Gebze Technical University

CSE 312 Spring 2024 Operating Systems

Assignment #01 Report

Abdulsamet ASLAN

200104004098

Contents

1. Project Overview
2. Process Table and Processes
3. Interrupt Handling
4. Process Scheduling
5. POSIX System Calls
6. Programs
7. Keyboard and Mouse Interaction
8. Part A
9. Part B
    a. First Strategy
    b. Second Strategy
    c. Third Strategy
    d. Fourth Strategy
10. Part C
    a. First Strategy
    b. Second Strategy
11. Compiling and Running

1) Project Overview

In this assignment, an operating system is redesigned and developed using Viktor Engelmann's YouTube video series. This OS includes several POSIX system calls such as fork, exit, execv, waitpid, and nice. Additionally, round-robin scheduling and priority-based scheduling are applied to the existing code. Eight different kernels are implemented as required by the assignment.

2) Process Table and Processes

To create a new **process**, a code segment selector and an entry point for the CPU to execute are required. Each process has its own stack space of 4096 kilobytes, a process ID (PID), a parent process ID, a waiting PID, status, priority, and execution time. The PID is used to identify the process. The stack stores local variables and CPU state registers, which are saved by the scheduler during a context switch. The priority represents the importance of the process and is used for priority-based scheduling. In this assignment, all the processes are created using the fork system call, except for the initialization functions.

The **process table** is an array of pointers to tasks, with a size of 256, stored in the Task Manager class, which is responsible for scheduling.

```
class Task
{
    friend class TaskManager;
    private:
        common::uint8_t stack[4096]; // 4 KiB
        CPUState* cpustate;

        common::uint32_t pid;
        common::uint32_t ppid;
        common::uint32_t waitingPid;

        Status status;
        Priority priority;
        int executionTime = 0;

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

3) Interrupt Handling

I have implemented a slow version of interrupts, which is enabled when the SLOW option is enabled. In this version, the scheduler is called for every 50th timer interrupt. Additionally, a static variable is used to disable the timer interrupt.

```
if(interrupt == hardwareInterruptOffset)
{
    #ifdef SLOW
    if (counter % 50 == 0 && !InterruptManager::isWaitingForInput)
    {
        esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
        counter = 0;
    }
    counter++;
    #else
     if (!InterruptManager::isWaitingForInput)
        esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
        counter = 0;
    #endif

}
```

4) Process Scheduling

Two different scheduling algorithms are implemented for this assignment: round-robin scheduling and priority-based scheduling. Both of these algorithms also check the blocked processes to determine if they need to be unblocked.

Round-Robin Scheduling: This algorithm gives each process an equal chance to run. When an interrupt occurs, it traverses the process table to find the next process to execute.

```
while (tasks[next_task]->status != Status::READY && next_task != currentTask)
{
    if(tasks[next_task]->status == Status::WAITING) //Check if the task is wai
    {
        if(!isWaitingForChild(tasks[next_task]->pid))
        {
            tasks[next_task]->waitingPid = -1;
            tasks[next_task]->status = Status::READY;
            continue;
        }
    }
    next_task = (next_task + 1) % numTasks;
}
```

Priority-Based Scheduling: In this algorithm, every process starts with the same default priority. However, the priority can be changed to a higher or lower level by the process itself using the nice system call. This algorithm examines the ready processes and selects the one with the highest priority to run. Lowest number is considered as highest priority.

```
enum Priority{VERY_HIGH, HIGH, MEDIUM, LOW, VERY_LOW};
```

```
while(next_task != currentTask)
{
    if (tasks[next_task]->status == Status::TERMINATED)
    {
        next_task = (next_task + 1) % numTasks;
        continue;
    }

    if(tasks[next_task]->status == Status::WAITING)
    {
        if(!isWaitingForChild(tasks[next_task]->pid))
        {
            tasks[next_task]->waitingPid = -1;
            tasks[next_task]->status = Status::READY;
        }
    }

    if( tasks[next_task]->status == Status::READY && ( (int) tasks[next_task]->priority < highest_priority))
    {
        highest_priority = tasks[next_task]->priority;
        highest_priority_task = next_task;
    }

    next_task = (next_task + 1) % numTasks;

}
```

5) Posix System Calls

When a system call is invoked, the required system call number is written to the eax register, and a 0x80 interrupt is generated for system call interrupts. If the system call has parameters, they are written to the ecx register. If the system call has a return value, it is read from the ebx register. The utilization of the registers is outlined on following webpage: https://www.eecg.utoronto.ca/~amza/www.mindsec.com/files/x86regs.html

When a 0x80 or 0x20 (Timer) interrupt occurs, the interrupt handler invokes the handle interrupt function. This function then calls the necessary system call function and, if needed, triggers the scheduler.

```
void syscalls::exit() {
    asm("int $0x80" : : "a" (SYSCALLS::EXIT));
}

uint32_t syscalls::execve(void (*entrypoint)()) {
    uint32_t ret = -1;

    asm("int $0x80" :: "a" (SYSCALLS::EXECV), "c" ((uint32_t)entrypoint));
    __asm__ volatile("" : "=b"(ret));
    return ret;

