GTU Department of Computer Engineering
CSE 312 / 504 - Spring 2023
Homework 1 Report

Emirkan Burak Yılmaz
1901042659

1 Table of Contents

2 Processes and Process Table

The provided source code contains class named Task to represent processes. Task class has fields for basic process information such as cpustate (record of the registers) and stack. I add additional field like pid, status and priority to have more detailed information about processes.

```
class Task
{
friend class TaskManager;
private:
    static const int STACK_SIZE = 4096; // 4 KiB
    common::uint8_t stack[STACK_SIZE];
    TaskState state;
    common::uint32_t pid;
    common::uint32_t ppid;
    TaskPriority priority;
    CPUState *cpustate;
    bool waitparent;
public:
    Task(GlobalDescriptorTable *gdt, void entrypoint(), common::uint32_t ppid, TaskPriority priority);
    Task();
    ~Task();
    void InitTask(GlobalDescriptorTable *gdt, void entrypoint(), common::uint32_t pid,
        common::uint32_t ppid, TaskPriority priority);
};
```

The TaskManager class keeps all the task in an array to represent the process table. I would normally prefer using circular linked list for process table, but since it was already implemented with array in the source code, I just be okey with that. I add additional functions in TaskManager class for system calls like fork, execve, waitpid and helper functions for printing the process table.

```
class TaskManager
{
private:
    Task tasks[256];
    int numTasks;
    int numActiveTasks;
    int currentTask;
    int nextPid;
    GlobalDescriptorTable *gdt;
    void PrintCPUState(Task *task);
    void PrintCPUState(CPUState *cpustate);
    int FindTask(common::uint32_t pid);
    void CopyTask(Task *src, Task *dest);
    void PrintTask(int tableIndex);
    void PrintTaskTable();
    void DispatchTask();
public:
    TaskManager(GlobalDescriptorTable *gdt);
    ~TaskManager();
    bool AddTask(Task *task);
    common::uint32_t Fork(CPUState *cpustate);
    common::uint32_t Execve(void (*entrypoint)());
    void Waitpid(common::uint32_t pid);
    void Exit();
    CPUState* Schedule(CPUState *cpustate);
};
```

3 Loading Multiple Program into Memory

To load the programs into memory AddTask function is used. It takes pointer to a task and gives it a next PID and adds into the process table. With this functionality multiple programs can be loaded into memory by the TaskManager.

```
bool TaskManager::AddTask(Task* task)
{
    if(numTasks >= 256)
        return false;
    task->pid = nextPid;
    CopyTask(task, &tasks[numTasks]);
    ++nextPid, ++numTasks, ++numActiveTasks;
    return true;
}
```

4 Multiprogramming

Multiprogramming and context switching is done by the TaskManager class. It has process table to keep multiple processes and scheduler functions switching between processes. In addition to that it contains implementation for the system call functions like fork, execve, and waitpid.

```
class TaskManager
{
private:
    Task tasks[256];
    int numTasks;
    int numActiveTasks;
    int currentTask;
    int nextPid;
    GlobalDescriptorTable *gdt;
    void PrintCPUState(Task *task);
    void PrintCPUState(CPUState *cpustate);
    int FindTask(common::uint32_t pid);
    void CopyTask(Task *src, Task *dest);
    void PrintTask(int tableIndex);
    void PrintTaskTable();
    void DispatchTask();
public:
    TaskManager(GlobalDescriptorTable *gdt);
    ~TaskManager();
    bool AddTask(Task *task);
    common::uint32_t Fork(CPUState *cpustate);
    common::uint32_t Execve(void (*entrypoint)());
    void Waitpid(common::uint32_t pid);
    void Exit();
    CPUState* Schedule(CPUState *cpustate);
};
```

