Implementing Operating System and System Calls with Cpp

Hikmet Mete Varol

Contents

1 Introduction

In this project, the Primitive Operating System implemented by Viktor Engelmann was worked on. New features requested from us have been added to the existing operating system features in this system. The report describes in detail the existing and newly added operating system improvements.

2 Operating System Kernel

The operating system kernel (kernel.cpp) performs basic functions such as managing hardware resources, handling interrupts, and providing system calls.

2.1 Core components of the kernel

2.1.1 GlobalDescriptorTable

Global Descriptor Table is used to manage memory segmentation.This table allows the operating system to safely access different memory segments.

2.1.2 TaskManager

TaskManager is the structure that enables multitasking in the operating system. This class keeps track of all processes and manages the status, priority level and other parameters of each process. TaskManager also controls context switching and decides which process to run.

2.1.3 InterruptManager

InterruptManager manages interrupts generated by hardware or software.

2.1.4 SyscallHandler

SyscallHandler is the class that handles system calls. It enables switching from user mode to kernel mode and enables user programs to access operating system resources.

2.1.5 DriverManager

DriverManager manages hardware drivers in the operating system kernel. It provides communication between hardware such as keyboard and mouse and the operating system.

```
extern "C" void kernelMain(const void* multiboot_structure, uint32_t
{
    printf("Hello from My OS --- Hikmet Mete Varol -- 1801042608\n");
    printf("----------------------------------------------------\n");

    GlobalDescriptorTable gdt;

    // ************************************* TESTS **************
    TaskManager taskManager(&gdt);


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



    // ********************** KEYBOARD ***************************
    DriverManager drvManager;
```

Figure 1: Kernel main

3 Operating System Functions

3.1 Multiprogramming

3.1.1 Process Management and Multitasking

```
//***************************** TASK MANAGER ******************************************

class TaskManager {

private:

    static const int MAX_TASKS = 256;

    Task tasks[MAX_TASKS];                                  // PROCESS_TABLE
    GlobalDescriptorTable *gdt;                             // GDT

    int numTasks;                                           // NUMBER_OF_TASKS
    int currentTask;                                        // CURRENT_TASK
    int interrupt_count = 0;

    int aging_pid = -1;


// ____________________________________________________________________________

public:

    static common::uint32_t next_pid;                       // STATIC NEXT_PID

    TaskManager(GlobalDescriptorTable *gdt);                // CONSTRUCTOR

    ~TaskManager();                                         // DESTRUCTOR

    // ************************** SYSCALLS ***********************

    void fork(CPUState* parentState);                        // FORK SYSCALL

    void exit();                                            // EXIT SYSCALL

    common::uint32_t execve(void (*entryPoint)());          // EXECVE SYSCALL
```

Figure 2: TaskManager Class

We manage all our processes with the TaskManager class in the multitasking.cpp file. This class contains a task array and stores all processes in this array. This class also contains a method called "AddTask", with which we can create a new process (Later, another way, the "fork" system call, will also be examined.). New processes can be created with AddTask and fork methods. In this project, all processes were created using fork. The "Schedule" function, where the schedule operation is performed, is also located in this class. All other main system call functions where processes are managed are also methods of this class. These features of the TaskManager class allow us to do multitasking.

3.1.2 Process Table

```
class TaskManager {

private:

    static const int MAX_TASKS = 256;

    Task tasks[MAX_TASKS];                                  // PROCESS_TABLE
```

Figure 3: tasks array

The array in which we store all tasks in the TaskManager class refers to the Process table. All processes are located in this array and are selected and processed with the Schedule method.

Pid	PPid	State	Priority
0	-1	Running	-1
1	0	Terminated	-1
2	0	Terminated	-1
3	0	Terminated	-1
4	0	Terminated	-1
5	0	Terminated	-1
6	0	Terminated	-1

Figure 4: Sample Process Table

3.1.3 Process

```
//****************************** TASK (PROCESS) ****************************************

class Task {

friend class TaskManager;

private:

    CPUState* cpustate;                                     // CPU_STATE
    common::uint8_t stack[4096];                            // STACK
    int eax_value;                                          // EAX_VALUE

    common::uint32_t pid;                                   // PID
    common::uint32_t ppid;                                  // P_PID
    int priority;                                   // PRIORITY
    State state;                                            // STATE

    common::uint32_t waitingProcess = -1;                       // WAITING_PID
```

Figure 5: Task Class

Each Task class stored in the tasks array represents a process. Task class contains all the fields that a process must have (pid, ppid, priority, cpustate, stack, etc.).

3.2 System Calls

3.2.1 POSIX System Calls

void sysfork( ) and int sysfork_return( )

```
void sysfork() {
    asm ("int $0x80" : : "a" (SYS_FORK));
}

int sysfork_return(){
    int ebx_value;
    asm("int $0x80" :"=b" (ebx_value): "a" (SYS_DOUBLE_RETURN));
    return ebx_value;
}
```

```
case SYS_FORK:
    taskManager->fork(cpu);
    break;

case SYS_DOUBLE_RETURN:
    cpu->ebx = taskManager->getEaxVal();
    break;
```

Figure 6: sysfork and sysfork_return

The 0x80 interrupt vector is used for software interrupts in Linux-like systems. We perform system calls using the 0x80 interrupt vector.

