTask]->getPriority());**
  - The priority of the new task is set to the priority of the current task.
11. **return tasks[currentTask]->pid;**
  - The function returns the PID of the current task. This is the return value of **fork** for the parent process.

```

int TaskManager::makeFork(CPUState* cpustate){
    if(numTasks >= 256)
        return 0;

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

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

    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);

    //Burada ECX' yeni taskın process id'sini atıyorum. Syscall'a return edebilmek için.
    tasks[numTasks]->cpustate->ecx = 0;
    tasks[numTasks]->cpustate->ebp = tasks[numTasks]->stack + cpustate->ebp - tasks[currentTask]->stack;
    tasks[numTasks]->processState=Ready;
    tasks[numTasks]->Ppid=tasks[currentTask]->pid;
    tasks[numTasks]->pid=numTasks+1;
    tasks[numTasks]->setPriority(tasks[currentTask]->getPriority());

    numTasks++;

    return tasks[currentTask]->pid;
}

```

### 3.5.1 Test and Results

When the init process is forked and different tasks are assigned using execve, it can be seen that the processes are running. Screenshots related to the tasks are provided below

```

// For First Part
void initProcess(){
    uint32_t num = getPid();
    printf("Init Process PID:");
    printfInt(num);

    int pid = fork();
    printf("    Fork PID:");
    printfInt(pid);
    printf("\n");
    if(pid ==0){
        int getPidChild =getPid();
        printf("Child pid:");
        printfInt(getPidChild);
        printf("\n");
        fork();

        setPriority(getPid(),5);
        fork();
        execve(CollatzSequence);

    }
    else{
        int getPidParent =getPid();
        printf("Parent PID:");
        printfInt(getPidParent);
        printf("\n");
        //waitpid(2);
        fork();
        fork();
        execve(long_running_program);
    }
    while(true);
}

```



- The screen shows which parent each program was copied from and their final states. Each program runs in sequence. As a result, multiple processes have been created from a single init process.

```

File Machine View Input Devices Help
-----
1 0 1 1055392 iFinish
The Virtual Machine reports that the guest OS does not support mouse pointer integration in the current
3 2 5 1055392 iFinish
4 2 5 1055392 iFinish
5 3 5 1055392 iFinish
6 1 1 1055392 iFinish
7 1 1 1055392 iFinish
8 6 1 1055432 iRunning

-----
Long running program Result:36100
number of tasks: 8 Current tasks: 1

-----
PID Parent PID Priority EIP State
-----
1 0 1 1055392 iFinish
2 1 5 1055392 iFinish
3 2 5 1055392 iFinish
4 2 5 1055392 iFinish
5 3 5 1055392 iFinish
6 1 1 1055392 iFinish
7 1 1 1055392 iFinish
8 6 1 1055392 iFinish

-----
All Process Finished
  
```

**Note:** Since the output screen passed quickly, the step-by-step process outputs could not be captured.

### 3.6 setpriority() Implementation

- We can change the priority of processes by making a syscall. With priority scheduling, we run tasks based on their priority. The first parameter specifies which task's priority we want to change, and the second parameter sets the priority level between 1 and 10. We pass the PID value to the TaskManager using the ebx register and the priority value using the edx register.

```

int myos::setPriority(int which, int prio){
    int result=-1;
    asm("int $0x80" : "=c" (result) : "a" (SYSCALLS::PRIORITY), "b" (which), "d" (prio) : );
    return result;
}
  
```

```

case PRIORITY:
    cpu->eax= interruptManager->setPriorityIM(cpu);
    esp = (uint32_t)cpu;
    break;
  
```

```

int InterruptManager::setPriorityIM(CPUState* cpustate){
    return taskManager->setPriorityTaskManager(cpustate);
}
  
```

- The information carried by the ebx and edx registers is assigned to the task with the given PID. If the assignment is successful, that value is returned.

```

int TaskManager::setPriorityTaskManager(CPUState* cpustate){
    int pid = cpustate->ebx;
    int priority= cpustate->edx;

    int index = getProcessIndex(pid);
    if(index == -1){
        cpustate->ecx = -1;
        return -1;
    }
    cpustate->ecx = 1;

    tasks[index]->setPriority(priority);

    return 0;
}