4.1 Context Switching and Round Robin Scheduling

The design of process table has already been described in the Processes and Process Tables part. So, it's basically an array of Tasks. To keep track of the currently running task, the integer field currentTask is used. It is just an index inside the process table. When the timer interrupt occurs handler function calls the scheduler for dispatching a new task. Scheduler finds the next task by gradually increasing currentTask and checking if the task is in ready state. When it finds the first

ready state task it immediately put in running state and returns its esp(stack pointer). If there is no runnable process except the previously running task, then it dispatched again.

```
void TaskManager::DispatchTask()
{
    // if the task is not blocked, set its state to ready
    if (tasks[currentTask].state == Running)
        tasks[currentTask].state = Ready;

    do {
        // apply Round Robin scheduling
        if (++currentTask >= numTasks)
            currentTask %= numTasks;
    } while (tasks[currentTask].state != Ready);

    tasks[currentTask].state = Running;
}

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

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

    DispatchTask();

    if (numActiveTasks > 1)
        PrintTaskTable();

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

5 Interrupt Handling

For handing interrupts, we have InterruptManager class which contains IDT (interrupt descriptor table). The IDT maps the interrupt number and the handler function so that handler function can be selected and executed to handle interrupt. In addition to InterruptManager class, we need to turn back to assembly to set and manipulate the CPU registers and stack to call interrupt handler.

```
uint32_t InterruptManager::DoHandleInterrupt(uint8_t interrupt, uint32_t esp)
{

    if(handlers[interrupt] != 0)
    {
        if (interrupt == 0x06) {
            printf("Invalid Opcode Error\n");
        }

        esp = handlers[interrupt]->HandleInterrupt(esp);
    }
    else if(interrupt != hardwareInterruptOffset)
    {
        if (interrupt == 0x0D) {
           printf("General Protection Fault\n");
        }
        else {
            printf("UNHANDLED INTERRUPT 0x");
            printfHex(interrupt);
            printf("\n");
        }
    }

    if(interrupt == hardwareInterruptOffset)
    {
        // if there is a I/O operation do not make a context switch
        if (IOLock == false && ++NumTimerInterrupt == NUM_TIMER_IGN) {
            esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
            NumTimerInterrupt = 0;
        }
    }

    // hardware interrupts must be acknowledged
    if(hardwareInterruptOffset <= interrupt && interrupt < hardwareInterruptOffset+16)
    {
        programmableInterruptControllerMasterCommandPort.Write(0x20);
        if(hardwareInterruptOffset + 8 <= interrupt)
            programmableInterruptControllerSlaveCommandPort.Write(0x20);
    }

    return esp;
}
```

5.1 Timer Interrupt (0x20)

Previously each timer interrupt was causing context switching. However, in this specific implementation, timer interrupts is ignored **at least 15 times** (NUM_TIMER_IGN), so that the other interrupts such as keyboard and mouse can be handled. But this is not enough because if there is I/O operation is done which can be investigated with the static variable IOLock than timer interrupt is being ignored until the lock is realised. When the two conditions are satisfied scheduler gets the next ready process from the process table and make it ready for the execution by chancing process states. If there is no process in ready state, the scheduler dispatches the current process and let it to continue its execution.

```
if(interrupt == hardwareInterruptOffset)
{
    // if there is a I/O operation do not make a context switch
    if (IOLock == false && ++NumTimerInterrupt == NUM_TIMER_IGN) {
        esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
        NumTimerInterrupt = 0;
    }
}
```

5.2 Keyboard Interrupts

All the user keyboard interactions are written to both screen and system buffer stdin. The stdin buffer

```
void PrintfKeyboardEventHandler::OnKeyDown(char c)
{
    // convert char c to str for printf
    char* foo = " ";
    foo[0] = c;
    printf(foo);
    InterruptManager::stdin.write(c);
}
```

System buffer is another class which contains a buffer and index variables for both write and read ends. When the keyboard interrupt occurs the user input taken to stdin buffer, and it can read with the system call scanf.

```
class SystemBuffer {
    private:
        const int BUFF_SIZE = 4096;
        char buff[4096]; // 4Kib
        int writeEnd;
        int readEnd;
    public:
        SystemBuffer();
        ~SystemBuffer();
        void write(char *str);
        void write(char c);
        char *read(char delim);
        char *read();
};
```

5.3 Mouse Interrupts

Mouse movements are kept track with the x and y coordination of the cursor. Whenever mouse cursor moves, the mouse interrupt occurs and the position on the screen is readjusted.