}
void syscalls::nice(Priority newPriority)
{
    asm("int $0x80" :: "a" (SYSCALLS::NICE), "c" ((int)newPriority));
}

uint32_t syscalls::fork()
{
    uint32_t ret = -1;

    asm("int $0x80" :: "a" (SYSCALLS::FORK));
    __asm__ volatile("" : "=b"(ret));

    return ret;
}

void syscalls::waitpid(uint16_t pid) {
    asm("int $0x80" : : "a" (SYSCALLS::WAITPID), "c" (pid));
}
```

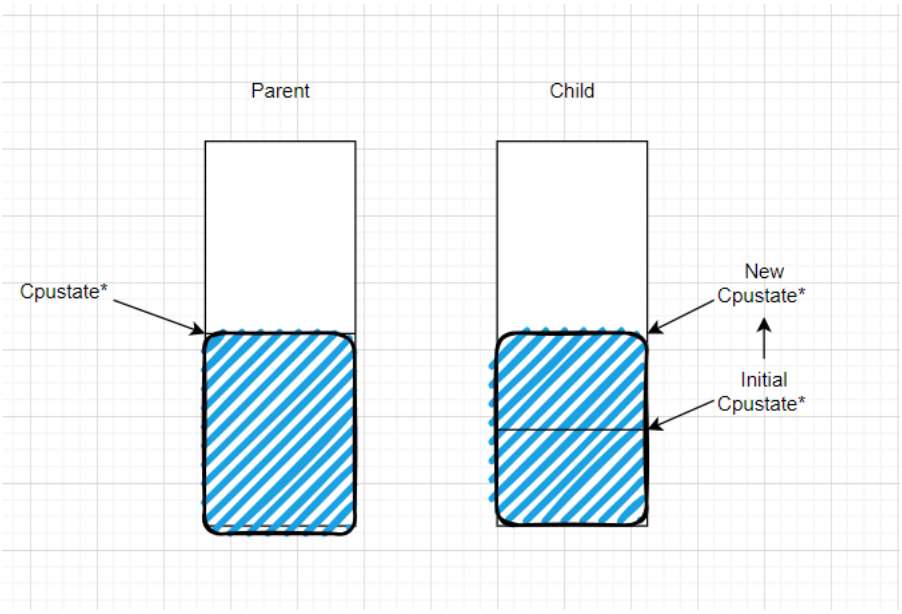
```
uint32_t SyscallHandler::HandleInterrupt(uint32_t esp)
{
    CPUState* cpu = (CPUState*)esp;


    switch(cpu->eax)
    {
        case SYSCALLS::EXIT:
            interruptManager->taskManager->sys_exit();
            return SyscallHandler::Schedule(esp);
            break;

        case SYSCALLS::FORK:
            cpu->ebx = interruptManager->taskManager->sys_fork(cpu);
            return SyscallHandler::Schedule(esp);
        break;

        case SYSCALLS::WAITPID:
            if(interruptManager->taskManager->sys_waitpid(cpu->ecx)){
                return SyscallHandler::Schedule(esp);
            }
            break;

        case SYSCALLS::EXECV:
            cpu->ebx = interruptManager->taskManager->sys_execve((void (*)())(cpu->ecx));
            return SyscallHandler::Schedule(esp);
            break;

        case SYSCALLS::NICE:

            interruptManager->taskManager->sys_nice((Priority)cpu->ecx);
            return SyscallHandler::Schedule(esp);
            break;

        default:
            break;
    }
```

Fork:

Forking essentially adds a new process to the process table. However, this new process shares the same stack and CPU state registers as the parent process. The ebx register is set to 0 to return 0 for the child process.

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

    Task* parent = tasks[currentTask];
    Task* childTask = new Task(gdt, (void(*)()) cpustate->eip);
    childTask->ppid = parent->pid;
    AddTask(childTask);


    for (int i = 0; i < 4096 ; i++)
    {
        childTask->stack[i] = parent->stack[i];
    }


    common::uint32_t currentTaskOffset = ((common::uint32_t)cpustate) - (common::uint32_t)(parent->stack) ;
    childTask->cpustate = (CPUState*)(childTask->stack +  currentTaskOffset);


    childTask->cpustate->ebx = 0;
    return childTask->pid;
}
```

The following section explains the process of fork. When copying the stack content to another process, the CPU state register must be updated to reflect the new location.



Execve:

It works similarly to fork, but in this system call, the new process has its own stack and entry point.

```
uint32_t TaskManager::sys_execve(void entry_point())
{
    if(numTasks >= 256)
        return 0;

    Task* task = new Task(gdt, entry_point);

    task->cpustate->ebx = 0;
    task->status = Status::READY;
    task->ppid = tasks[currentTask]->pid;
    AddTask(task);
    return task->pid;
}
```

Waitpid:

This system call sets the current process status to “waiting.” Consequently, in the scheduler, this process will be evaluated as a blocked process.

```
bool TaskManager::sys_waitpid(common::uint32_t childPid)
{
    if (getTaskByPid(childPid)->status == Status::TERMINATED)
        return false;

    if (childPid == tasks[currentTask]->pid)
        return false;

    tasks[currentTask]->status = Status::WAITING;
    tasks[currentTask]->waitingPid = childPid;
    return true;
}
```

Exit:

The current process status is changed to “Terminated,” and therefore, the process is evaluated as a finished process in the scheduler.

```
bool TaskManager::sys_exit()
{
    if(numTasks <= 0)
        return false;

    tasks[currentTask]->status = Status::TERMINATED;
}
```

Nice:

The Nice system call is created to adjust the priority of the current task. The process priority is used for priority-based scheduling.

```
void TaskManager::sys_nice(Priority priority)
{
    tasks[currentTask]->priority = priority;
    printfHex((int)priority);
}
```

Example using of all system calls:

```
void syscalls_test()
{
    int pid = fork();
    if(pid == 0)
    {
        int pid2 = fork();
        if (pid2 == 0)
        {
            printf("2. Child is running\n");
            for (int i = 0; i < 100000000; i++)
                printf("");
            exit();

        }
        else
        {
            printf("1. Child is running\n");
            waitpid(pid2);
            int execpid = execve(taskC);
            while(true){
                printf("");
            }
        }
    }
    else
    {
        printf("Parent is running\n");
        while(true){
            printf("");
        }
    }
}
```