```

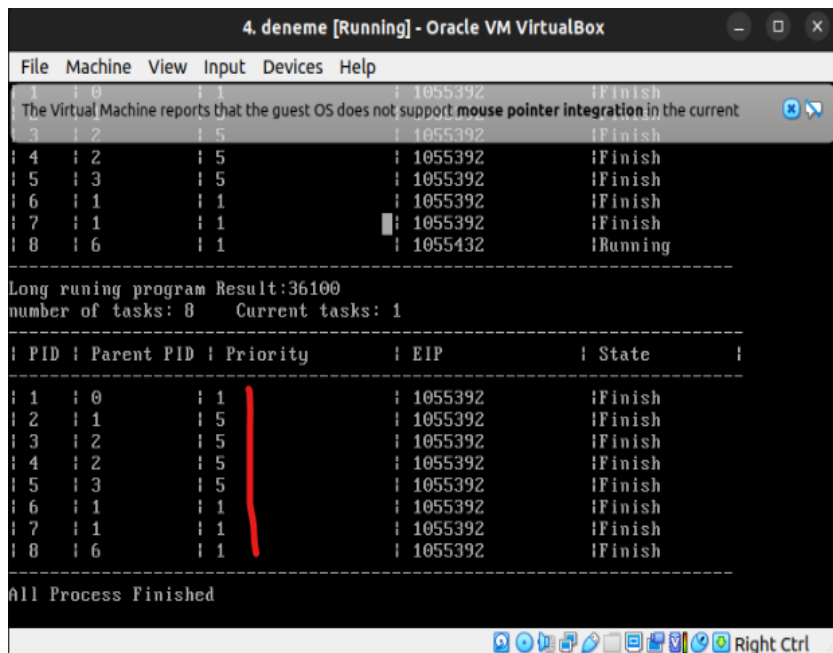
### 3.6.1 Test and Result

"In the code below, it is shown in the process table that the PID of the processes created by the child process is 5."

```

3 void initProcess(){
4     uint32_t num = getpid();
5     printf("Init Process PID:");
6     printfInt(num);
7
8     int pid = fork();
9     printf("    Fork PID:");
10    printfInt(pid);
11    printf("\n");
12    if(pid == 0){
13        int getPidChild =getPid();
14        printf("Child pid:");
15        printfInt(getPidChild);
16        printf("\n");
17        fork();
18        setPriority(getPid(),5);
19        fork();
20        execve(CollatzSequence);
21    }
22    else{
23        int getPidParent =getPid();
24        printf("Parent PID:");
25        printfInt(getPidParent);
26        printf("\n");
27        waitpid(2);
28        fork();
29        fork();
30        execve(long_running_program);
31    }
32    while(true);
33 }

```



### 3.7 Round Robin Scheduling

- In Part A, the initial version of the required Round Robin scheduling algorithm is present at the beginning of the code. The desired algorithm has been enhanced by adding process states and quantum time. Within the specified quantum time, we switch to the next task. Below are the relevant code and sample images.
- The information that each process needs to uniquely possess is contained within the Task class. Variables such as pid, pPid, processState, processPriority, and waitProcessPid are used to store this information.
- The process states are also included here.

```
class Task
{
    friend class TaskManager;
private:
    common::uint8_t stack[4096]; // 4 KiB
    CPUState* cpustate;
    common::uint32_t pid = -1;
    common::uint32_t Ppid = -1;
    common::int32_t waitpPidProcess = -99;
    ProcessState processState = Ready;
    int processPriority = 1;
    GlobalDescriptorTable *gdt;
};

enum ProcessState {Ready, Wait, Finish, Runing};
```

```

class TaskManager
{
private:
    Task *tasks[256];
    int numTasks;
    int currentTask;
    TaskManager *taskManager;
    GlobalDescriptorTable *gdt;
    //Task * readyQueue[256];
public:
    TaskManager();
    ~TaskManager();
    bool AddTask(Task* task);
    CPUState* Schedule(CPUState* cpustate);
    common::uint32_t getCurrentTaskPid();
    bool setCurrentTaskFinished(CPUState* cpustate);
    bool parentProcessWaitAllChild(CPUState* cpustate);
    CPUState* waitOneProcess(CPUState* cpustate);
    int getProcessIndex(int pid);
    int makeFork(CPUState* cpustate);
    CPUState* makeExec(CPUState* cpustate);
    void printProcessTable();
    void makeReadyQueue();
    CPUState* prioritySchedule(CPUState* cpustate);
    void prioritySort();
    int setPriorityTaskManager(CPUState* cpustate);
};

```

- Scheduling is done with a 60-quanta time.

- 

```

if (interrupt == hardwareInterruptorSet)
{
    quantaTime++;
    if (quantaTime == 60) {
        esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
        taskManager->printProcessTable();
        quantaTime = 0;
    }
}

```

me/src/drivers/vga.cpp

- The scheduling function has been enhanced from its initial state to accommodate process states and is supported by waitpid and exit system calls.
- The step-by-step explanation of the scheduling function is provided below
- First, the number of tasks is checked. If there are no tasks, the same cpustate is returned. In the second check, if the current task is greater than 0, the cpustate is saved to the current task's cpustate. Then, the current task is incremented and checked to move to the next task. If the current task is the last task, it loops back to the first task.

```

CPUState* TaskManager::Schedule(CPUState* cpustate)
{
    if (numTasks <= 0)
        return cpustate;
    if (currentTask >= 0)
        tasks[currentTask]->cpustate = cpustate;

    currentTask++;
    if (currentTask >= numTasks)
        currentTask %= numTasks;
    int i=0;
}