```
MouseToConsole::MouseToConsole()
{
    common::uint16_t* VideoMemory = (common::uint16_t*)0xb8000;
    x = 40;
    y = 12;
    VideoMemory[80*y+x] = (VideoMemory[80*y+x] & 0x0F00) << 4
                        | (VideoMemory[80*y+x] & 0xF000) >> 4
                        | (VideoMemory[80*y+x] & 0x00FF);
}

void MouseToConsole::OnMouseMove(int xoffset, int yoffset)
{
    static common::uint16_t* VideoMemory = (common::uint16_t*)0xb8000;
    VideoMemory[80*y+x] = (VideoMemory[80*y+x] & 0x0F00) << 4
                        | (VideoMemory[80*y+x] & 0xF000) >> 4
                        | (VideoMemory[80*y+x] & 0x00FF);

    x += xoffset;
    if(x >= 80) x = 79;
    if(x < 0) x = 0;
    y += yoffset;
    if(y >= 25) y = 24;
    if(y < 0) y = 0;

    VideoMemory[80*y+x] = (VideoMemory[80*y+x] & 0x0F00) << 4
                        | (VideoMemory[80*y+x] & 0xF000) >> 4
                        | (VideoMemory[80*y+x] & 0x00FF);
}
```

5.4 Invalid Opcode Error (0x06)

This happens after the process comes to the end of the program. To handle this interrupt, I implement ProcessExecutionHandler which terminates the current process by removing it to process table and call scheduler to schedule the next task.

5.5 General Protection Fault (0x0D)

This is protection mechanism in response to an access violation caused by access of invalid code segment as my understanding. I faced with this interrupt during the implementation of fork system call. I try to find the error and I suppose it occurs due to invalid state of the parent process which can be viewed with cs (code segment) register in CPUState.

6 POSIX System Calls

System calls are made with asm statement and providing arguments like entrypoint for sys_execve and string for sys_write.

```
void SyscallHandler::sys_exit() {
    asm("int $0x80" : : "a" (1));
}

void SyscallHandler::sys_execve(void (*entrypoint)()) {
    asm("int $0x80" : : "a" (11), "b" (entrypoint));
}

void SyscallHandler::sys_waitpid(int pid) {
    asm("int $0x80" : : "a" (7), "b"(pid));
}

void SyscallHandler::sys_fork() {
    asm("int $0x80" : : "a" (2));
}

void SyscallHandler::sys_write(char *str) {
    asm("int $0x80" : : "a" (4), "b" (str));
}

void SyscallHandler::sys_read(char *out) {
    asm("int $0x80" : : "a" (3), "b" (out));
}
```

When the system call is happened, syscall handler handles the interrupt by calling proper functions.

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

    switch(cpu->eax)
    {
        case 1: //  sys_exit
            taskManager->Exit();
            esp = (uint32_t) taskManager->Schedule(cpu);
            break;
        case 2: //  sys_fork
            taskManager->Fork(cpu);
            break;
        case 3: // sys_read
            scanf((char *)cpu->ebx);
            break;
        case 4: // sys_write
            printf((char*)cpu->ebx);
            break;
        case 7: // sys_waitpid
            taskManager->Waitpid(cpu->ebx);
            break;
        case 11: // sys_execve
            esp = taskManager->Execve((void (*)()) cpu->ebx);
            break;
        default:
            break;
    }

    return esp;
}
```

6.1 fork

Fork system call is implemented by creating a new task and copying the parent's stack and CPU state (registers) to child. With this way the child task become exact copy of its parent. After the creation of the child process is done, it's taken and put into process table to be executed.

```
common::uint32_t TaskManager::Fork(CPUState *cpustate) {
    printf("### syscall fork ###\n");

    if (numTasks >= 256)
        return -1;

    Task *parent = &tasks[currentTask];
    Task *child = &tasks[numTasks];

    child->InitTask(gdt, (void(*)()) parent->cpustate->eip, nextPid, parent->pid, parent->priority);

    // copy the current cpustate to the child
    *(child->cpustate) = *cpustate;

    ++nextPid, ++numTasks, ++numActiveTasks;
    return child->pid;
}
```

6.2 waitpid

Task class has type of boolean field waitparent which is basically a flag to indicate if the parent of current process waiting for the termination of the current process. During the termination of a process, waitparent flag is checked. If its value is true, then the status of the parent set to ready by reaching PPID. With this way each child process will make their parent status to ready before their termination. The system call waitpid takes an integer value as an argument to refer PID. First, status of the current process is set to blocked and the waitparent flag of the child process is set to true. The parent process is scheduled again when the child is killed with the waitparent flag.

```
void TaskManager::Waitpid(common::uint32_t pid) {
    printf("### syscall waitpid ###\n");
    Task *parent = &tasks[currentTask];

    // if the pid is -1, then waits any child to finish, otherwise waits the specific child
    if (pid > 0) {
        int childIndex = FindTask(pid);
        // make sure there is such a child process
        if (childIndex != -1 && tasks[childIndex].ppid == parent->pid) {
            // set parent status to blocked and
            parent->state = Blocked;
            // when the child is terminated, parent will wakes up
            tasks[childIndex].waitparent = true;
        }
    }
    else {
        // set true to the waitparent for all the child
        for (int i = 0; i < numTasks; ++i) {
            if (tasks[i].ppid == parent->pid) {
                parent->state = Blocked;
                tasks[i].waitparent = true;
            }
        }
    }
}
```

