CSE 312 HW 1 Report

Ertugrul Kasikci
200104004097

Introduction

This report details the implementation and operation of "Ertugrul's Operating System". The system features include process forking, round-robin scheduling, and handling of keyboard and mouse interrupts. This document covers the methods used to implement these features, as well as the output results from various lifecycle strategies.

1 System Setup

To run the operating system, the command `make run` must be executed within the directory containing the makefile. The name of the OS in the makefile should match "Ertugrul's Operating System". The resulting `mykernel.iso` file must be added to a virtual machine via Virtual Box to boot the OS. Initial setup may result in errors which can be resolved by correctly adding the ISO file to Virtual Box.

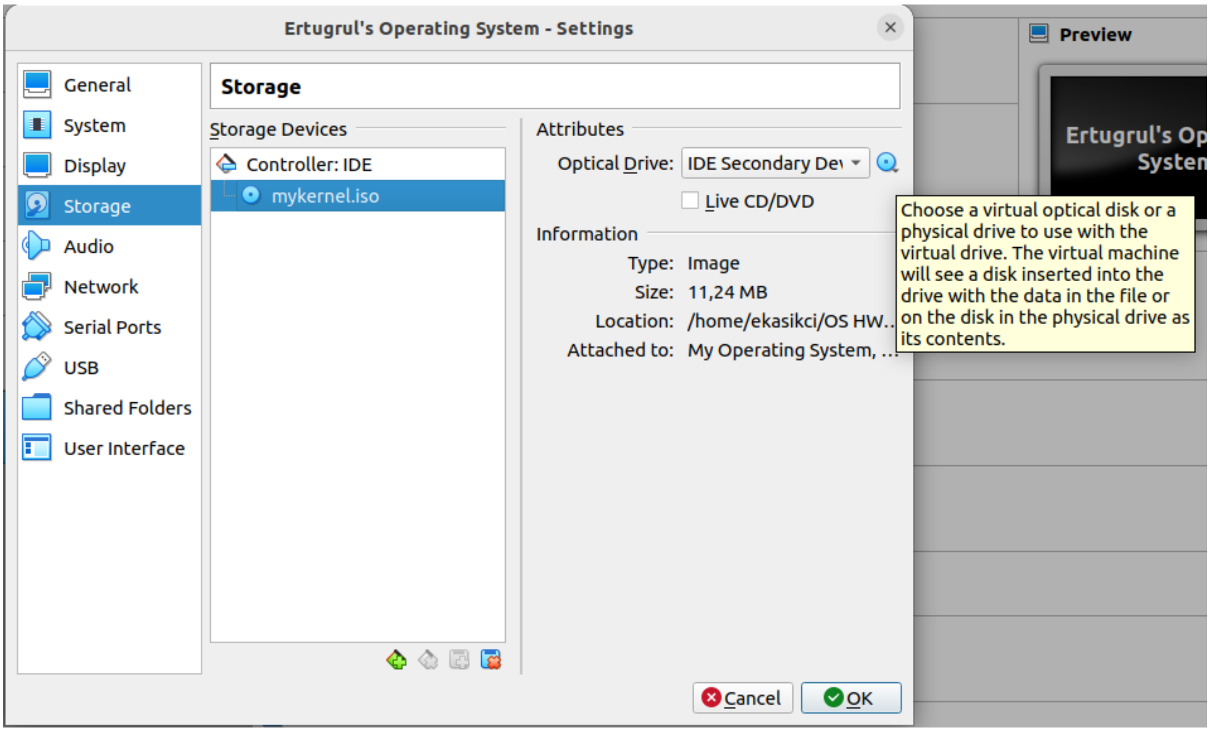

Figure 1: The ISO file is added as shown.

1.1 Makefile Settings

In order to use Virtual Box, I needed to use VirtualBoxVM keyword that is different from the source code we were provided.

```
install: mykernel.bin
    sudo cp $< /boot/mykernel.bin

run: mykernel.iso
    (killall VirtualBoxVM && sleep 1) || true
    VirtualBoxVM --startvm "Ertugrul's Operating System" &
```

Figure 2: VirtualBoxVM keyword is used.