```

- In the outer if statement, we directly execute the task if it is in the ready state. If it is not ready, we check if it is in the wait state. If it is waiting and the task it is waiting for has finished, we change its state to ready. The initial if statement with a while loop was designed for the parent process to wait for child processes when the value is -1, but it is not used. The else statement checks the other processes. If the process it is waiting for has finished, its state is set to ready.

```

if(tasks[currentTask]->processState != Ready){
    //eğer parent process wait ise
    if(tasks[currentTask]->processState == Wait && tasks[currentTask]->waitPidProcess == -1){
        int i=0;
        int readyControllFlag =0;
        while(i<numTasks){
            if(tasks[i]->Ppid == tasks[currentTask]-> pid){
                if(tasks[i]->processState == Ready){
                    readyControllFlag = 1;
                    break;
                }
            }
            i++;
        }
        if(readyControllFlag == 0){
            tasks[currentTask]->processState = Ready;
        }
    }
    else if(tasks[currentTask]->processState == Wait && tasks[currentTask]->waitPidProcess != -99){
        int processIndex = getProcessIndex(tasks[currentTask]->waitPidProcess);
        if(tasks[processIndex]->processState == Finish){
            tasks[currentTask]->processState = Ready;
        }
    }
}
//if there arent ready process

```

- If there are no processes left to run, the operating system enters busy waiting.

```

//if there arent ready process
int flag= 0;
while(tasks[currentTask]->processState != Ready ){
    currentTask++;
    if(currentTask >= numTasks){
        currentTask %= numTasks;
        flag++;
    }

    if(flag == 2){
        printProcessTable();
        printf("All Process Finished\n");
        while(true);
        break;
    }
}

//printProcessTable();
return tasks[currentTask]->cpustate;

```

### 3.7.1 Test and Result

- We have created 8 processes as required. These processes will be used for 4 Collatz programs and 4 long-running programs. The long-running program will run after the Collatz program because the parent process will wait for the child process it created with waitpid. All process steps will be explained

```

void initProcess(){
    uint32_t num = getpid();
    printf("Init Process PID:");
    printfInt(num);

    int pid = fork();
    printf("    Fork PID:");
    printfInt(pid);
    printf("\n");
    if(pid ==0){
        int getPidChild =getPid();
        printf("Child pid:");
        printfInt(getPidChild);
        printf("\n");
        fork();
        setPriority(getPid(),5);
        fork();
        execve(CollatzSequence);
    }
    else{
        int getPidParent =getPid();
        printf("Parent PID:");
        printfInt(getPidParent);
        printf("\n");
        waitpid(2);
        fork();
        fork();
        execve(long_running_program);
    }
    while(true);
}

```

- As seen, the long-running program did not execute and put itself in a wait state. The child process will continue by duplicating itself.

The screenshot shows a terminal window with the following content:

```

number of tasks: 1    Current tasks: 1
The Virtual Machine reports that the guest OS does not support mouse pointer integration in the current
| PID | Parent PID | Priority | EIP | State |
|-----|-----|-----|-----|-----|
| 1 | 0 | 1 | 1088083 | Running |

Init Process PID:1    Fork PID:1
Parent PID:1
number of tasks: 2    Current tasks: 2
| PID | Parent PID | Priority | EIP | State |
|-----|-----|-----|-----|-----|
| 1 | 0 | 1 | 1055349 | Wait |
| 2 | 1 | 1 | 1055432 | Running |

Fork PID:0
Child pid:2
Collatz:2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 49,
KEYBOARD 0x1DKEYBOARD 0x38

```

- The child process duplicated itself and executed the Collatz operation. Here, it can be seen that the program is running step by step.

```

4. deneme [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
The Virtual Machine reports that the guest OS does not support mouse pointer integration in the current
Collatz:2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 49,
number of tasks: 5    Current tasks: 3
-----
| PID | Parent PID | Priority | EIP | State |
-----
| 1 | 0 | 1 | 1055349 | Wait |
| 2 | 1 | 5 | 1087339 | Ready |
| 3 | 2 | 5 | 50 | Running |
| 4 | 2 | 5 | 1055432 | Ready |
| 5 | 3 | 5 | 1055432 | Ready |
-----
KEYBOARD 0x1DKEYBOARD 0x38

```

- Up to this point, exit was not included at the end of the programs to show the steps step-by-step. Because the programs did not finish, the parent process could not continue its program flow. After adding exit at the end of the programs and running them again, the parent was able to continue its flow. The final screenshot is provided below.

```

4. deneme [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
The Virtual Machine reports that the guest OS does not support mouse pointer integration in the current
-----
| PID | Parent PID | Priority | EIP | State |
-----
| 1 | 0 | 1 | 1055392 | Finish |
| 2 | 1 | 5 | 1055392 | Finish |
| 3 | 2 | 5 | 1055392 | Finish |
| 4 | 2 | 5 | 1055392 | Finish |
| 5 | 3 | 5 | 1055392 | Finish |
| 6 | 1 | 1 | 1055392 | Finish |
| 7 | 1 | 1 | 1055392 | Finish |
| 8 | 6 | 1 | 1055432 | Running |
-----
Long running program Result:36100
number of tasks: 8    Current tasks: 1
-----
| PID | Parent PID | Priority | EIP | State |
-----
| 1 | 0 | 1 | 1055392 | Finish |
| 2 | 1 | 5 | 1055392 | Finish |
| 3 | 2 | 5 | 1055392 | Finish |
| 4 | 2 | 5 | 1055392 | Finish |
| 5 | 3 | 5 | 1055392 | Finish |
| 6 | 1 | 1 | 1055392 | Finish |
| 7 | 1 | 1 | 1055392 | Finish |
| 8 | 6 | 1 | 1055392 | Finish |
-----
All Process Finished

```

- The program was run and completed according to round-robin scheduling. In the screenshot, the final state of the processes is shown in the process table.