6.3 execve

The system call execve takes an entry point to replace the execution of the current process. Then it basically resets the cpustate of the current task and assign new entrypoint to the instruction register eip.

```
common::uint32_t TaskManager::Execve(void (*entrypoint)()) {
    printf("### syscall execve ###\n");
    Task *curr = &tasks[currentTask];
    curr->InitTask(gdt, entrypoint, curr->pid, curr->ppid, curr->priority);
    return (common::uint32_t) curr->cpustate;
}
```

6.4 exit

The current process is set to terminated and its parent is waked up if the child is waited by its parent.

```
void TaskManager::Exit() {
    printf("### syscall exit ###\n");

    Task *current = &tasks[currentTask];

    /* security guard for init process. it cannot be destroyed */
    if (current->pid == 1)
        return;

    // if it's parent waits its child to terminate
    if (current->waitparent) {
        int parentIndex = FindTask(current->ppid);
        if (parentIndex >= 0)
            tasks[parentIndex].state = Ready;
    }

    --numActiveTasks;
    current->state = TaskState::Terminated;
}
```

6.5 read (scanf)

All the user keyboard interactions are stored in system wide buffer stdin. stdin is a static varible in InterruptManager class and it is an object of SystemBuffer class. SystemBuffer class provides a buffer which has read and write ends.

```
    class SystemBuffer {
        private:
            const int BUFF_SIZE = 4096;
            char buff[4096]; // 4Kib
            int writeEnd;
            int readEnd;
        public:
            SystemBuffer();
            ~SystemBuffer();
            void write(char *str);
            void write(char c);
            char *read(char delim);
            char *read();
    };
```

Whenever user press a key, the keyboard interrupt occurred, and that value is written to stdin buffer by the keyboard handler. When there is a read system call happen all the user input are taken from the stdin buffer until the new line character. So, scanf reads whole line in a single call. One more important thing about scanf is it sets a lock on timer interrupt with the setIOLock function. With this lock all the timer interrupts are ignored, and no context switching is made during this process.

```
int scanf(char *out) {
    char *str;
    int i;
    InterruptManager::setIOLock(true);
    // read the whole line
    while ((str = InterruptManager::stdin.read('\n')) == nullptr) ; /* busy waiting */

    for (i = 0; str[i] != '\0' && str[i] != '\n'; ++i)
        out[i] = str[i];
    out[i] = '\0';
    InterruptManager::setIOLock(false);
    return i; // return number of charachters being read
}

int scanfPrompt(char *prompt, int *out) {
    int rv;
    char buff[128];

    printf(prompt);
    rv = scanf(buff);
    while (atoi(buff, out) == -1) {
        printf("Invalid format error\n");
        printf(prompt);
        rv = scanf(buff);
    }
    return rv;
}
```

In addition to scanf, there is a useful extension of it called scanfPrompt which prompts the user for an integer until the user enters a proper input. This function is mostly used in test functions to get proper parameter from the user such as the target value for binary search.

7 Lifecycles

7.1 First Strategy

In the first strategy init process initializes process table, load 3 different programs to the memory start them and will enter an infinite loop until all the processes terminate.

```
void lifecyle1(GlobalDescriptorTable *gdt, TaskManager *taskManager)
{
    Task taskLinearSearch(gdt, epLinearSearch, 1, Normal);
    Task taskBinarySearch(gdt, epBinarySearch, 1, Normal);
    Task taskCollatz(gdt, epCollatz, 2, Normal);

    taskManager->AddTask(&taskLinearSearch);
    taskManager->AddTask(&taskBinarySearch);
    taskManager->AddTask(&taskCollatz);
}
```