System Call Results:

```
Welcome to My Operating System LOOS
2. Child is running
Parent is running
1. Child is running
2. Child is exited
1. child called execv is running
1. child called execv is exited
1. Child is exited
Parent is exited

¦ PID: 00¦ PPID: FF¦ PRIORITY: 02 TERMINATED
¦ PID: 01¦ PPID: 00¦ PRIORITY: 02 TERMINATED
¦ PID: 02¦ PPID: 01¦ PRIORITY: 02 TERMINATED
¦ PID: 03¦ PPID: 01¦ PRIORITY: 02 TERMINATED

¦ PID: 00¦ PPID: FF¦ PRIORITY: 02 TERMINATED
¦ PID: 01¦ PPID: 00¦ PRIORITY: 02 TERMINATED
¦ PID: 02¦ PPID: 01¦ PRIORITY: 02 TERMINATED
¦ PID: 03¦ PPID: 01¦ PRIORITY: 02 TERMINATED
```

6) Programs

Collatz, Binary Search, Linear Search, Long Running Program

```c
void collatz(int n){
    printf("\n------------(collatz)------------ \n");
    printfDigit(n);
    printf(": ");
    while (n != 1)
    {
        if (n % 2 == 0)
            n /= 2;
        else
            n = 3 * n + 1;

        printfDigit(n);
        printf(" ");
        // sleep(1000000);
    }
    printf("\n");
    exit();
}
```

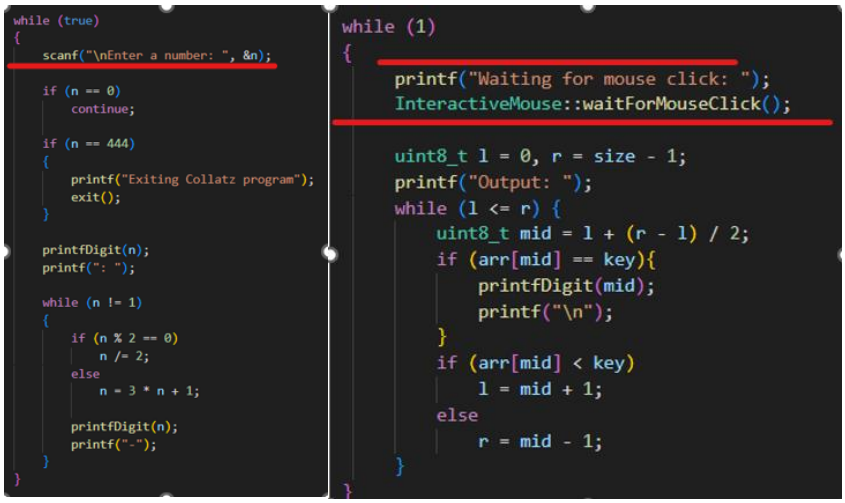
```c
void binarySearch(int arr[], int size, int key) {
   printf("\n------------(Binary Search)------------ \n");
   uint8_t l = 0, r = size - 1;
   printf("Output: ");
   while (l <= r) {
      uint8_t mid = l + (r - l) / 2;
      if (arr[mid] == key){
         printfDigit(mid);
         printf("\n");
         exit();
      }
      if (arr[mid] < key)
         l = mid + 1;
      else
         r = mid - 1;
   }
    printf("Not found");
    exit();
}
```

```c
void linearSearch(int arr[], int size, int key) {

    printf("\n------------(Linear Search)------------ \n");
    printf("Output: ");

    for(int i=0; i< size; i++) {
        if(arr[i] == key){
            printfDigit(i);
            printf("\n");
            exit();
        }
    }
    printf("Not Found");
    exit();
}
```

```c
void long_running_program(int n)
{
    printf("\n------------(Long Running Program)------------\n");

    int result = 0;
    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j < n; j++)
        {
            result += i * j;
        }
    }
    printf("Result: ");
    printfDigit(result);
    printf("\n");
    exit();
}
```

Implementation of I/O Bounds Programs

I utilized the "scanf" function in the collatz function to enable user input. The binary search and linear search functions are programmed to await a mouse click. No context switch is allowed while these functions are waiting for user input. However, if a timer interrupt occurs while the process is not waiting, it triggers context switching.



```
while (true)
{
    scanf("\nEnter a number: ", &n);

    if (n == 0)
        continue;

    if (n == 444)
    {
        printf("Exiting Collatz program");
        exit();
    }

    printfDigit(n);
    printf(": ");

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

        printfDigit(n);
        printf("-");
    }
}
```

```
while (1)
{
    printf("Waiting for mouse click: ");
    InteractiveMouse::waitForMouseClick();

    uint8_t l = 0, r = size - 1;
    printf("Output: ");
    while (l <= r) {
        uint8_t mid = l + (r - l) / 2;
        if (arr[mid] == key){
            printfDigit(mid);
            printf("\n");
        }
        if (arr[mid] < key)
            l = mid + 1;
        else
            r = mid - 1;
    }
}
```

7)Keyboard and Mouse Interaction:

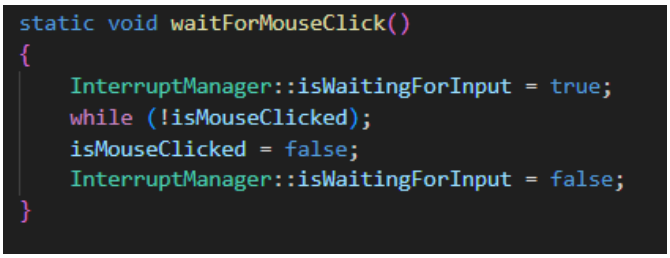
When a process calls scanf, it disables the timer interrupts and waits for the "\n" character from the user by continuously reading from a static buffer.

```
void scanf(char* label, int *res)
{
    char str[256];
    printf(label);
    readline(str);
    if(atoi(str, res) == -1)
    {
        printf("Invalid input !");
        res = 0;
    }
}
```

```
void readline(char * res)
{
    InterruptManager::isWaitingForInput = true;

    char *str;
    while ((str = KeyboardEventHandler::readFromBuffer())

    InterruptManager::isWaitingForInput = false;
    int i;
    for(i = 0; str[i] != '\0'; i++)
        res[i] = str[i];
    res[i] = '\0';
}
```

When a key is pressed, the keyboard handler writes it to the buffer.