In the first image, the call functions for our system call can be seen in the kernel.cpp file, and in the second image, the relevant TaskManager main functions in the SyscallHandler can be seen. We perform the operation by loading the number of the system call we want to call into the eax register.

```
void TaskManager::fork(CPUState* parentState) {

    Task* parentTask = &tasks[currentTask];

    if (numTasks >= MAX_TASKS) {
        parentTask->eax_value = -1; // double return
        return;
    }

    Task* childTask = &tasks[numTasks];

    childTask->createProcess(gdt, (void(*)())parentTask->cpustate->eip,next_pid,parentTask->pid);

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

    *(childTask->cpustate) = *parentState;

    childTask->cpustate->eax = 0;
    parentState->eax = childTask->getPID();

    // ------------------- double return -----------------
    childTask->eax_value = 0;
    parentTask->eax_value = childTask->getPID();
    // ------------------- double return -----------------

    numTasks++,next_pid++;

}
```

Figure 7: fork implementation

With the fork system call, we create a new process as in the original fork. We set a new pid for the child process we created and the pid of the parent process as ppid. Other properties are set to be exactly the same as the parent.

Linux Fork Double Return Feature

It is very difficult to fully implement the double return feature in the Linux fork in our system. The realization of this mechanism in Linux can actually be called the magic of the real fork system call. While the Linux fork is performing this process, it stops exactly at the point marked with an asterisk in the picture below and continues as two separate processes. And as a result, a pid value other than 0 for the child process and 0 for the parent process is returned from the fork function.

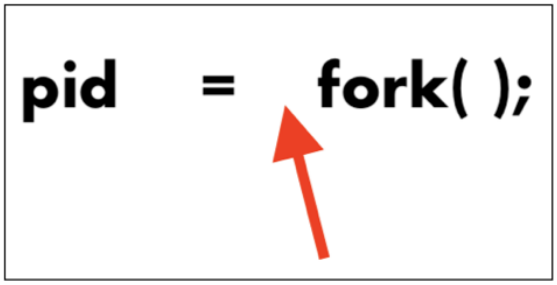


Figure 8: double return stop location

My Solution for Double Return Feature

```
void sysfork() {
    asm ("int $0x80" : : "a" (SYS_FORK));
}

int sysfork_return(){
    int ebx_value;
    asm("int $0x80" :"=b" (ebx_value): "a" (SYS_DOUBLE_RETURN));
    return ebx_value;
}
```

```
case SYS_FORK:
    taskManager->fork(cpu);
    break;

case SYS_DOUBLE_RETURN:
    cpu->ebx = taskManager->getEaxVal();
    break;
```

Figure 9: sysfork and sysfork_return

It is possible to fully implement the functionality of this feature, which we cannot fully implement. For this, I used a second system call that implements the double return feature. In the "sysfork_return" system call fork function, we save the value in the eax register, which is set differently for the child and parent, to the ebx register and we can use the double return feature in the program.

Sample Runs and Results

```
void forktask_3() {
    printf("\nfork() test:\n");

    sysfork();

    // syscall for double return feature
    int pid = sysfork_return();

    if(pid == 0){
        printf("\nHello from child = ");
        print_integer(getpid());
    }else{
        printf("\nHello from parent = ");
        print_integer(getpid());
    }

    while(1);
}
```

```
fork() test:

Hello from parent = 0

Hello from child = 1
```

```
void forktask_4() {
    printf("\nThree times fork() test:\n");
    printf("pid - message - getpid()\n");

    sysfork();
    sysfork();
    sysfork();

    int pid = sysfork_return();

    if(pid == 0){
        printf("\n");
        print_integer(pid);
        printf(" Hello from child  = ");
        print_integer(getpid());
        //sleep(1);
        sysexit();
    }else{
        printf("\n");
        print_integer(pid);
        printf(" Hello from parent = ");
        print_integer(getpid());
        //sleep(1);
        sysexit();
    }

    //sysprintf(" ");
    while(1);
}
```

```
Three times fork() test:
pid - message - getpid()

3 Hello from parent = 0
5 Hello from parent = 1
6 Hello from parent = 2
0 Hello from child  = 3
7 Hello from parent = 4
0 Hello from child  = 5
0 Hello from child  = 6
0 Hello from child  = 7
```

Figure 10: Sample run fork

int getpid( )

```
int getpid(){
    int ecx_value;
    asm("int $0x80" :"=c" (ecx_value): "a" (SYS_GETPID));
    return ecx_value;
}
```

```
case SYS_GETPID:
    cpu->ecx = taskManager->getPID();
    break;
```

```
int getPID(){return tasks[currentTask].getPID();}          // GET_PID SYSCALL
```

Figure 11: getpid syscall

We obtain the pid value of the current process with the getpid( ) system call. Pid values start from zero and increase by 1 each time a new process is added.

Unlike the double return value in the fork function, the processes' own pid values are stored in ecx registers.

void sysexit( )

```
void sysexit() {
    asm ("int $0x80" : : "a" (SYS_EXIT));
}
```

```
case SYS_EXIT:
    taskManager->exit();
    break;
```

```
void TaskManager::exit() {

    Task* currentTask = &tasks[this->currentTask];

    // terminate current task
    currentTask->state = Terminated;

    // wake up parent
    for (int i = 0; i < numTasks; i++) {
        if (tasks[i].waitingProcess == currentTask->pid && tasks[i].state == Blocked) {
            tasks[i].state = Ready;
        }
    }

    // force reschedule
    asm volatile("int $0x20");
}
```

Figure 12: exit syscall

The current process is terminated with the exit system call. For this, the state value of the current process is set to Terminated. In addition, processes waiting for this process to end are woken up by switching to the Ready state so that they can be processed by Scheduler.