7.2 Second Strategy

Second strategy is randomly choosing one of the programs and loads it into memory 10 times (Same program 10 different processes), start them and will enter an infinite loop until all the processes terminate.

```
void lifecyle2(GlobalDescriptorTable *gdt, TaskManager *taskManager)
{
    const int NUM_OF_EXEC = 10;
    const int NUM_OF_PROG = 3;
    void (*prog[NUM_OF_PROG]) (void) = {epLinearSearch, epBinarySearch, epCollatz};

    void (*entrypoint)(void) = prog[rand(0, NUM_OF_PROG)];

    Task task;

    // load randomly selected program in specified number of times
    for (int i = 0; i < NUM_OF_EXEC; ++i) {
        task.InitTask(gdt, entrypoint, 1, 1, Normal);
        taskManager->AddTask(&task);
    }
}
```

7.3 Final Strategy

Final Strategy is choosing 2 out 3 programs randomly and loading each program 3 times start them and will enter an infinite loop until all the processes terminate.

```
void lifecyle3(GlobalDescriptorTable *gdt, TaskManager *taskManager)
{
    const int NUM_OF_EXEC = 3;
    const int NUM_OF_PROG = 3;
    void (*prog[NUM_OF_PROG]) (void) = {epLinearSearch, epBinarySearch, epCollatz};

    int r1 = rand(0, NUM_OF_PROG), r2 = rand(0, NUM_OF_PROG);
    while (r1 == r2)
        r2 = rand(0, NUM_OF_PROG);

    Task task;

    // load randomly selected 2 program in specified number of times
    for (int i = 0; i < NUM_OF_EXEC; ++i) {
        task.InitTask(gdt, prog[r1], 1, 1, Normal);
        taskManager->AddTask(&task);
        task.InitTask(gdt, prog[r2], 1, 1, Normal);
        taskManager->AddTask(&task);
    }
}
```

8 Test Cases

```
#ifndef __TEST_H
#define __TEST_H

#include <gdt.h>
#include <multitasking.h>

namespace myos
{
    /* ep denotes entrypoint */

    void lifecyle1(GlobalDescriptorTable *gdt, TaskManager *taskManager);
    void lifecyle2(GlobalDescriptorTable *gdt, TaskManager *taskManager);
    void lifecyle3(GlobalDescriptorTable *gdt, TaskManager *taskManager);

    int binarySearch(int *arr, int target, int lo, int hi);
    void epBinarySearch();

    int linearSearch(int * arr, int size, int target);
    void epLinearSearch();

    void printSeq(int *seq);
    void collatzSeq(int n, int *buff);
    void epCollatz();

    void testWaitpid(GlobalDescriptorTable *gdt, TaskManager *taskManager);

    void testFork(GlobalDescriptorTable *gdt, TaskManager *taskManager);
    void epTestFork();

    void testExecve(GlobalDescriptorTable *gdt, TaskManager *taskManager);
    void epTestExecve();

    void testForkAndExecve(GlobalDescriptorTable *gdt, TaskManager *taskManager);
    void epTestForkAndExecve();

    void testScanf(GlobalDescriptorTable *gdt, TaskManager *taskManager);
    void epTestScanf();
}
#endif
```

```
void initProcess(GlobalDescriptorTable *gdt, TaskManager *taskManager)
{
    printf("Welcome to MyOS\n");

    Task taskInit(gdt, entrypointInit, 0, High);
    taskManager->AddTask(&taskInit);

    // lifecyle1(gdt, taskManager);

    // lifecyle2(gdt, taskManager);

    // lifecyle3(gdt, taskManager);

    // lifecyleTest(gdt, taskManager);

    // testFork(gdt, taskManager);

    // testExecve(gdt, taskManager);

    // testForkAndExecve(gdt, taskManager);

    // testScanf(gdt, taskManager);

    // testWaitpid(gdt, taskManager);
}
```

The initProcess (in src/kernel.cpp) loads the testTask which are commented in the above code. You can see the result of the test case by uncomment the part of the code and compile and run or you can check screen shot result that are given in the next part. You can comment printProcessTable function in TaskManager::Schedule function to see less output on the screen.