The same approach is used for mouse interaction. It also disables interrupts and waits for a mouse press.



8) PART A

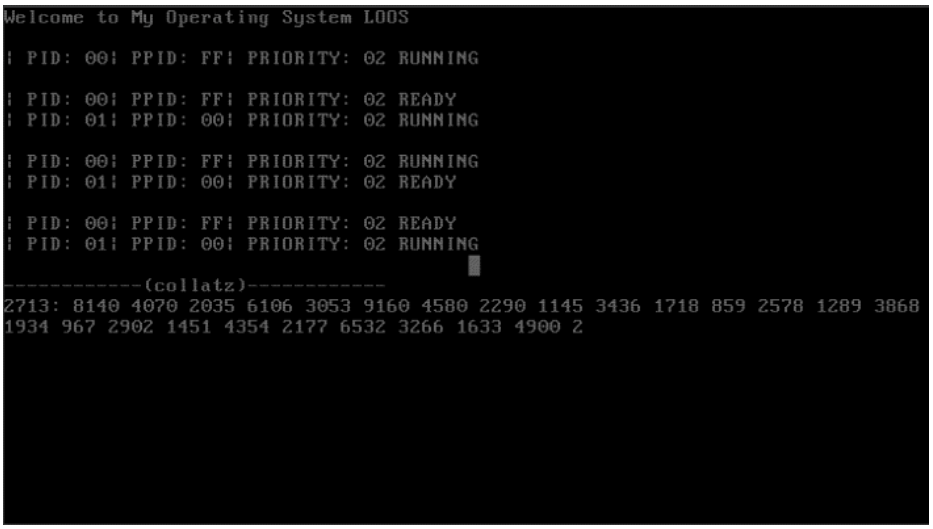
In this part, the kernel adds the first task named init. The init function then forks six times: three for the collatz program and three for the long_running_program. After the creation of the child processes, the parent process starts waiting for them to terminate. Once all the child processes have finished, the parent process is terminated. **Round robin** scheduling algorithm is used for this strategy.

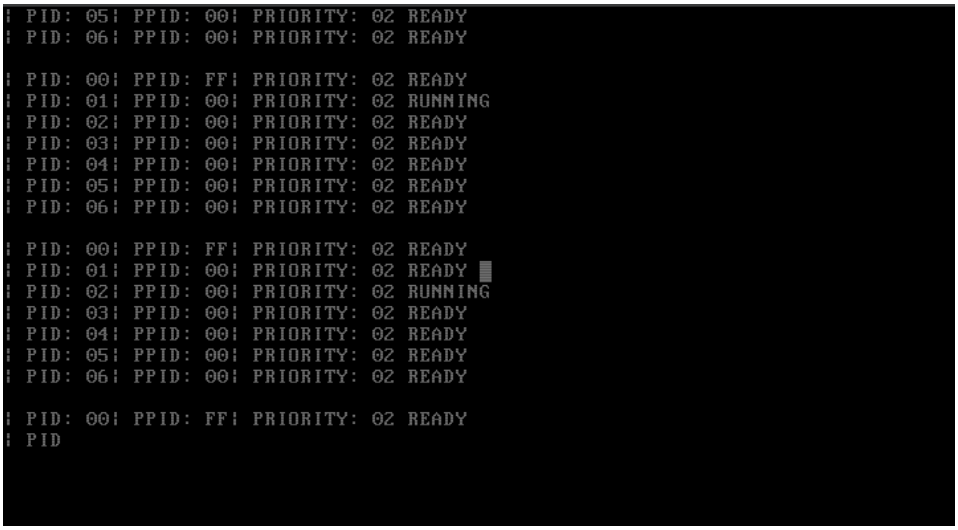
Strategy:

```c
void init_partA_1()
{
    /*This Strategy is loading each program 3 times, starting them and will enter an
    infinite loop until all the processes terminate. */
    const int num_process = 3*2;
    int pids[num_process];

    for(int i = 0; i <num_process; i++)
    {
        pids[i] = fork();
        if(pids[i] == 0)
        {
            switch(i%2)
            {
                case 0:
                    collatz(2713);
                    break;
                case 1:
                    long_running_program(1000);
                    break;
                default:
                    break;
            }
        }
    }
    for(int i = 0; i< num_process; i++)
        waitpid(pids[i]);


    printf("All child finished and collected. Init is exiting...\n");
    exit();
}
```

Test Results:

I added a delay to allow the process tables to be observed. As a result, processes are interrupted frequently and may not have a chance to execute many of their instructions.