### **3.8 Conclusion**

- As a result, the required syscalls have been implemented, and multiple processes have been created from the init process using fork. Different tasks have been assigned to these processes using execve, and they have been executed using round-robin scheduling. There are no missing parts for the Part A section of the assignment. Due to limited resources, it has been difficult to support the report with screenshots. If there are any missing parts in the report, they can be explained during the demo.



## 4.0 Completion of Requirements for Part B

### 4.1 First Strategy

- The task of loading and running a specified program 10 times for the first strategy has been successfully completed. The scheduling algorithm written in the previous part was sufficient to accomplish this task. Below are the implementation, testing, and result steps.

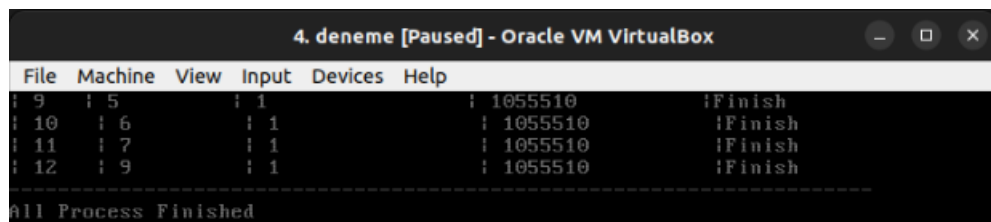
#### 4.1.1 Implementation

```
//Task task1(&gdt, initProcess);
Task task2(&gdt, firstStrategy);
//Task task3(&gdt, secondStrategy);
//Task task4(&gdt, initProcessPriorityStrategy);
//Task task5(&gdt, initProcessPartC);

//taskManager.AddTask(&task1);
taskManager.AddTask(&task2);
//taskManager.AddTask(&task3);
//taskManager.AddTask(&task4);
//taskManager.AddTask(&task5);
```

```
void firstStrategy(){
    fork();
    fork();
    int pid =fork();
    if(pid ==0){
        fork();
    }
    execve(linearSearch);
    while(true);
}
```

- After every 60 interrupts, the next process will be taken, or if a process exits and finishes, the next one will be taken.



PID	PPID	PNAME	State
9	5	1055510	Finish
10	6	1055510	Finish
11	7	1055510	Finish
12	9	1055510	Finish

All Process Finished

**Note:** All processes run quickly, but screenshots cannot be captured.

## 4.2 Second Strategy

- For this strategy, the scheduling implemented in Part A was used, and nothing extra was added.
- Two out of the specified four processes were loaded into memory and executed. Below are the screenshots and implementation details

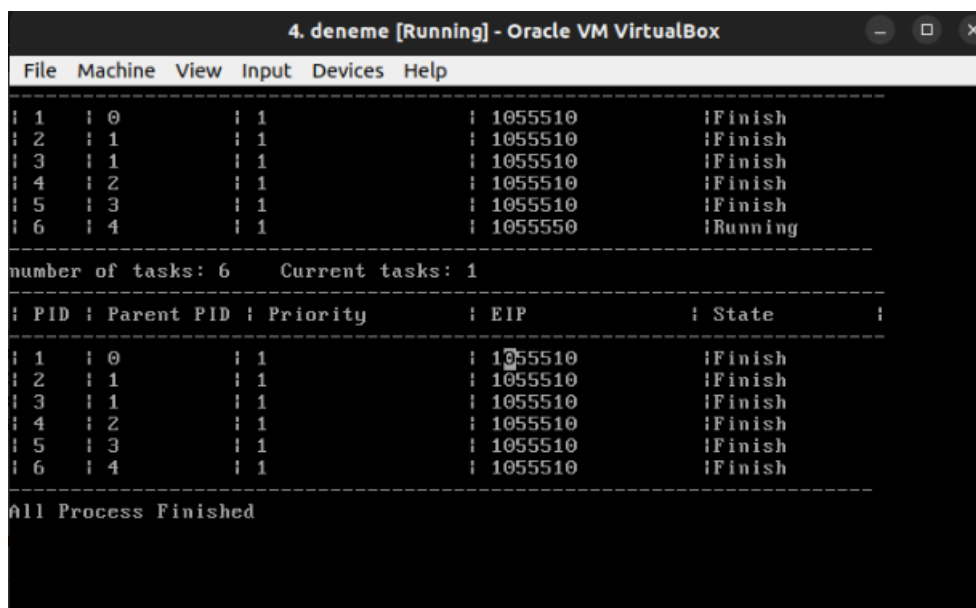
### 4.2.1 Implementation

```
//Task task1(&gdt, initProcess);
//Task task2(&gdt, firstStrategy);
Task task3(&gdt, secondStrategy);
//Task task4(&gdt, initProcessPriorityStrategy);
//Task task5(&gdt, initProcessPartC);

//taskManager.AddTask(&task1);
//taskManager.AddTask(&task2);
taskManager.AddTask(&task3);
//taskManager.AddTask(&task4);
//taskManager.AddTask(&task5);
```

```
403 void secondStrategy(){
404     int pid =fork();
405     if(pid ==0){
406         int cpid= fork();
407         if(cpid ==0){
408             fork();
409         }
410         exece(linearSearch);
411     }
412     else{
413         int cpid =fork();
414         if(cpid ==0){
415             fork();
416         }
417         exece(long_running_program);
418     }
419     while(true);
420 }
421
```