2 Process Management

The OS implements a custom version of the `fork` function. Uses round robin scheduling default.

2.1 Fork Implementation

This function is designed to create a new process and immediately direct it to execute a specific function, identified as `entry_point`. This custom version extends the traditional UNIX fork operation by allowing the new process to begin execution at a predefined point in the program, rather than duplicating the parent process's execution state.

```
pid_t fork(void (*entry_point)(void))
{
    uint32_t ret = -1;

    __asm__ volatile(
        "movl %0, %%eax;"                // Load syscall number for fork into eax
        "movl %1, %%ebx;"                // Load function pointer (entry_point) into ebx
        "int $0x80;"                     // Trigger system call via interrupt
        :                                // No output operands
        : "r"(sys_fork), "r"(entry_point) // Inputs: syscall number, entry_point function
        : "eax", "ebx");                 // Registers eax and ebx will be modified
    __asm__ volatile("" : "=a"(ret));    // Store return value from eax into ret

    return ret;
}
```

Figure 3: fork implementation.

2.2 Forking Process of the Functions

Here 6 functions are forked. 4 of them are given in the homework document. I added two idle process to show the CPU switch between processes and my OS's ability to respond keyboard and mouse interrupts. They have basic while loops that loop infinitely.

```c
void initProcess()
{
    printf("Init: Init process is started\n");
    const uint32_t numOfProcess = 4;
    pid_t pids[numOfProcess];
    void (*process[])() = {collatz, long_running_program_wrapper, idleProcess, idleProcess2};

    printf("Init: Forking processes is started\n");
    for (uint32_t i = 0; i < numOfProcess; ++i)
        pids[i] = fork(process[i]);
    printf("Init: Forking processes is finished\n");

    printf("Init: Waiting for all child processes to finish\n");
    for (uint32_t i = 0; i < numOfProcess; ++i)
        waitpid(pids[i]);
    printf("Init: All child processes are finished\n");

    exit(exit_success);
}
```

Figure 4: Init process with forking of the functions.

2.3 long_running_program Function

In order to pass parameter to long_running_program, I created a new function called long_running_program_wrapper.

```c
void long_running_program_wrapper()
{
    int32_t result = long_running_program(n);
    int32_t printArr[] = {result};
    print("Long running program: Result: %d\n", 1, printArr);
    exit(exit_success);
}
```

Figure 5: long_running_program_wrapper implementation.

This approach makes it possible to run functions with parameters via fork call. Fork call does not accept functions with returning value and parameters.

```c
int32_t long_running_program(int32_t n)
{
    int32_t result = 0;
    for (uint32_t i = 0; i < n; i++)
    {
        for (uint32_t j = 0; j < n; j++)
        {
            result += i * j;
        }
    }
    return result;
    exit(exit_success);
}
```

Figure 6: long_running_program implementation.

3 System Call Implementation

This section explains the implementation of crucial system calls in the OS. These system calls facilitate essential operations such as process creation, execution, waiting, and termination. The functions utilize inline assembly to interact directly with the kernel via software interrupts.

3.1 fork Implementation

The `fork` function initiates the creation of a new process. The entry point of the new process is specified by the `entry_point` function pointer. The system call number for `fork` is loaded into the `eax` register, and the `entry_point` address is loaded into the `ebx` register. An interrupt (`int $0x80`) is then triggered to execute the syscall.

```
pid_t fork(void (*entry_point)(void))
{
    uint32_t ret = -1;

    __asm__ volatile(
        "movl %0, %%eax;"                   // Load syscall number for fork into eax
        "movl %1, %%ebx;"                   // Load function pointer (entry_point) into ebx
        "int $0x80;"                        // Trigger system call via interrupt
        :                                   // No output operands
        : "r"(sys_fork), "r"(entry_point) // Inputs: syscall number, entry_point function
        : "eax", "ebx");                    // Registers eax and ebx will be modified
    __asm__ volatile("" : "=a"(ret));      // Store return value from eax into ret

    return ret;
}
```

Figure 7: Assembly code for the `fork` system call.