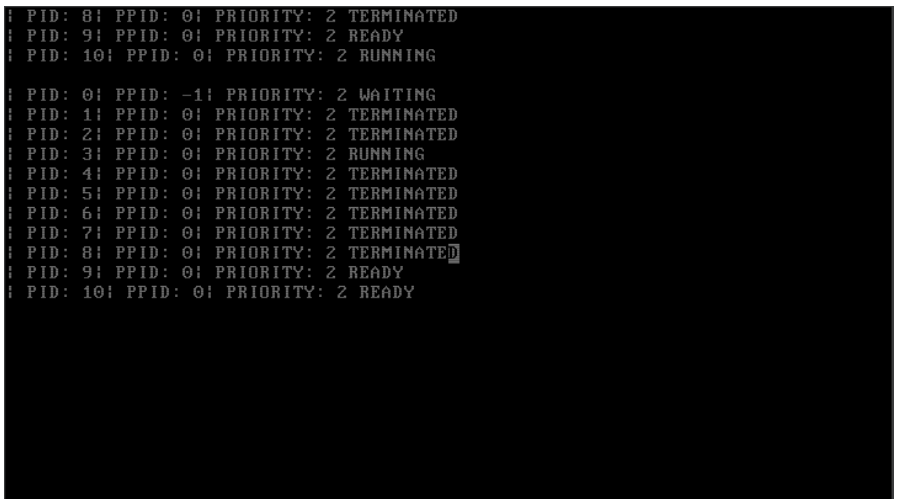
9)PART B

For this part, four different strategies are implemented.

First Strategy: This strategy involves selecting one of the programs and loading it into memory ten times using the fork system call. The parent process waits for all child processes to terminate. Additionally, a rand function is implemented to obtain random numbers. It uses **round-robin** scheduling.

```
int random_number = rand() % 4;


for(int i = 0; i < 10; i++)
{
    pids[i] = fork();
    if(pids[i] == 0)
    {
        switch(random_number)
        {
            case 0:
                collatz(9999999);
                break;
            case 1:
                binarySearch(arr, 10, 60);
                break;
            case 2:
                linearSearch(arr, 10, 60);
                break;
            case 3:
                long_running_program(1000);
                break;
            default:
                break;
        }
    }
}
for(int i = 0; i< num_process; i++)
    waitpid(pids[i]);
```

First Strategy Results:



Second Strategy: It involves randomly selecting two out of four programs and loading each selected program three times using the fork system call. The parent process waits until all child processes have terminated. It uses **round-robin** scheduling.

```
int random_numbers[2];
random_numbers[0] = rand() % 4;
do
{
    random_numbers[1] = rand() % 4;
} while (random_numbers[0] == random_numbers[1]);


for(int i = 0; i < 2; i++)
{
    for (int j = 0; j < 3; j++)
    {
        pids[i] = fork();
        if(pids[i] == 0)
        {
            switch(random_numbers[i])
            {
                case 0:
                    collatz(9999999);
                    break;
                case 1:
                    binarySearch(arr, 10, 60);
                    break;
                case 2:
                    linearSearch(arr, 10, 60);
                    break;
                case 3:
                    long_running_program(1000);
                    break;
                default:
                    break;
            }
        }
```

Second Strategy Results:

Third strategy: It loads each program once but assigns high priority (low number) to the collatz program by calling nice system call. This scenario utilizes a priority-based scheduling algorithm.

```
for (int i = 0; i < num_process; i++)
{
    pids[i] = fork();

    if (pids[i] == 0)
    {
        switch(i)
        {

            case 0:
                nice(Priority::HIGH);
                collatz(7542186342);
                break;
            case 1:
                binarySearch(arr, 25, 20);
                break;
            case 2:
                linearSearch(arr, 25, 20);
                break;
            case 3:
                long_running_program(10000);
                break;
```

After the fifth interrupt, all processes are set to have the same priority:

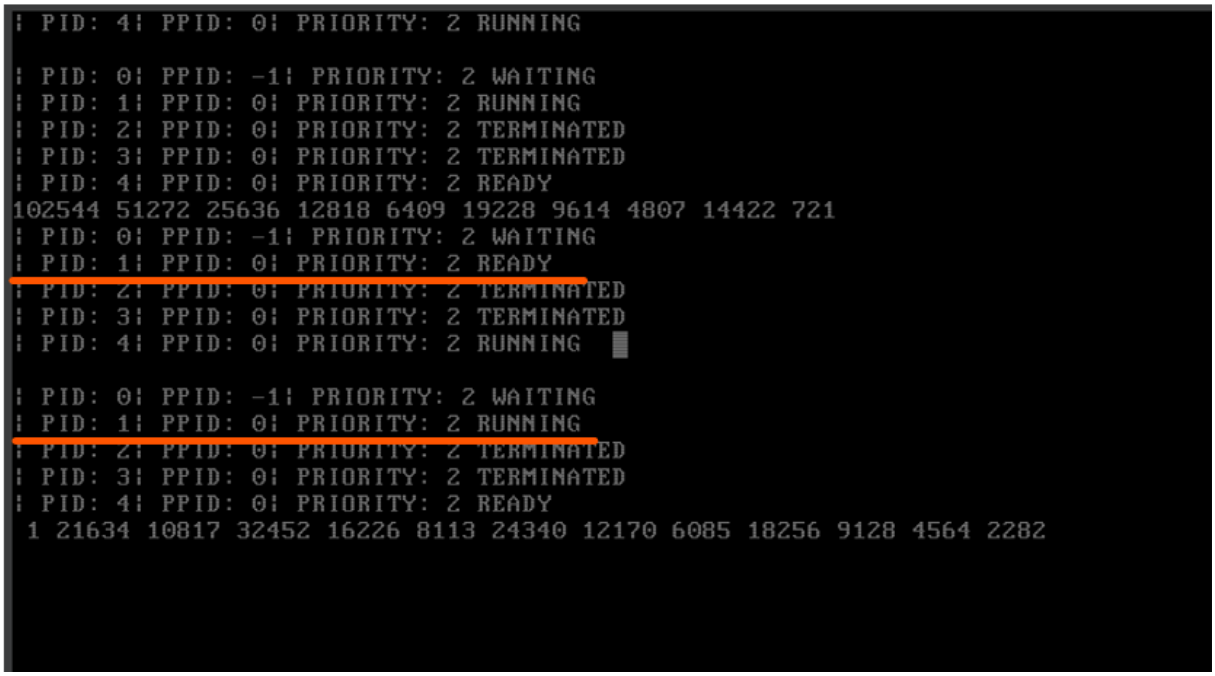
```
#ifdef PARTB_3
    if (interrupt_count == 5)
    {
        for (int i = 0; i < numTasks; i++)
            tasks[i]->priority = Priority::MEDIUM;
    }
    else if(interrupt_count < 5)
        interrupt_count++;

    currentTask = findNextTaskByPriority();
#endif
```