- The program was executed as required, and results were produced.



```
4. deneme [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
-----
| 1 | 0 | 1 | 1055510 | iFinish
| 2 | 1 | 1 | 1055510 | iFinish
| 3 | 1 | 1 | 1055510 | iFinish
| 4 | 2 | 1 | 1055510 | iFinish
| 5 | 3 | 1 | 1055510 | iFinish
| 6 | 4 | 1 | 1055550 | iRunning
-----
number of tasks: 6    Current tasks: 1
-----
| PID | Parent PID | Priority | EIP | State |
-----
| 1 | 0 | 1 | 1055510 | iFinish
| 2 | 1 | 1 | 1055510 | iFinish
| 3 | 1 | 1 | 1055510 | iFinish
| 4 | 2 | 1 | 1055510 | iFinish
| 5 | 3 | 1 | 1055510 | iFinish
| 6 | 4 | 1 | 1055510 | iFinish
-----
All Process Finished
```

### 4.3 Third Strategy

- For this strategy, the processes stored in the tasks[256] array were sorted by priority. Processes with the highest priority were placed at the beginning, while those with lower priority were placed at the end. The process was switched every 5 interrupts.

```
//Task task1(&gdt, initProcess);
//Task task2(&gdt, firstStrategy);
//Task task3(&gdt, secondStrategy);
Task task4(&gdt, initProcessPriorityStrategy);
//Task task5(&gdt, initProcessPartC);

//taskManager.AddTask(&task1);
//taskManager.AddTask(&task2);
//taskManager.AddTask(&task3);
taskManager.AddTask(&task4);
//taskManager.AddTask(&task5);

void initProcessPriorityStrategy(){
    int pid = fork();

    if(pid == 0){
        fork();
        fork();
        execve(CollatzSequence);
    }
    else{
        int getpidParent =getPid();

        fork();
        fork();
        setPriority(getPid(),5);
        execve(long_running_program);
    }
    while(true);
}
```

- The scheduling algorithm is provided below. According to the scheduling algorithm, processes with lower priority are placed at the end, while those with higher priority are placed at the beginning. Each time scheduling occurs, the processes are sorted again.

```
if(quantumTime==5){
    esp = (uint32_t)taskManager->prioritySchedule((CPUState*)esp);
    taskManager->printProcessTable();
    quantumTime=0;
}
```

- The function that sorts tasks by priority is provided below.

```
224 void TaskManager::prioritySort()
225 {
226     for (int i = 0; i < numTasks - 1; i++)
227     {
228         for (int j = 0; j < numTasks - i - 1; j++)
229         {
230             if (tasks[j]->getPriority() < tasks[j + 1]->getPriority())
231             {
232                 Task* temp = tasks[j];
233
234                 tasks[j] = tasks[j + 1];
235                 tasks[j + 1] = temp;
236             }
237         }
238     }
239 }
240
```

- The algorithm selects and runs the highest priority process at the smallest index.

```

CPUState* TaskManager::prioritySchedule(CPUState* cpustate)
{
    if (numTasks <= 0)
        return cpustate;
    if (currentTask >= 0)
        tasks[currentTask]->cpustate = cpustate;

    for(int i=numTasks/2; i< numTasks; i++){
        tasks[i]->setPriority(tasks[i]->getPriority()+1);
    }
    prioritySort();
    int highestPriorityIndex = -1;
    int highestPriority = -1;

    for (int i = 0; i < numTasks; i++)
    {
        if (tasks[i]->processState == Ready && tasks[i]->getPriority() > highestPriority)
        {
            highestPriority = tasks[i]->getPriority();
            highestPriorityIndex = i;
        }
    }
    if (highestPriorityIndex != -1)
    {
        currentTask = highestPriorityIndex;
    }

    if (currentTask >= 0)
    {
        tasks[currentTask]->processState = Ready;
        return tasks[currentTask]->cpustate;
    }

    return tasks[currentTask]->cpustate;
}

```

- As seen, the high-priority processes are moved to the front and executed first.