void waitpid(int pid)

```
void waitpid(int pid){
    asm("int $0x80" : : "a" (SYS_WAITPID), "b" (pid));
}
```

```
case SYS_WAITPID:
    taskManager->waitpid(cpu->ebx);
    break;
```

```
int TaskManager::waitpid(common::uint32_t pid) {

    int index = findTaskByPID(pid);

    if (index == -1) return -1;  // Process not found

    Task& child = tasks[index];

    if (child.getState() == Terminated) {
        return 0; // Child already terminated
    }

    // Set the current task to Blocked state
    tasks[currentTask].setState(Blocked);

    // Set current task as waiting process of the child
    child.setWaitingProcess(tasks[currentTask].getPID());

    // force a reschedule
    asm volatile("int $0x20");

    return 0;
}
```

Figure 13: waitpid syscall

With the waitpid system call, we make the process wait for another process whose pid is known. For this, while we put the current process in Blocked state, we also store the pid value of the suspended process in the process that keeps it waiting. This allows us to wake up the process waiting in case of exit.

Sample Runs and Results

```
void waitpidtask_2() {
    printf("\nThree times fork() test with waitpid:\n");

    sysfork();
    sysfork();
    sysfork();

    int pid = sysfork_return();

    // childs
    if(pid == 0){
        printf("\n");
        printf("Hello from child = ");
        print_integer(getpid());
        sysexit();          // terminate childs
    }else{
        waitpid(pid);       // wait childs
        printf("\n");
        printf("Hello from parent = ");
        print_integer(getpid());
    }

    while(1);
}
```

```
Three times fork() test with waitpid:

Hello from child = 3
Hello from child = 5
Hello from child = 6
Hello from child = 7
Hello from parent = 0
Hello from parent = 1
Hello from parent = 2
Hello from parent = 4
```

Figure 14: waitpid sample run

void sysexecve(void entrypoint( ))

```
void sysexecve(void entrypoint()){
    asm("int $0x80" : : "a" (SYS_EXECVE), "b" ((uint32_t)entrypoint));
}
```

```
case SYS_EXECVE:
    entrypoint = cpu->ebx;
    esp = taskManager->execve((void (*)())entrypoint);
    break;
```

```
common::uint32_t TaskManager::execve(void (*entryPoint)()) {

    Task* currentTask = &tasks[this->currentTask];

    // set process with new entrypoint
    currentTask->createProcess(this->gdt, entryPoint, currentTask->pid, currentTask->ppid, currentTask->priority);

    return (uint32_t)currentTask->cpustate;
}
```

Figure 15: execve syscall

With the execve system call, we determine a new entry point for the process and enable it to run the desired function.

We send it to the new entry point via the ebx register. Then, after updating the process with this entrypoint, we replace the stack pointer (esp) with the updated CPU state of the process.

Sample Runs and Results

```
void execvetask_1() {
    printf("\nchild-> execve(hello_from_execve) test:\n");

    sysfork();

    int pid = sysfork_return();

    if(pid == 0){
        sysexecve(hello_from_execve);

        printf("\nHello from child = ");
        print_integer(getpid());
    }else{
        printf("\nHello from parent = ");
        print_integer(getpid());
    }

    while(1);
}
```

```
void hello_from_execve(){
    printf("\nHello from execve = ");
    print_integer(getpid());
    while(1);
}
```

```
child-> execve(hello_from_execve) test:

Hello from parent = 0
Hello from execve = 1
```

Figure 16: execve sample run

3.2.2 Custom System Calls

In order to implement the strategies, I had to implement some extra system calls. This section explains them.

```
case SYS_SET_PRIORITY:
    taskManager->setCurrenttaskPriority(cpu->ebx);
    break;

case SYS_GET_PRIORITY:
    cpu->edx = taskManager->getPriority();
    break;

case SYS_INTRPT_COUNT:
    cpu->edx = taskManager->getIntrptCount();
    break;

case SYS_ADD_AGING:
    taskManager->setAgingPid(cpu->ebx);
    break;

case SYS_SET_STATE:
    if(cpu->ebx == 0)
    taskManager->setCurrenttaskState(Ready);
    if(cpu->ebx == 2)
    taskManager->setCurrenttaskState(Blocked);
    break;

case SYS_PRINT_TABLE:
    taskManager->printProcessTable();
    break;
```

Figure 17: Custom System Calls

void sys_process_table( )

System call whose job is to print the process table.

void set_priority(int priority)

System call that changes the current processing priority used in priority based strategies.

int sys_get_priority( )

System call used to see priority changes in priority based strategies.

int sys_interrupt_count( )

System call used to change the priority after a certain time in priority based strategies.

void set_aging(int pid)

System call used to give process aging that undergoes starvation in priority based strategies.

void set_state(int state)

System call used to update the process state after the input used in Part C.

3.3 Interrupt Handling

3.3.1 Timer Interrupts

```
if(interrupt == hardwareInterruptOffset)
{
    esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
}
```

Figure 18: Timer Interrupt

Our operating system can handle timer interrupt. Whenever there is a time interrupt, Schedule is activated and multiprogramming is provided. Timer interrupt uses the 0x20 interrupt vector.

3.3.2 Keyboard Interrupts

```
class PrintfKeyboardEventHandler : public KeyboardEventHandler
{
public:
    void OnKeyDown(char c)
    {
        char* foo = " ";
        foo[0] = c;
        printf(foo);
    }
};
```

Figure 19: KeyboardEventHandler

Our system can handle keyboard interrupts by implementing override methods of KeyboardEventHandler.