9 Running Results

9.1 Process Table and Sample System Call

```
*************************************************************************************************
PID    PPID    Priority    State
1      0       High        Running
2      1       Normal      Ready
3      1       Normal      Ready
4      1       Normal      Ready
5      1       Normal      Ready
6      1       Normal      Ready
7      1       Normal      Ready
8      1       Normal      Ready
9      1       Normal      Ready
10      1       Normal      Ready
11      1       Normal      Ready
12      1       High        Ready
*************************************************************************************************
### syscall execve ###
```

9.2 fork

```
void epTestFork()
{
    for (int i = 0; i < 50; ++i) {
        printDigit(i);
        printf(" ");
    }
    printf("\n");
    SyscallHandler::sys_fork();
    printf("Helloo from fork\n");
    while (true) ; // infinitive loop
}
```

```
Hello World! --- http://www.AlgorithMan.de
0 1 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
### syscall fork ###
Helloo from fork
Helloo from fork
```

After system call fork, printf statement executed with two different processes.

9.3 execve

```
void epTestExecve()
{
    for (int i = 0; i < 50; ++i) {
        printDigit(i);
        printf(" ");
    }
    printf("\n");
    SyscallHandler::sys_execve(epCollatz);
    printf("This will not be executed\n");
    while (true) ; // infinitive loop
}
```

```
Hello World! --- http://www.AlgorithMan.de
0 1 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
### syscall execve ###
25: 76 38 19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
24: 12 6 3 10 5 16 8 4 2 1
23: 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
22: 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
21: 64 32 16 8 4 2 1
20: 10 5 16 8 4 2 1
19: 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
18: 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
17: 52 26 13 40 20 10 5 16 8 4 2 1
16: 8 4 2 1
15: 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
14: 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
13: 40 20 10 5 16 8 4 2 1
12: 6 3 10 5 16 8 4 2 1
11: 34 17 52 26 13 40 20 10 5 16 8 4 2 1
10: 5 16 8 4 2 1
9: 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