4. deneme [Paused] - Oracle VM VirtualBox

File Machine View Input Devices Help

Collatz: 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,

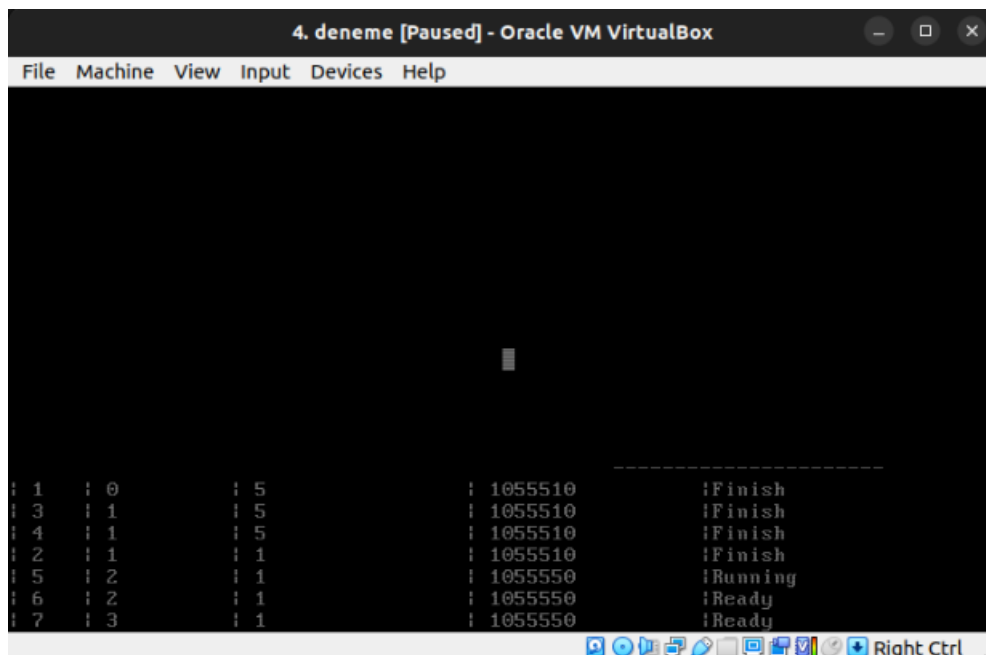
number of tasks: 6      Current tasks: 3

PID	Parent PID	Priority	EIP	State
1	0	5	1055510	Finish
2	1	1	1055510	Finish
3	1	1	1055550	Running
4	1	1	1055550	Ready
5	2	1	1055550	Ready
6	2	1	1055550	Ready

number of tasks: 6      Current tasks: 3

PID	Parent PID	Priority	EIP	State
1	0	5	1055510	Finish
2	1	1	1055510	Finish
3	1	1	1055550	Running
4	1	1	1055550	Ready
5	2	1		

Right Ctrl



#### 4.4 Dynamic Priority Strategy

- For the Dynamic Priority Strategy step, the priority of low-priority processes was increased by one at each scheduling, and then the processes were sorted again. This step was done by increasing the priority of the tasks in the lower half, i.e., those with low priority at the higher indices, by 1. The following code snippet was added to the priorityScheduling function in addition to the third strategy.
- A marked for loop was added to increase the priorities.

```

CPUState* TaskManager::prioritySchedule(CPUState* cpustate)
{
    if (numTasks <= 0)
        return cpustate;
    if (currentTask >= 0)
        tasks[currentTask]->cpustate = cpustate;

    for(int i=numTasks/2; i< numTasks; i++){
        if(tasks[i]->processState == Ready){
            tasks[i]->setPriority(tasks[i]->getPriority()+1);
        }
    }

    prioritySort();
    int highestPriorityIndex = -1;
    int highestPriority = -1;

    for (int i = 0; i < numTasks; i++)
    {
        if (tasks[i]->processState == Ready && tasks[i]->getPriority() > highestPriority){
            highestPriority = tasks[i]->getPriority();
            highestPriorityIndex = i;
        }
    }
    if (highestPriorityIndex != -1)
    {
        currentTask = highestPriorityIndex;
    }
}

```

- The same init process was used as in the third strategy.

```
//Task task1(&gdt, initProcess);
//Task task2(&gdt, firstStrategy);
//Task task3(&gdt, secondStrategy);
Task task4(&gdt, initProcessPriorityStrategy);
//Task task5(&gdt, initProcessPartC);

//taskManager.AddTask(&task1);
//taskManager.AddTask(&task2);
//taskManager.AddTask(&task3);
taskManager.AddTask(&task4);
//taskManager.AddTask(&task5);
```

```
void initProcessPriorityStrategy(){
    int pid = fork();

    if(pid == 0){
        fork();
        fork();
        execve(CollatzSequence);
    }
    else{
        int getPidParent =getPid();

        fork();
        fork();
        setPriority(getPid(),5);
        execve(long_running_program);
    }
    while(true);
}
```

- It can be seen that the priority of processes initially set to 1 has increased.