3.2 execve Implementation

The `execve` function executes a new program specified by `path`. The arguments to the program are passed via `argv`, and the environment variables are passed via `envp`. Each of these pointers is loaded into the respective registers (`ebx`, `ecx`, `edx`) before the system call is invoked via an interrupt.

```
void execve(const char *path, char *const argv[], char *const envp[])
{
    __asm__ volatile(
        "movl %0, %%eax\n\t"                                    // Load syscall number for execve into eax
        "movl %1, %%ebx\n\t"                                    // Load path pointer into ebx
        "movl %2, %%ecx\n\t"                                    // Load argv pointer into ecx
        "movl %3, %%edx\n\t"                                    // Load envp pointer into edx
        "int $0x80"                                             // Trigger system call via interrupt
        :                                                       /* no outputs */
        : "i"(sys_execve), "m"(path), "m"(argv), "m"(envp) // Inputs: syscall number, path, argv, envp
        : "eax", "ebx", "ecx", "edx");                          // Registers eax, ebx, ecx, and edx will be modified
}
```

Figure 8: Assembly code for the `execve` system call.

3.3 `waitpid` Implementation

The `waitpid` function allows a process to wait for another process to terminate. The process ID (`pid`) is loaded into `ebx`, and the system call is triggered, with the process status being continually checked until the process terminates.

```
void waitpid(uint32_t pid)
{
    Process::Status ret = Process::Running;

    while (ret != Process::Terminated)
    {
        __asm__ volatile(
            "movl %0, %%eax;"               // Load syscall number for waitpid into eax
            "movl %1, %%ebx;"               // Load process ID (pid) into ebx
            "int $0x80;"                    // Trigger system call via interrupt
            :                               // No output operands
            : "r"(sys_waitpid), "r"(pid)    // Inputs: syscall number, process ID
            : "eax", "ebx");                // Registers eax and ebx will be modified
        __asm__ volatile("" : "=a"(ret)); // Store process status from eax into ret
    }
}
```

Figure 9: Assembly code for the `waitpid` system call.

3.4 `exit` Implementation

Finally, the `exit` function terminates a process and exits the program. The exit status is loaded into `ebx`, and the system call is made to effectively stop the process's execution.

```
void exit(Exit exit)
{
    __asm__ volatile(
        "movl %0, %%eax;"            // Load syscall number for exit into eax
        "movl %1, %%ebx;"            // Load exit status into ebx
        "int $0x80;"                 // Trigger system call via interrupt
        :                            // No output operands
        : "r"(sys_exit), "r"(exit)   // Inputs: syscall number, exit status
        : "eax", "ebx");             // Registers eax and ebx will be modified
    while (true)                     // Prevent any further execution after exit
        ;
}
```

Figure 10: Assembly code for the `exit` system call.

4 Output Visibility

With the current configuration, the output is not displayed correctly because shortly after the operating system starts, the existing output is replaced by new content. The idle processes continually print output, which contributes to this issue. To view the output correctly, the `numOfProcess` parameter can be set to 4. This adjustment eliminates the idle functions, allowing the output to remain visible. Below, I will demonstrate the results both with and without the idle functions.