To use in Part C, I derived a new IntegerScanfKeyboardEventHandler to implement the scanf function:

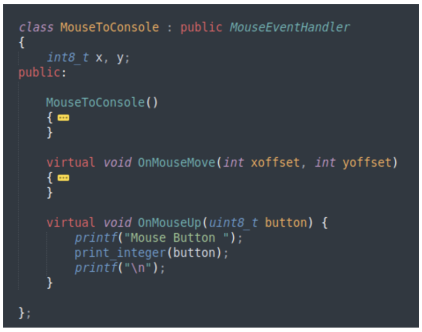
```
class IntegerScanfKeyboardEventHandler : public KeyboardEventHandler {
    int value;
    bool completed;
    bool negative;

public:
    IntegerScanfKeyboardEventHandler() : value(0), completed(false), negative(false) {}

    virtual void OnKeyDown(char c) {
        if (completed) {
            return; // if completed do not enter new char
        }

        if (c == '\n') {
            if (negative) {
                value = -value;
            }
            completed = true; // complete input with enter button
        } else if (c == '-' && value == 0 && !negative) {
            negative = true; // negative number
        } else if (c >= '0' && c <= '9') {
            value = value * 10 + (c - '0'); // convert char to number
        }

        // print char to screen for user interaction
        printf(&c);
    }

    bool IsCompleted() const {
    }

    void Reset() {
    }

    int GetValue() const {
    }
};
```

```
void scanf(int* outValue) {

    intScanfHandler.Reset();

    while (!intScanfHandler.IsCompleted());

    *outValue = intScanfHandler.GetValue();

    set_state(0); // Set state to READY
}
```

Figure 20: IntegerScanfKeyboardEventHandler

Using this EventHandler, processes can receive integer input from the user via the keyboard and use it in tasks. This scanf function is used in Part C.

3.3.3 Mouse Interrupts



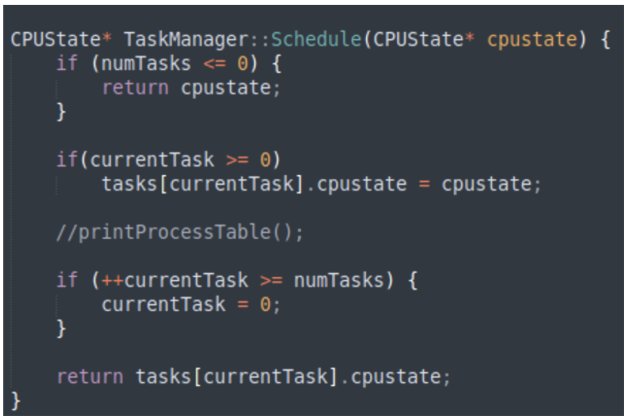
```
class MouseToConsole : public MouseEventHandler
{
    int8_t x, y;
public:

    MouseToConsole()
    { ...
    }

    virtual void OnMouseMove(int xoffset, int yoffset)
    { ...
    }

    virtual void OnMouseUp(uint8_t button) {
        printf("Mouse Button ");
        print_integer(button);
        printf("\n");
    }

};
```

Figure 21: MouseEventHandler

Our operating system can already handle mouse interrupts such as mouse movement, similar to keyboard interrupts. Additionally, I override the OnMouseUp method so that the mouse buttons appear on the screen.

3.4 Scheduler and Scheduling Strategies

3.4.1 First Version: Simple Round Robin Scheduler

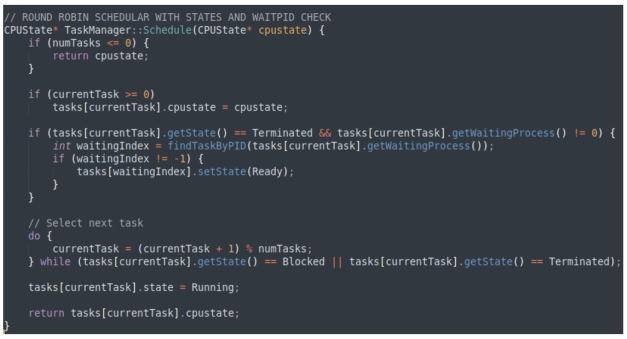


```
CPUState* TaskManager::Schedule(CPUState* cpustate) {
    if (numTasks <= 0) {
        return cpustate;
    }

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

    //printProcessTable();

    if (++currentTask >= numTasks) {
        currentTask = 0;
    }

    return tasks[currentTask].cpustate;
}
```

Figure 22: Simple Round Robin Scheduler

Our operating system already applies round robin scheduling in its default version. I developed schedulers for each part using this scheduler as a basis.

3.4.2 Second Version: Round Robin Scheduler with States and Waitpid Control



```
// ROUND ROBIN SCHEDULAR WITH STATES AND WAITPID CHECK
CPUState* TaskManager::Schedule(CPUState* cpustate) {
    if (numTasks <= 0) {
        return cpustate;
    }

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

    if (tasks[currentTask].getState() == Terminated && tasks[currentTask].getWaitingProcess() != 0) {
        int waitingIndex = findTaskByPID(tasks[currentTask].getWaitingProcess());
        if (waitingIndex != -1) {
            tasks[waitingIndex].setState(Ready);
        }
    }

    // Select next task
    do {
        currentTask = (currentTask + 1) % numTasks;
    } while (tasks[currentTask].getState() == Blocked || tasks[currentTask].getState() == Terminated);

    tasks[currentTask].state = Running;

    return tasks[currentTask].cpustate;
}
```

Figure 23: Round Robin Scheduler with States and Waitpid Control

The second version I developed for Part A strategy improves the scheduler base version by providing state control and waitpid features.