4. deneme [Paused] - Oracle VM VirtualBox

File Machine View Input Devices Help

You have the **Auto capture keyboard** option turned on. This will cause the Virtual Machine to automatically

The Virtual Machine reports that the guest OS does not support **mouse pointer integration** in the current

```
numbetasks: 8    Current tasks: 2
```

PID	Parent PID	Priority	EIP	State
8	6	7	1055510	Finish
6	2	6	1086543	Running
7	2	6	1055550	Ready
1	0	5	1055510	Finish
3	1	5	1055510	Finish
4	1	5	1055510	Finish
5	3	5	1055510	Finish
2	1	5	1055510	Finish

z:2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23  
 , 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43  
 , 44, 45, 46, 47, 48, 49,

Right Ctrl

## 5.0 Completion of Requirements for Part C

### 5.1 Random Process Spawning with Interactive Input Handling Strategy

In this part, the requested process switching using the keyboard or mouse has been implemented. When the key '1' is pressed, the processes start running, and when the key '2' is pressed, the process table is printed.

- Here, the necessary modifications were made in the keyboard.cpp file to implement the functionality.

```
uint32_t KeyboardDriver::HandleInterrupt(uint32_t esp)
{
    uint8_t key = dataport.Read();

    if(handler == 0)
        return esp;

    if(key < 0x80)
    {
        switch(key)
        {
            case 0x02: handler->OnKeyDown('1');
                       esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
                       taskManager->printProcessTable();
                       break;
            case 0x03: handler->OnKeyDown('2');
                       taskManager->printProcessTable();
                       break;
        }
    }
}
```

- Part c için bir init process ve bir çok çocuk process oluşturup çalıştırdık. process geçişleri 1 tuşuna basınca diğer processe başlıyor.

```
//Task task1(&gdt, initProcess);
//Task task2(&gdt, firstStrategy);
//Task task3(&gdt, secondStrategy);
//Task task4(&gdt, initProcessPriorityStrategy);
Task task5(&gdt, initProcessPartC);

//taskManager.AddTask(&task1);
//taskManager.AddTask(&task2);
//taskManager.AddTask(&task3);
//taskManager.AddTask(&task4);
taskManager.AddTask(&task5);
```



```

void initProcessPartC(){
    int initProcessPid = getPid();

    int pid = fork();

    if(pid ==0){
        int getPidChild =getPid();
        printf("Child pid:");
        printfInt(getPidChild);
        printf("\n");
        fork();
        fork();

        execve(CollatzSequence);
    }
    else{
        int getPidParent =getPid();
        //printf("Parent PID:");
        //printfInt(getPidParent);
        //printf("\n");
        //waitpid(2);
        if(fork()==0){
            waitpid(2);
            execve(infTaskA);
        }
        fork();
        fork();
        //setPriority(getPid(),5);
        execve(infTaskC);
    }
    while(true);
}

```

- We are commenting out the interrupt-based scheduling part. We will perform scheduling only with the key press.

```

if(interrupt == hardwareInterruptOffset)
/*
    if(quantumTime==60){
        esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
        taskManager->printProcessTable();
        quantumTime=0;
    }
    */
    /*
    if(quantumTime==5){
        esp = (uint32_t)taskManager->prioritySchedule((CPUState*)esp);

        taskManager->printProcessTable();
        quantumTime=0;
    }
    */

```

- The program waits for the '1' key to be pressed to switch to the next task. Below are the outputs of the programs after pressing '1'.





```

4. deneme [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
| 9 | 6 | 3 | 1055432 | Ready
| 4 | 1 | 2 | 1089022 | Ready
| 5 | 1 | 2 | 1089022 | Ready
| 6 | 2 | 2 | 1055392 | Finish
| 7 | 2 | 2 | 1055432 | Ready
| 1 | 0 | 2 | 1089022 | Ready
| 3 | 1 | 2 | 1055432 | Ready
| 2 | 1 | 1 | 1055392 | Finish
-----
Number of tasks: 9    Current tasks: 1
-----
| PID | Parent PID | Priority | EIP | State |
-----
| 8 | 4 | 3 | 100 | Running
| 9 | 6 | 3 | 1055432 | Ready
| 4 | 1 | 2 | 1089022 | Ready
| 5 | 1 | 2 | 1089022 | Ready
| 6 | 2 | 2 | 1055392 | Finish
| 7 | 2 | 2 | 1055432 | Ready
| 1 | 0 | 2 | 1089022 | Ready
| 3 | 1 | 2 | 1055432 | Ready
| 2 | 1 | 1 | 1055392 | Finish
-----
KEYBOARD 0x1DKEYBOARD 0x38

```

- The first part of Part C has been completed as described.