Third Strategy Results:

The priority for the Collatz program has been changed, giving it high priority and allowing it to execute more frequently than the other processes.

```
Welcome to My Operating System LOOS

¦ PID: 0¦ PPID: -1¦ PRIORITY: 2 RUNNING

¦ PID: 0¦ PPID: -1¦ PRIORITY: 2 READY
¦ PID: 1¦ PPID: 0¦ PRIORITY: 2 RUNNING
Priority changed to: 01
¦ PID: 0¦ PPID: -1¦ PRIORITY: 2 READY
¦ PID: 1¦ PPID: 0¦ PRIORITY: 1 RUNNING

------------(collatz)------------
-0477482501: -238741255 -5716223741 -858111877 1937533736 968766868 484383434 24
2191717 726575152 363287576 181643788 90821894 45410947 136232842 68116421
¦ PID: 0¦ PPID: -1¦ PRIORITY: 2 READY
¦ PID: 1¦ PPID: 0¦ PRIORITY: 1 RUNNING
204349264 102174632 51087316 25543658 12771829 38315488 19157744 9578872 4789436
 2394718 1197359 3592078 1796039 5388118 2694059
¦ PID: 0¦ PPID: -1¦ PRIORITY: 2 READY
¦ PID: 1¦ PPID: 0¦ PRIORITY: 1 RUNNING
8082178 4041089 12123268 6061634 3030817 9092452 4546226 2273113 6819340 3409670
 1704835 5114506
```

After the 5. Interrupt, all the programs have same priority



Fourth Strategy:

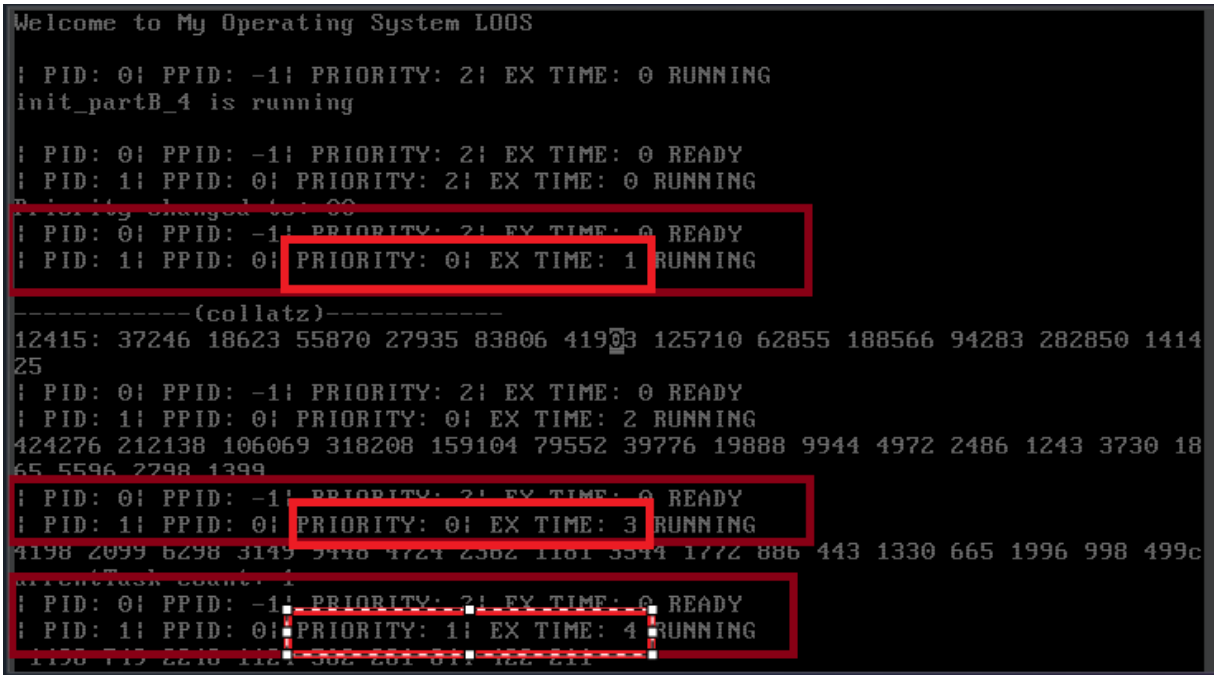
Collatz program is started with lowest priority number (highest priority) in this strategy:

```
for (int i = 0; i < num_process; i++)
{
    pids[i] = fork();

    if (pids[i] == 0)
    {
        switch(i)
        {

            case 0:
                nice(Priority::VERY_HIGH);
                collatz(7542186342);
                break;
            case 1:
                binarySearch(arr, 25, 20);
                break;
            case 2:
                linearSearch(arr, 25, 20);
                break;
            case 3:D
                long_running_program(10000);
                break;
            default:
                break;
        }
    }
```