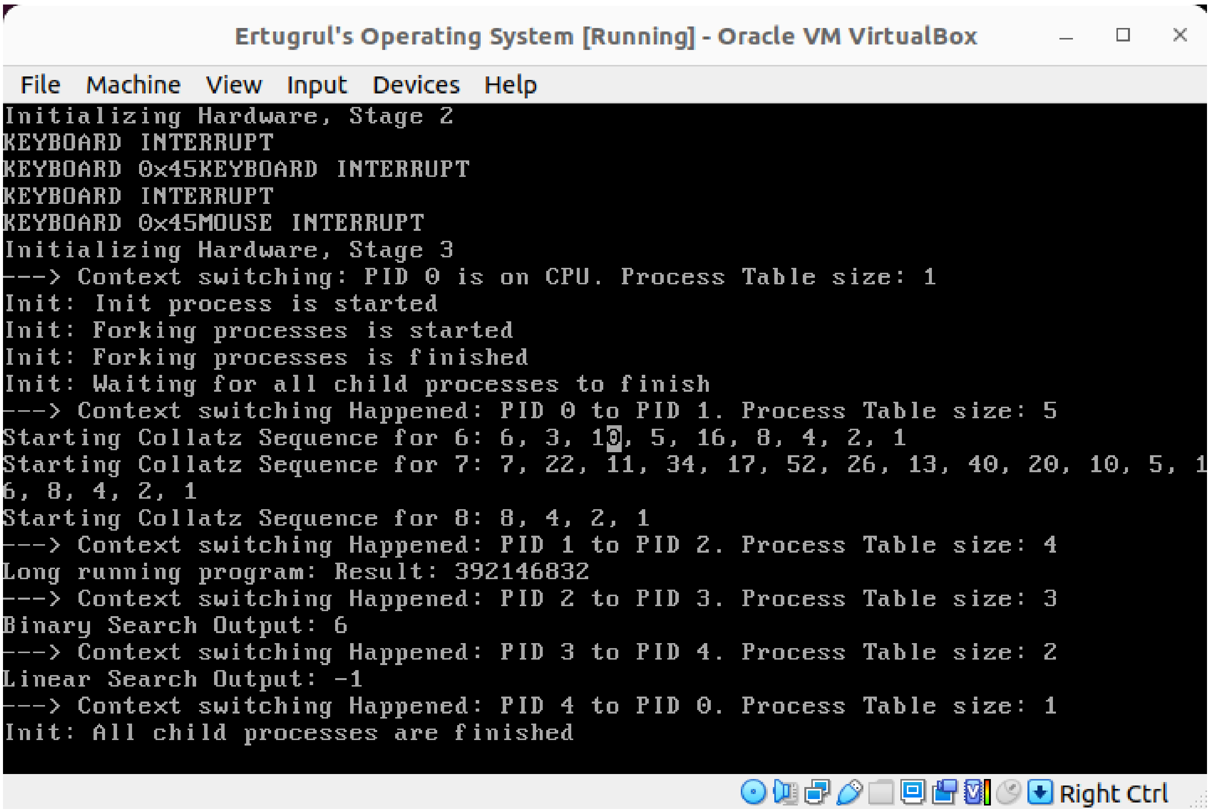

Figure 11: Result without idle funcitons.

In this configuration, all outputs are clearly visible. Only three values for the Collatz function are used here because using more would overcrowd the screen. Once the screen is full, any new output will overwrite the existing content, starting again from the top.