## 5.2 Interactive Input Priority Strategy

- For the Interactive Input Priority Strategy, we use the prioritySort function that we used in Part B. This allows us to sort the tasks in the tasks[256] array by priority and execute them accordingly. When the mouse is clicked, we select the tasks with higher priority at the lower indices. Conversely, when the mouse is clicked, we select tasks with lower priority at the higher indices. In this way, we accomplish what is required.
- We obtained the pointer to the TaskManager within the MouseHandler, making it usable. We developed an algorithm with our custom lowerLevelPriority function to execute the low-priority processes at the last indices. When the mouse is clicked, our special scheduling function runs.

```
for(uint8_t i = 0; i < 3; i++)
{
    if((buffer[0] & (0x1<<i)) != (buttons & (0x1<<i)))
    {
        if(buttons & (0x1<<i))
            handler->OnMouseUp(i+1);
        else{
            handler->OnMouseDown(i+1);
            esp = taskManager->lowerLevelSchedule(esp);
            taskManager->printProcessTable();
        }
    }
    buttons = buffer[0];
}
```

```
365 common::uint32_t TaskManager::lowerLevelSchedule(common::uint32_t esp){
366     CPUState* cpustate = (CPUState*) esp;
367
368     if (numTasks <= 0)
369         return (common::uint32_t) cpustate;
370     if (currentTask >= 0)
371         tasks[currentTask]->cpustate = cpustate;
372
373     prioritySort();
374     for(int i = numTasks-1; i > -1 ; i--){
375         if(tasks[i]->processState == Ready){
376             currentTask = i;
377             break;
378         }
379     }
380
381     return (common::uint32_t) tasks[currentTask]->cpustate ;
382 }
```

- When the '2' key is pressed on the keyboard, we run the high-priority task.

```
uint32_t KeyboardDriver::HandleInterrupt(uint32_t esp)
{
    uint8_t key = dataport.Read();

    if(handler == 0)
        return esp;

    if(key < 0x80)
    {
        switch(key)
        {
            case 0x02: handler->OnKeyDown('1');
                       esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
                       taskManager->printProcessTable();
                       break;
            case 0x03: handler->OnKeyDown('2');
                       esp = (uint32_t)taskManager->prioritySchedule((CPUState*) esp);
                       taskManager->printProcessTable();
                       break;
            case 0x04: handler->OnKeyDown('3');
                       taskManager->printProcessTable();
                       break;
            case 0x05: handler->OnKeyDown('4'); break;
        }
    }
}
```

- The '2' key was pressed, and the first process ran

```
File Machine View Input Devices Help
Number of tasks: 1    Current tasks: 1
-----
| PID | Parent PID | Priority      | EIP                | State          |
-----+-----+-----+-----+-----+
| 1   | 0         | 2           | 1089372            | Running        |
-----
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCnumber of tasks: 5    Current tasks: 1
-----
| PID | Parent PID | Priority      | EIP                | State          |
-----+-----+-----+-----+-----+
| 1   | 0         | 5           | 100                 | Running        |
| 2   | 1         | 2           | 1055506             | Ready          |
| 3   | 1         | 2           | 1055506             | Ready          |
| 4   | 1         | 2           | 1055506             | Ready          |
| 5   | 1         | 2           | 1055506             | Ready          |
-----
KEYBOARD 0x1DKEYBOARD 0x3B
```

- Now I clicked the mouse, and it started running the last process.

```
File Machine View Input Devices Help
| 5 | 1 | 2 | 1055506 | Running
-----
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCC3number of tasks: 5      Current tasks: 5
-----
| PID | Parent PID | Priority | EIP | State |
-----
| 1 | 0 | 5 | 1089370 | Ready |
| 2 | 1 | 2 | 1055506 | Ready |
| 3 | 1 | 2 | 1055506 | Ready |
| 4 | 1 | 2 | 1055506 | Ready |
| 5 | 1 | 5 | 100 | Running |
-----
KEYBOARD 0x1DKEYBOARD 0x38
```

- I pressed the '2' key on the keyboard and ran the first process again.

```
File Machine View Input Devices Help
| 5 | 1 | 2 | 1055506 | Running
-----
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCC3number of tasks: 5    Current tasks: 5
-----
| PID | Parent PID | Priority | EIP | State |
-----
| 1 | 0 | 5 | 1089370 | Ready |
| 2 | 1 | 2 | 1055506 | Ready |
| 3 | 1 | 2 | 1055506 | Ready |
| 4 | 1 | 2 | 1055506 | Ready |
| 5 | 1 | 5 | 100 | Running |
-----
KEYBOARD 0x1DKEYBOARD 0x38number of tasks: 5    Current tasks: 1
-----
| PID | Parent PID | Priority | EIP | State |
-----
| 5 | 1 | 6 | 1089370 | Running |
| 1 | 0 | 5 | 1089370 | Ready |
| 3 | 1 | 3 | 1055506 | Ready |
| 4 | 1 | 3 | 1055506 | Ready |
| 2 | 1 | 2 | 1055506 | Ready |
-----
KEYBOARD 0x1DKEYBOARD 0x38
```

The process with the lowest priority at the last index started running again.

```
| File Machine View Input Devices Help
```

PID   Parent PID   Priority   EIP   State
5   1   6   1089370   Ready
1   0   5   1089370   Ready
3   1   3   1055506   Ready
4   1   3   1055506   Ready
2   1   2   1055506   Running

```
-----  
Child pid:2  
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  
BBBBBBBBBBBBBBBBBBBBKEYBOARD 0x1DKEYBOARD 0x3E
```

Right Ctrl

## **6 Conclusions**

As a result, the parts of the assignment have been completed. Some sections were interpreted and implemented according to our understanding. For testing, some parts need to be commented out, and some parts need to be uncommented.