3.4.3 Third Version: Priority Based Round Robin Scheduler with with States and Waitpid Control

```
// PRIORITY BASED ROUND ROBIN SCHEDULAR WITH STATES AND WAITPID CHECK
CPUState* TaskManager::Schedule(CPUState* cpustate) {
    if (numTasks <= 0) {
        return cpustate;
    }

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

    // waitpid mechanism
    if (tasks[currentTask].getState() == Terminated && tasks[currentTask].getWaitingProcess() != -1) {
        int waitingIndex = findTaskByPID(tasks[currentTask].getWaitingProcess());
        if (waitingIndex != -1) {
            tasks[waitingIndex].setState(Ready);
        }
    }

    int highestPriority = 255; // max priority
    int nextTaskIndex = currentTask;

    // -------------------------- aging ----------------- // for part b dynamic strategy
    int index = findTaskByPID(aging_pid);

    if(interrupt_count == 20 && tasks[index].getState() != Terminated && index != -1){
        tasks[index].priority = 1;
        printf("---------------- AGING ------------ \n");
    }

    //-------------------------- aging --------------------

    int startIndex = (currentTask + 1) % numTasks;

    // find highest priority
    for (int i = 0; i < numTasks; i++) {
        int idx = (startIndex + i) % numTasks; // for round robin

        if ((tasks[idx].getState() != Blocked && tasks[idx].getState() != Terminated) &&
            tasks[idx].priority < highestPriority) {

            highestPriority = tasks[idx].priority;
            nextTaskIndex = idx;

            // Reset loop to find the next tasks if multiple with same priority
            i = -1;
            startIndex = (nextTaskIndex + 1) % numTasks;
        }
    }

    currentTask = nextTaskIndex;
    tasks[currentTask].state = Running;

    this->interrupt_count++; // for part b third strategy

    return tasks[currentTask].cpustate;
}
```

Figure 24: Priority Based Round Robin Scheduler with with States and Waitpid Control

With this advanced third version scheduler, in addition to the second version features, priority control is performed and high priority processes run first. If processes have equal priority, the scheduler exhibits round robin behavior. This ensures that multiprogramming is not interrupted.

Additionally, this version also supports the aging feature. The priority of the process that is overwhelmed by other processes for a certain period of time can be set as the highest. Thus, the starvation problem is solved.

4 Strategies and Implementations

```c
void collatztask(){

    int n = 10;
    for(int i = 0; i < n; i++){
         printf("collatz_task(100) running for ");
         print_integer(i+1);
         printf("/10 times");
         printf(" PID = ");print_integer(getpid());
         printf(" Priority = ");print_integer(sys_get_priority());
         printf("\n");
         calculate_Collatz(100);
         sleep(1);
    }

    sysexit();
}

void longrunningtask(){

    int n = 10;
    for(int i = 0; i < n; i++){
        printf("long_running_task(1000) running for ");
        print_integer(i+1);
        printf("/10 times");
        printf(" PID = ");print_integer(getpid());
        printf(" Priority = ");print_integer(sys_get_priority());
        printf("\n");
        long long result = long_running_program(1000);
        sleep(1);
    }

    sysexit();
}
```

```c
void binar_search_task(){
    int arr[] = {10, 20, 30, 50, 60, 80, 100, 110, 130, 170};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 110;

    int m = 10;
    for(int i = 0; i < m; i++){
         printf("binar_search_task(arr) running for ");
         print_integer(i+1);printf("/10 times");
         printf(" PID = ");print_integer(getpid());
         printf(" Priority = ");print_integer(sys_get_priority());
         printf("\n");
         binarySearch(arr, n, x);
         sleep(1);
    }

    sysexit();
}

void linear_search_task(){
    int arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 175;

    int m = 10;
    for(int i = 0; i < m; i++){
         printf("linear_search_task(arr) running for ");
         print_integer(i+1);printf("/10 times");
         printf(" PID = ");print_integer(getpid());
         printf(" Priority = ");print_integer(sys_get_priority());
         printf("\n");
         linearSearch(arr, n, x);
         sleep(1);
    }

    sysexit();
}
```

Figure 25: Task Functions

In order to see the context switches properly in the strategies, each task function performs the calculations 10 times in the loop and does not print the calculation results. They only print iteration numbers, pid and priority values.

```
collatz_task(100) running for 1/10 times PID = 2 Priority = -1
collatz_task(100) running for 1/10 times PID = 3 Priority = -1
long_running_task(1000) running for 1/10 times PID = 4 Priority = -1
long_running_task(1000) running for 1/10 times PID = 5 Priority = -1
long_running_task(1000) running for 1/10 times PID = 6 Priority = -1
```

Figure 26

4.1 Part A: Round Robin and Process Management

4.1.1 Part A Strategy

In this strategy, collatz and long_running_program tasks are loaded into memory three times and the program runs until each task is terminated.

Each child process is created with a fork system call and the relevant program is assigned to the process with an execve system call. The parent process waits for all processes to terminate.

Each time a timer interrupt occurs, a transition is made between processes with Round Robin scheduling.