When a context switch occurs, the interrupt number and the task execution number are incremented. The scheduler then checks if any program has executed too many times. To ensure fairness dynamically, it increases the priority number of such programs.

```
#ifdef PARTB_4
    if (interrupt_count == 5 )
    {
        // if it has been working more than 2 times, increase its priority
        interrupt_count = 0;
        if (tasks[currentTask]->executionTime > 2)
            tasks[currentTask]->priority = (Priority)((int)tasks[currentTask]->priority + 1);

    }
    else if(interrupt_count < 5)
        interrupt_count++;

    currentTask = findNextTaskByPriority();

    if (oldTask == currentTask)
        tasks[currentTask]->executionTime++;
    else
        tasks[currentTask]->executionTime = 0;
#endif
```

Fourth Strategy Results:



10) PART C

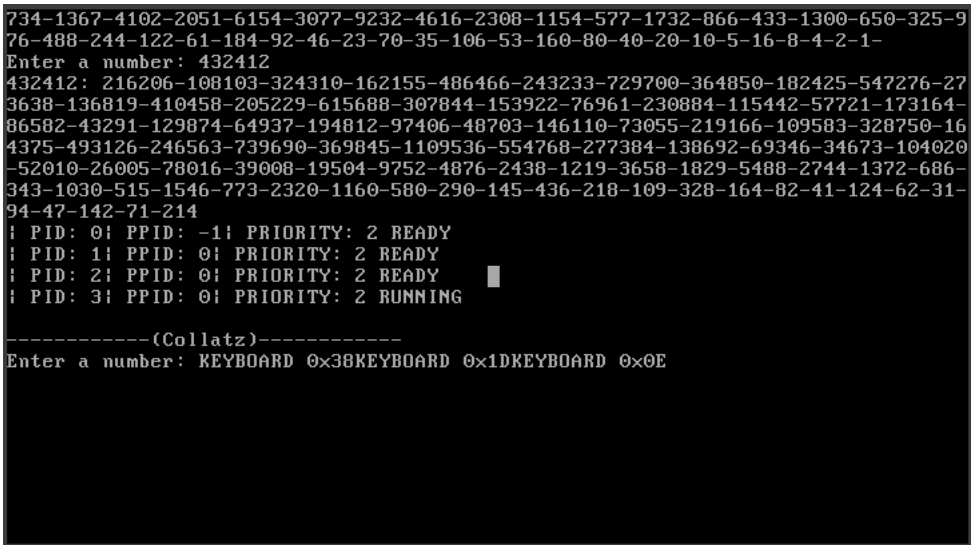
First Strategy: In this strategy, init process randomly chooses one of the programs and loads it multiple times, each time calling fork system call. These processes then enter an infinite loop, awaiting interactive input events. Processes disable the timer interrupts until an input is entered. Once an input is provided, interrupts are re-enabled, allowing a context switch to occur.

```
for(int i = 0; i <num_process; i++)
{
    pids[i] = fork();
    int random_number = rand() % num_process;
    if(pids[i] == 0)
    {
        switch(random_number)
            {
                case 0:
                    binarySearch_IO(arr, 10, 60);
                    break;
                case 1:
                    Collatz_IO();
                    break;
                case 2:
                    linearSearch_IO(arr, 10, 60);
                    break;
                case 3:
                    long_running_program(1000);
                    break;
                default:
                    break;
            }
    }
}
```

First Strategy Results:

Init choose randomly Collatz and it waits for keyboard input:

It blocked the interrupts until I entered input. Then it starts to execute, if an interrupt occur in execution, context switch happens:



Second Strategy: In this approach, child processes have different priority levels. While the collatz process waits for keyboard events, the binary search and linear search processes wait for mouse clicks. In this approach, I used the execution times of the processes to ensure fairness. If a process runs three times consecutively, it is scheduled to the next task regardless of priority.

I assigned high priority to keyboard events and low priority to mouse events using the nice system call as illustrated below.

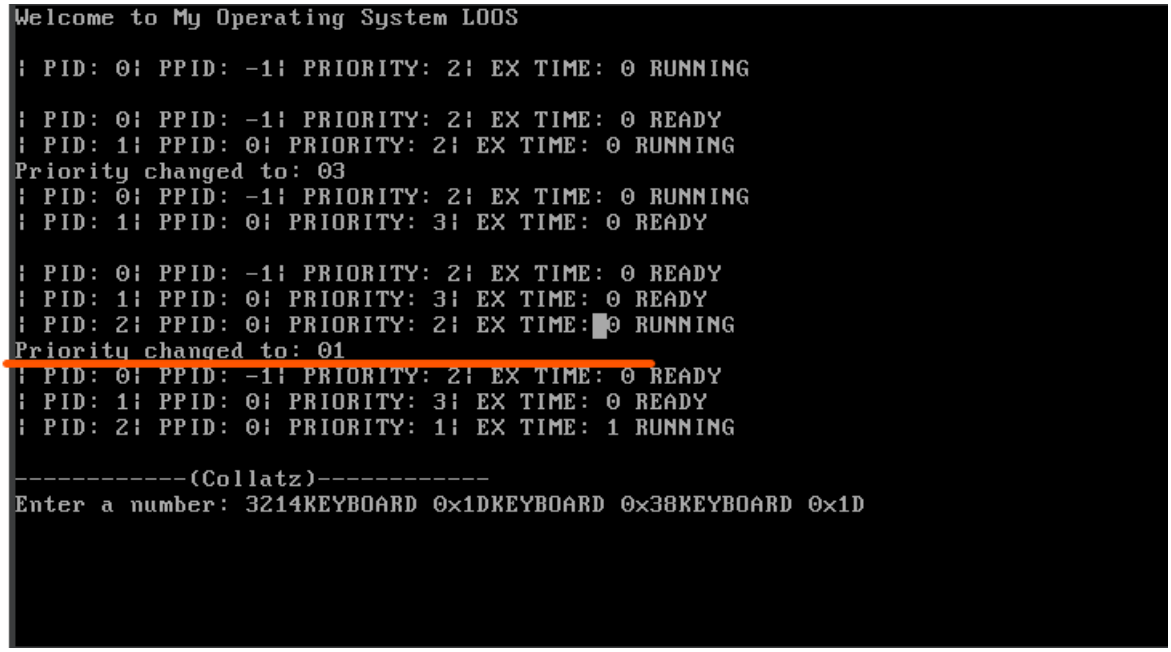
```
for(int i = 0; i < num_process; i++)
{
    pids[i] = fork();
    if(pids[i] == 0)
    {
        switch(i)
            {
                case 0:
                    nice(Priority::LOW);
                    binarySearch_IO(arr, 10, 60);
                    break;
                case 1:
                    nice(Priority::LOW);
                    long_running_program(1000);
                    break;
                case 2:
                    nice(Priority::LOW);
                    linearSearch_IO(arr, 10, 60);
                    break
                case 3:
                    nice(Priority::HIGH);
                    Collatz_IO();
                    break;
                default:
                    break;
            }
    }
```