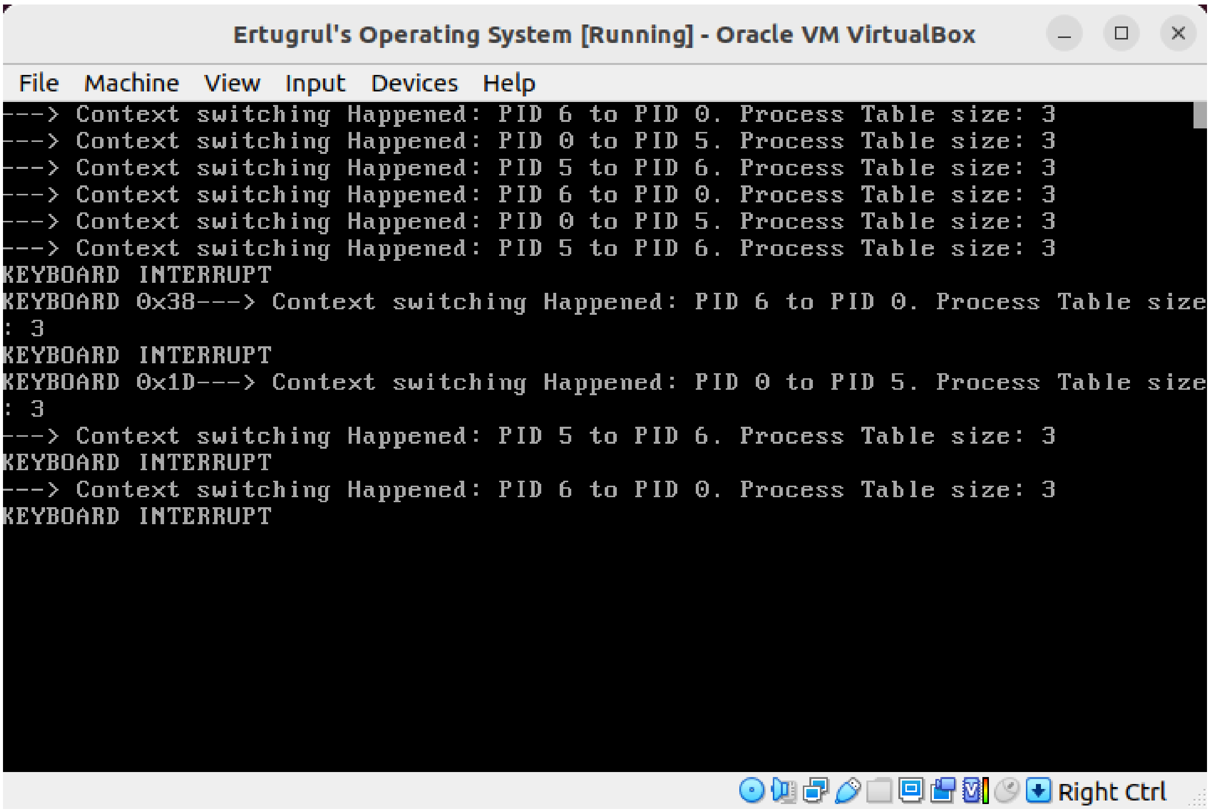

Figure 12: Result with idle funcitons.

With idle functions running, context switching occurs continuously. Despite this, the operating system remains responsive to both keyboard and mouse interrupts. Note that capturing a mouse interrupt screenshot was impractical due to the rapid context switching.

5 Round Robin Scheduling

This section details the implementation of round robins scheduling within the ProcessTable.cpp file. The system demonstrates context switching by printing relevant outputs during operation.

```cpp
// Schedules the next process to run based on the Round Robin algorithm.
CPUState *ProcessTable::Schedule(CPUState *cpustate)
{
    if (numProcesses <= 0)
        return cpustate;

    if (currentProcess >= 0)
    {
        table[currentProcess].cpustate = cpustate; // Store the old CPU state

        if (table[currentProcess].status == Process::Running)
            table[currentProcess].status = Process::Ready;
    }
    // Store the current process to print screen
    int32_t printScreen[3];
    printScreen[0] = currentProcess;

    // Get the next process to run (Round Robin)
    do
    {
        if (++currentProcess >= numProcesses)
            currentProcess = 0;

    } while (table[currentProcess].status == Process::Terminated); // Skip terminated processes (they are not ready to run)

    table[currentProcess].status = Process::Running;

    // Store the next process to print screen
    printScreen[1] = currentProcess;

    // Print the context switching information
    if (printScreen[0] == -1) // First process
    {
        printScreen[0] = printScreen[1];
        printScreen[1] = GetSize();
        print("---> Context switching: PID %d is on CPU. Process Table size: %d\n", 2, printScreen);
    }
    else if (printScreen[0] == printScreen[1]) // The same process is se
        // print("---> Context switching: PID %d is on CPU STILL\n", 1, printScreen);
        ;
    else // Context switching
    {
        printScreen[2] = GetSize();
        print("---> Context switching Happened: PID %d to PID %d. Process Table size: %d\n", 3, printScreen);
    }

    return table[currentProcess].cpustate;
}
```

Figure 13: Round Robin Scheduling implementation.