Implementation and Running Result

```c
void partA_strategy() {
    printf("PART A - Strategy - Each program loading 3 times..\n\n");

    for (int i = 0; i < 3; i++) {
        sysfork(); // fork
        int pid = sysfork_return();
        if(pid == 0){
            sysexecve(collatztask); // collatztask
            while(1);
        }
    }

    for (int i = 0; i < 3; i++) {
        sysfork(); // fork
        int pid = sysfork_return();
        if(pid == 0){
            sysexecve(longrunningtask); // longrunningtask
            while(1);
        }
    }

    // Main process waits
    while(1){
       printf("\nMain task waiting for all processes to terminate...\n");
       sys_process_table(); // Process Table
       sleep(1);
    }
}
```

```
PART A - Strategy - Each program loading 3 times..


Main task waiting for all processes to terminate...

Pid     PPid     State     Priority
--------------------------------
0       -1        Running        -1
1       0        Ready        -1
2       0        Ready        -1
3       0        Ready        -1
4       0        Ready        -1
5       0        Ready        -1
6       0        Ready        -1
--------------------------------
collatz_task(100) running for 1/10 times PID = 2 Priority = -1
collatz_task(100) running for 1/10 times PID = 3 Priority = -1
long_running_task(1000) running for 1/10 times PID = 4 Priority = -1
long_running_task(1000) running for 1/10 times PID = 5 Priority = -1
long_running_task(1000) running for 1/10 times PID = 6 Priority = -1
collatz_task(100) running for 1/10 times PID = 1 Priority =
```