enum myos::Priority

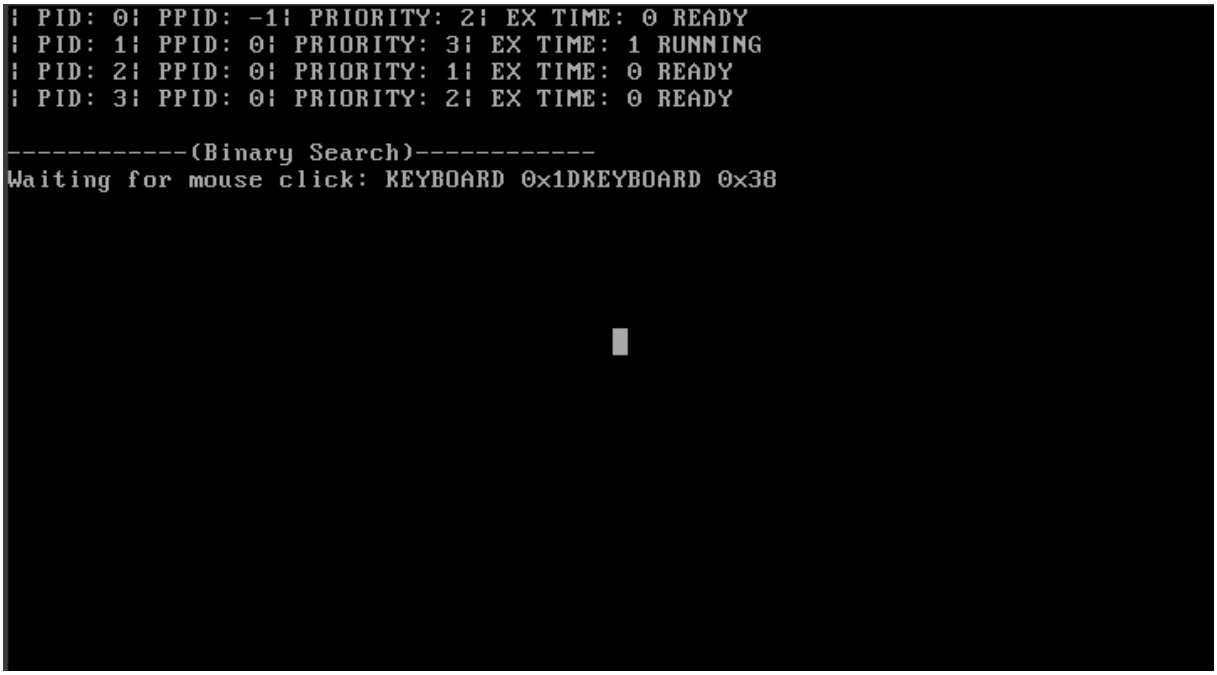
```
#ifdef PARTC_2
    // if the task has been working for 3 times, change the task
    if (oldTask == currentTask)
        tasks[currentTask]->executionTime++;

    if (tasks[currentTask]->executionTime == 3)
    {
        tasks[currentTask]->executionTime = 0;
        currentTask = findNextTask();
    }
    else
    {
        currentTask = findNextTaskByPriority();
    }

#endif
```

Second Strategy Results:

The Collatz process has high priority, not allowing the parent process to create other child processes for now.



After the Collatz process has run for three execution cycles, the scheduler switches to the next task regardless of priority and Binary search comes into play. I used dynamic priority scheduling approach here because Collatz is executed infinitely in Round-Robin scheduling.



11) Compiling and Running

Use make command to compile the OS. It will compiled with default mikrokerneş which test the system calls.

Default: Test for System Calls

```
$ make
```

Compile Options:

```
$make PART=-D<part_name>  PT=-D<visibility__of_process_table> SPEED = -
D<speed_of_interrupt>
```

part_name:

PARTA_1: For Part A strategy

PARTB_1: Part B 1. strategy

PARTB_2: Part B 2. Strategy

PARTB_3: Part B 3. Strategy

PARTB_4: Part B 4. Strategy

PARTC_1: Part C 1. Strategy

PARTC_2: Part C 2. Strategy

visibility_of_process_table:

SHOW: Print process table for each context switch

speed_of_interrupts:

SLOW: Implement context switches every 50 timer interrupts and print dummy in the process table print.