5.1 Round Robin Scheduling Implementation Details

If `numProcess` is 0 or less we return `cpustate` directly. This means we have no other process remaining. Otherwise we store the `cpustate` inside the current process and set the current process to `Ready` state from `Running` state.

The next process which is not `Terminated` is selected via the do-while loop. The process table information printed after determining if this process is the same with previous one.

6 Interrupt Handling

This section outlines how the operating system handles keyboard and mouse interrupts, including how these events affect output behaviors.

6.1 Interrupts Behavior

When an interrupt occurs, the corresponding character is printed to the screen, and the mouse remains responsive as long as the operating system is running. Rather than printing a straightforward message like "INTERRUPT" upon the occurrence of an interrupt, this task is delegated to the `interruptAwaitingProcess` function. This function is responsible for notifying the user by printing messages, as detailed in Part C of the homework.

```
void interruptAwaitingProcess()
{
    printf("Interrupt awaiting process is started...\n");
    while (1)
    {
        // Wait for a keyboard interrupt
        if (myos::drivers::KeyboardDriver::returnFromKeyboardDriver)
        {
            printf("Keybord interrup is detected by the process. The process is terminated\n");
            myos::drivers::KeyboardDriver::returnFromKeyboardDriver = false;
            exit(exit_success);
        }

        // Wait for a mouse interrupt
        if (myos::drivers::MouseDriver::returnFromMouseDriver)
        {
            printf("Mouse interrup is detected by the process. The process is terminated\n");
            myos::drivers::MouseDriver::returnFromMouseDriver = false;
            exit(exit_success);
        }
    }
}
```

Figure 14: Implementation of `interruptAwaitingProcess`.

To facilitate communication about interrupt occurrences to `interruptAwaitingProcess`, I implemented static boolean variables for both keyboard and mouse interrupts.

```
class KeyboardDriver : public myos::hardwarecommunication::InterruptHandler, public Driver
{
    myos::hardwarecommunication::Port8Bit dataport;
    myos::hardwarecommunication::Port8Bit commandport;

    KeyboardEventHandler* handler;
public:
    KeyboardDriver(myos::hardwarecommunication::InterruptManager* manager, KeyboardEventHandler *handler);
    ~KeyboardDriver();
    virtual myos::common::uint32_t HandleInterrupt(myos::common::uint32_t esp);
    static bool returnFromKeyboardDriver; // Informs about keyboard interrupt
    virtual void Activate();
};
```

Figure 15: Declaration of `returnFromKeyboardDriver`.

The `returnFromKeyboardDriver` variable is set to true when a keyboard interrupt occurs.

```
uint32_t KeyboardDriver::HandleInterrupt(uint32_t esp)
{
    // printf("KEYBOARD INTERRUPT\n");
    uint8_t key = dataport.Read();

    returnFromKeyboardDriver = true;
```

Figure 16: Usage of `returnFromKeyboardDriver` when a keyboard interrupt occurs.

This variable signals to `interruptAwaitingProcess` that an interrupt has occurred, and is reset to false within that function after the notification is processed.

Similarly, the variable called `returnFromMouseDriver` functions in the same manner for mouse interrupts.

7 Lifecycle Scenarios

This section illustrates the operational principles of my operating system through different lifecycle scenarios as described in the homework PDF.

7.1 Part A Lifecycle

In the Part A lifecycle strategy, each program is loaded three times and initiated. These processes then enter an infinite loop, continuing until all processes have terminated.

```
void initProcess()
{
    printf("Init: Init process is started\n");

    // Part A lifecycle
    const uint32_t numOfProcess = 12;
    pid_t pids[numOfProcess];
    void (*process[])() = {collatz, long_running_program_wrapper, binarySearch, linearSearch,
                           collatz, long_running_program_wrapper, binarySearch, linearSearch,
                           collatz, long_running_program_wrapper, binarySearch, linearSearch};
```

Figure 17: Setting of lifecycle inside the `initProcess` function.