Figure 27

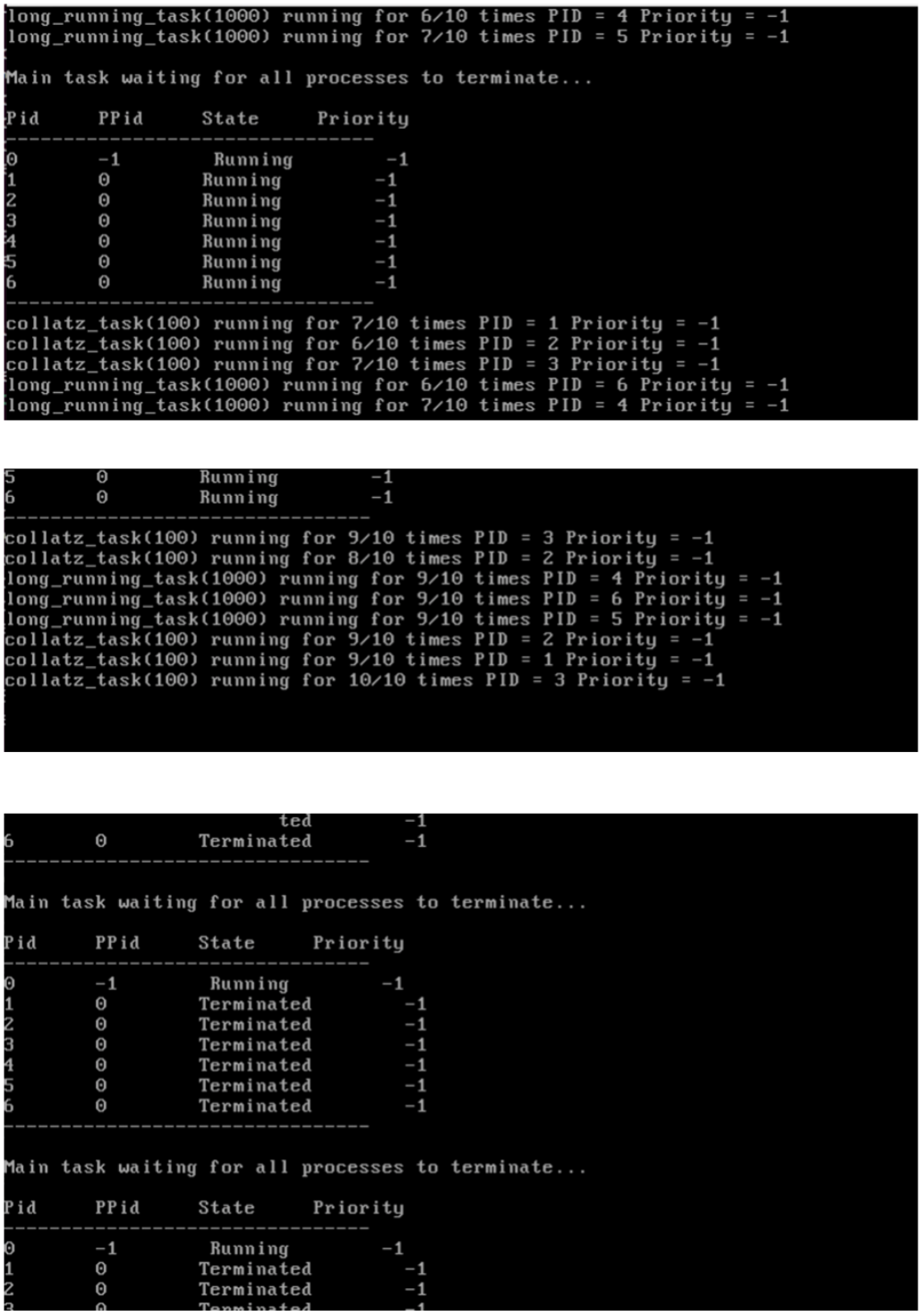

long_running_task(1000) running for 6/10 times PID = 4 Priority = -1
long_running_task(1000) running for 7/10 times PID = 5 Priority = -1
Main task waiting for all processes to terminate...
Pid PPid State Priority
0 -1 Running -1
1 0 Running -1
2 0 Running -1
3 0 Running -1
4 0 Running -1
5 0 Running -1
6 0 Running -1
collatz_task(100) running for 7/10 times PID = 1 Priority = -1
collatz_task(100) running for 6/10 times PID = 2 Priority = -1
collatz_task(100) running for 7/10 times PID = 3 Priority = -1
long_running_task(1000) running for 6/10 times PID = 6 Priority = -1
long_running_task(1000) running for 7/10 times PID = 4 Priority = -1
5 0 Running -1
6 0 Running -1
collatz_task(100) running for 9/10 times PID = 3 Priority = -1
collatz_task(100) running for 8/10 times PID = 2 Priority = -1
long_running_task(1000) running for 9/10 times PID = 4 Priority = -1
long_running_task(1000) running for 9/10 times PID = 6 Priority = -1
long_running_task(1000) running for 9/10 times PID = 5 Priority = -1
collatz_task(100) running for 9/10 times PID = 2 Priority = -1
collatz_task(100) running for 9/10 times PID = 1 Priority = -1
collatz_task(100) running for 10/10 times PID = 3 Priority = -1
ted -1
6 0 Terminated -1
Main task waiting for all processes to terminate...
Pid PPid State Priority
0 -1 Running -1
1 0 Terminated -1
2 0 Terminated -1
3 0 Terminated -1
4 0 Terminated -1
5 0 Terminated -1
6 0 Terminated -1
Main task waiting for all processes to terminate...
Pid PPid State Priority
0 -1 Running -1
1 0 Terminated -1
2 0 Terminated -1

Figure 28

4.2 Part B: Priority-Based Scheduling

4.2.1 First Strategy

In this strategy, one of the programs is selected and the same program is loaded into memory 10 times. The process continues until each child process terminates.

Each process is created with a fork also.

Implementation and Running Result

```
void part_b_first_strategy(){
    printf("PART B - First Strategy - program loading 10 times..\n\n");

    for (int i = 0; i < 10; i++) {
        sysfork(); // fork
        int pid = sysfork_return();
        if(pid == 0){
            sysexecve(collatztask); // collatztask
            while(1);
        }

    }

    // Main process waits
    while(1){
       printf("\nMain task waiting for all processes to terminate...\n");
       sys_process_table(); // Process Table
       sleep(1);
    }

}
```

```
-----------------------------------------------------
PART B - First Strategy - program loading 10 times..


Main task waiting for all processes to terminate...

Pid     PPid     State     Priority
--------------------------------
0       -1        Running        -1
1       0        Ready        -1
2       0        Ready        -1
3       0        Ready        -1
4       0        Ready        -1
5       0        Ready        -1
6       0        Ready        -1
7       0        Ready        -1
8       0        Ready        -1
9       0        Ready        -1
10       0        Ready        -1
--------------------------------
collatz_task(100) running for 1/10 times PID = 1 Priority = -1
collatz_task(100) running for 1/10 times PID = 2 Priority = -1
collatz_task(100) running for 1/10 times PID = 3 Priority = -1
```

Figure 29

4.2.2 Second Strategy

In this strategy, similar to the Part A strategy, the two selected programs are added to memory three times.

This time, binary search and linear search tasks were selected. Each process is created by fork.

Implementation and Running Result

```
void part_b_second_strategy() {
    printf("PART B - Second Strategy - Each program loading 3 times..\n\n");

    for (int i = 0; i < 3; i++) {
        sysfork(); // fork
        int pid = sysfork_return();
        if(pid == 0){
            sysexecve(binar_search_task); // collatztask
            while(1);
        }
    }

    for (int i = 0; i < 3; i++) {
        sysfork(); // fork
        int pid = sysfork_return();
        if(pid == 0){
            sysexecve(linear_search_task); // longrunningtask
            while(1);
        }
    }

    // Main process waits
    while(1){
       printf("\nMain task waiting for all processes to terminate...\n");
       sys_process_table(); // Process Table
       sleep(1);
    }

}
```

```
PART B - Second Strategy - Each program loading 3 times..

Main task waiting for all processes to terminate...

Pid     PPid     State     Priority
--------------------------------
0       -1        Running        -1
1       0        Ready        -1
2       0        Ready        -1
3       0        Ready        -1
4       0        Ready        -1
5       0        Ready        -1
6       0        Ready        -1
--------------------------------
binar_search_task(arr) running for 1/10 times PID = 1 Priority = -1
binar_search_task(arr) running for 1/10 times PID = 2 Priority = -1
binar_search_task(arr) running for 1/10 times PID = 3 Priority = -1
linear_search_task(arr) running for 1/10 times PID = 4 Priority = -1
linear_search_task(arr) running for 1/10 times PID = 5 Priority = -1
linear_search_task(arr) running for 1/10 times PID = 6 Priority = -1
```

Figure 30

4.2.3 Third Strategy

In this strategy, the collatz program is loaded with low priority. After a certain number of interrupts (I set it to 25), other programs are loaded with the same priority. It is expected that programs with the same priority will work in round robin.

Implementation and Running Result

```
void part_b_third_strategy() {
    printf("PART B - Third Strategy - Fixed Priority starting with col

    set_priority(5);
    int pid;


    // low priority Collatz task
    sysfork();
    pid = sysfork_return(); // double return
    if (pid == 0) {
        set_priority(5);
        sysexecve(collatztask);
        while(1);
    }

    int interrupt_count = 0;

    // after 50th interrupt other programs loading
    while(true){
        interrupt_count = sys_interrupt_count();
        if(interrupt_count > 50){
            break;
        }
    }

    //BinarySearch
    sysfork();
    pid = sysfork_return(); // double return
    if (pid == 0) {
        set_priority(5);
        sysexecve(binar_search_task);
        while(1);
    }

    //LinearSearch
    sysfork();
```

```
------------------------------------------------------
PART B - Third Strategy - Fixed Priority starting with collatztask..

collatz_task(100) running for 1/10 times PID = 1 Priority = 5
collatz_task(100) running for 2/10 times PID = 1 Priority = 5
collatz_task(100) running for 3/10 times PID = 1 Priority = 5
collatz_task(100) running for 4/10 times PID = 1 Priority = 5
collatz_task(100) running for 5/10 times PID = 1 Priority = 5
collatz_task(100) running for 6/10 times PID = 1 Priority = 5
Main task waiting for all processes to terminate...

Pid     PPid     State     Priority
-------------------------------
0       -1        Running        10
1       0        Running        5
2       0        Ready        -1
3       0        Ready        -1
4       0        Ready        -1
-------------------------------
binar_search_task(arr) running for 1/10 times PID = 2 Priority = 5
linear_search_task(arr) running for 1/10 times PID = 3 Priority = 5
long_running_task(1000) running for 1/10 times PID = 4 Priority = 5
collatz_task(100) running for 7/10 times PID = 1 Priority = 5
```

Figure 31

4.2.4 Dynamic Priority Strategy

In this strategy, the collatz task is started with a lower priority than other processes. However, the aging feature is activated for the collatz task. The scheduler sets the priority of the collatz task to the highest after a certain interrupt count. So, after a while, the collatz task takes priority and runs until it terminates itself.

Implementation and Running Result

```
// PRIORITY BASED ROUND ROBIN
void part_b_dynamic_strategy() {
    printf("PART B - Third Strategy - Dynamic Priority starting..\n\n");

    int numProcesses = 4; // Toplam süreç sayısı
    int initializedProcesses = 0;

    set_priority(5); // Düşük öncelik
    int pid;

    // low priority Collatz task
    sysfork();
    pid = sysfork_return(); // double return
    if (pid == 0) {
        set_aging(getpid()); // set aging for collatz task
        set_priority(5);
        while (initializedProcesses < numProcesses);
        sysexecve(collatztask);
        while(1);
    }
    else{
        initializedProcesses++;
    }

    //BinarySearch
    sysfork();
    pid = sysfork_return(); // double return
    if (pid == 0) {
        set_priority(4);
        while (initializedProcesses < numProcesses);
        sysexecve(binar_search_task);
        while(1);
    }
```

```
PART B - Third Strategy - Dynamic Priority starting..

Main task waiting for all processes to terminate...

Pid     PPid     State     Priority
--------------------------------
0       -1        Running        5
1       0        Ready        -1
2       0        Ready        -1
3       0        Ready        -1
4       0        Ready        -1
--------------------------------
binar_search_task(arr) running for 1/10 times PID = 2 Priority = 4
linear_search_task(arr) running for 1/10 times PID = 3 Priority = 2
long_running_task(1000) running for 1/10 times PID = 4 Priority = 3
collatz_task(100) running for 1/10 times PID = 1 Priority = 5
linear_search_task(arr) running for 2/10 times PID = 3 Priority = 2
linear_search_task(arr) running for 3/10 times PID = 3 Priority = 2
linear_search_task(arr) running for 4/10 times PID = 3 Priority = 2
---------------- AGING -------------
collatz_task(100) running for 2/10 times PID = 1 Priority = 1
collatz_task(100) running for 3/10 times PID = 1 Priority = 1
```

Figure 32

4.3 Part C

4.3.1 Interactive Input Handling Strategy

In this strategy, the version of the collatz task that takes input with scanf was chosen. The collatz task program is loaded into memory 3 times. Each process waits for input in an infinitive loop. The states of processes that receive input are set to ready. The strategy continues to work continuously across different processes.

Implementation and Running Result

```
void part_c_first_strategy() {

    printf("PART C - First Strategy\n\n");

    for (int i = 0; i < 3; i++) {
        sysfork(); // fork
        int pid = sysfork_return();
        if(pid == 0){
         sysexecve(collatztask_with_input); // collatztask
        }

    }

    while(1);


}
```

```
void scanf(int* outValue) {

    intScanfHandler.Reset();

    while (!intScanfHandler.IsCompleted());

    *outValue = intScanfHandler.GetValue();

    set_state(0); // Set state to READY
}
```

```
void collatztask_with_input(){
    while(1){
    printf("\nPID: ");print_integer(getpid());
    printf(" Please enter an integer: ");
    int number;
    scanf(&number);
    //printf("\nYou entered: ");print_integer(number);
    printf("PID: ");print_integer(getpid());printf(" ");printCollatz(number);
    sys_process_table();
    }

}
```

Figure 33

4.3.2 Interactive Input Priority Strategy

Actually I don't understand exactly how this strategy can be done. We must set priority between keyboard and mouse interrupts. However, since it is not possible to press the keyboard and mouse at the same time, it will not be a testable part. If we want to compare the priorities of processes waiting for keyboard and mouse input, we already do this in Part B.