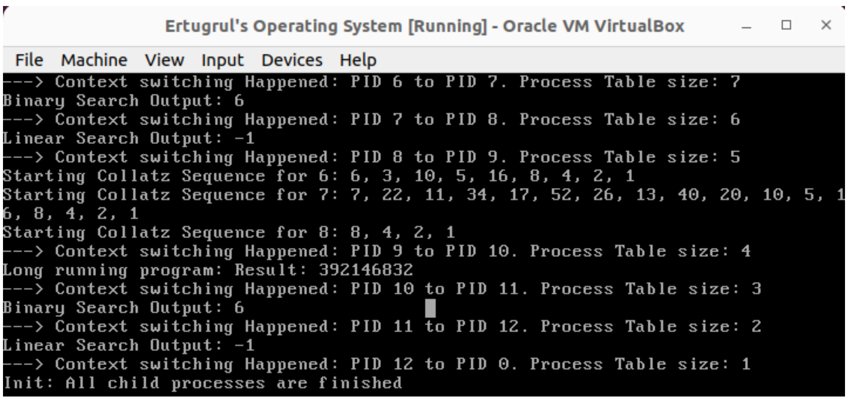

Figure 18: Output from the OS during Part A lifecycle.

7.2 Part C Random Process Spawning

This strategy utilizes the function `interruptAwaitingProcess`, which was previously discussed in the Interrupt Handling section. This function operates in an infinite loop, processing keyboard and mouse interrupts by printing notifications and then terminating. This mechanism demonstrates dynamic process handling based on user input.

```cpp
void interruptAwaitingProcess()
{
    printf("Interrupt awaiting process is started...\n");
    while (1)
    {
        // Wait for a keyboard interrupt
        if (myos::drivers::KeyboardDriver::returnFromKeyboardDriver)
        {
            printf("Keybord interrup is detected by the process. The process is terminated\n");
            myos::drivers::KeyboardDriver::returnFromKeyboardDriver = false;
            exit(exit_success);
        }

        // Wait for a mouse interrupt
        if (myos::drivers::MouseDriver::returnFromMouseDriver)
        {
            printf("Mouse interrup is detected by the process. The process is terminated\n");
            myos::drivers::MouseDriver::returnFromMouseDriver = false;
            exit(exit_success);
        }
    }
}
```

Figure 19: Implementation of `interruptAwaitingProcess`.

```cpp
// Random Process Spawning with Interactive Input Handling Strategy
const uint32_t numOfProcess = 10;
pid_t pids[numOfProcess];
void (*process[])() = {interruptAwaitingProcess, interruptAwaitingProcess, interruptAwaitingProcess,
                       interruptAwaitingProcess, interruptAwaitingProcess, interruptAwaitingProcess,
                       interruptAwaitingProcess, interruptAwaitingProcess, interruptAwaitingProcess,
                       interruptAwaitingProcess};
```

Figure 20: `interruptAwaitingProcess` loaded 10 times into memory and executed simultaneously.

```
Ertugrul's Operating System [Running] - Oracle VM VirtualBox
File  Machine  View  Input  Devices  Help
Keybord interrup is detected by the process. The process is terminated
---> Context switching Happened: PID 7 to PID 8. Process Table size: 4
Mouse interrup is detected by the process. The process is terminated
---> Context switching Happened: PID 8 to PID 9. Process Table size: 3
Mouse interrup is detected by the process. The process is terminated
---> Context switching Happened: PID 9 to PID 10. Process Table size: 2
Keybord interrup is detected by the process. The process is terminated
---> Context switching Happened: PID 10 to PID 0. Process Table size: 1
Init: All child processes are finished
```

Figure 21: Output from the OS under the Part C lifecycle.