```
8: 4 2 1
7: 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
6: 3 10 5 16 8 4 2 1
5: 16 8 4 2 1
```

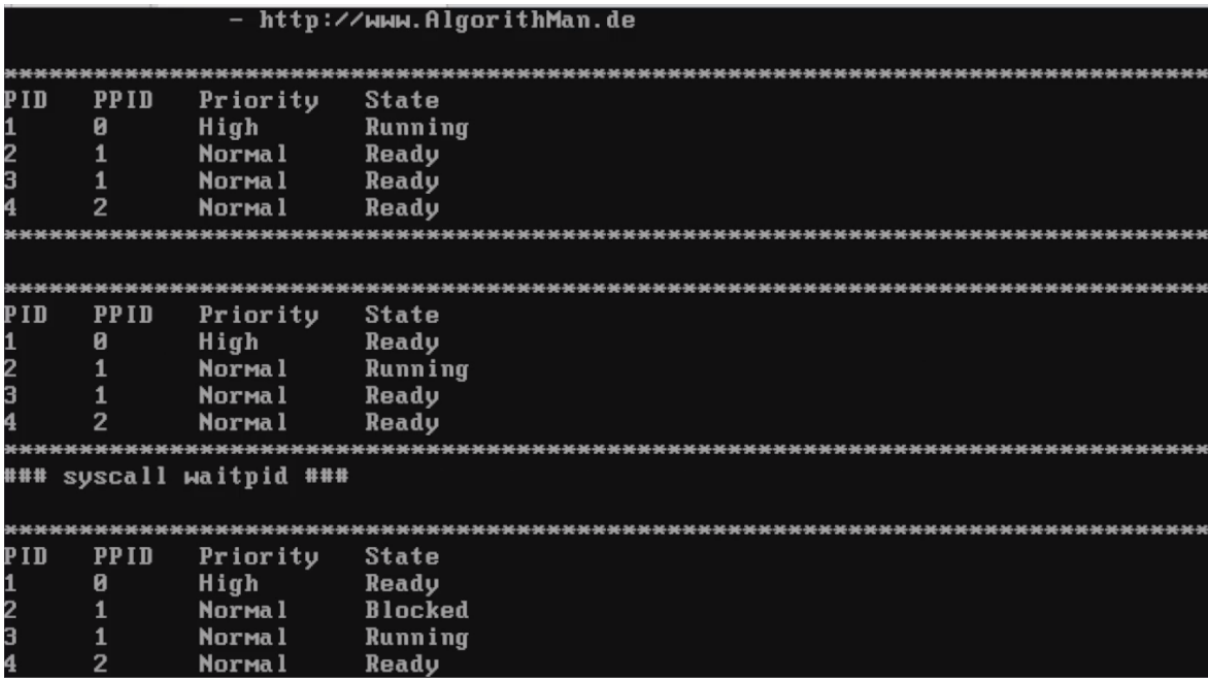
```
4: 2 1
3: 10 5 16 8 4 2 1
2: 1
1: 1
```

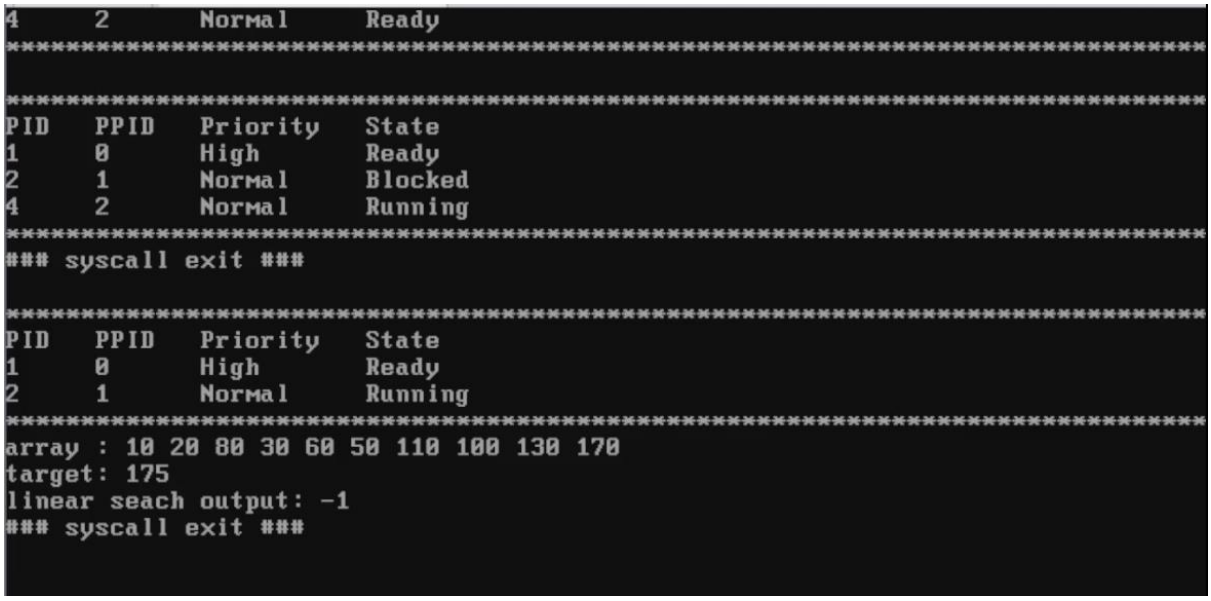
After the execve system call, the process memory image is changed with collatz sequence.

9.4 waitpid

```
void testWaitpid(GlobalDescriptorTable * gdt, TaskManager * taskManager)
{
    // uncomment waitpid sentence from the entrypointLinearSearch function
    lifecyle1(gdt, taskManager);
}
```

Basically process 2 waits process 4. As you can see process 2 is parent of process 4.





After the process 4 is terminated with the exit syscall, process 2 which is parent of process 2 is waked up.

9.5 scanf

```
void epTestScanf()
{
    int age;
    scanfPrompt("How old are you? ", &age);
    printf("Your age is "); printDigit(age); printf("\n");
    SyscallHandler::sys_exit();
}
```

```
Welcome to MyOS
How old are you? 34
Your age is 34
### syscall exit ###
KEYBOARD 0x38KEYBOARD 0x1D
```

The above result is from testScanf which basically ask the age of user and prints it.

```
Welcome to MyOS
program: LinearSearch
array : 10 20 80 30 60 50 110 100 130 170
target value: 45
LinearSeach(45): -1
program: BinarySearch
array: 10 20 80 30 60 50 110 100 130 170
sorted array: 10 20 30 50 60 80 100 110 130 170
target value: fsdknldkf
Invalid format error
target value: 110
BinarySeach(110): 7
program: Collatz
sequence number: 4
Collatz(4):
4: 2 1
3: 10 5 16 8 4 2 1
2: 1
1: 1
### syscall exit ###
### syscall exit ###
KEYBOARD 0x38### syscall exit ###
KEYBOARD 0x1D
```

The above result is from lifecyle1. It can be seeing that inputs are read properly.

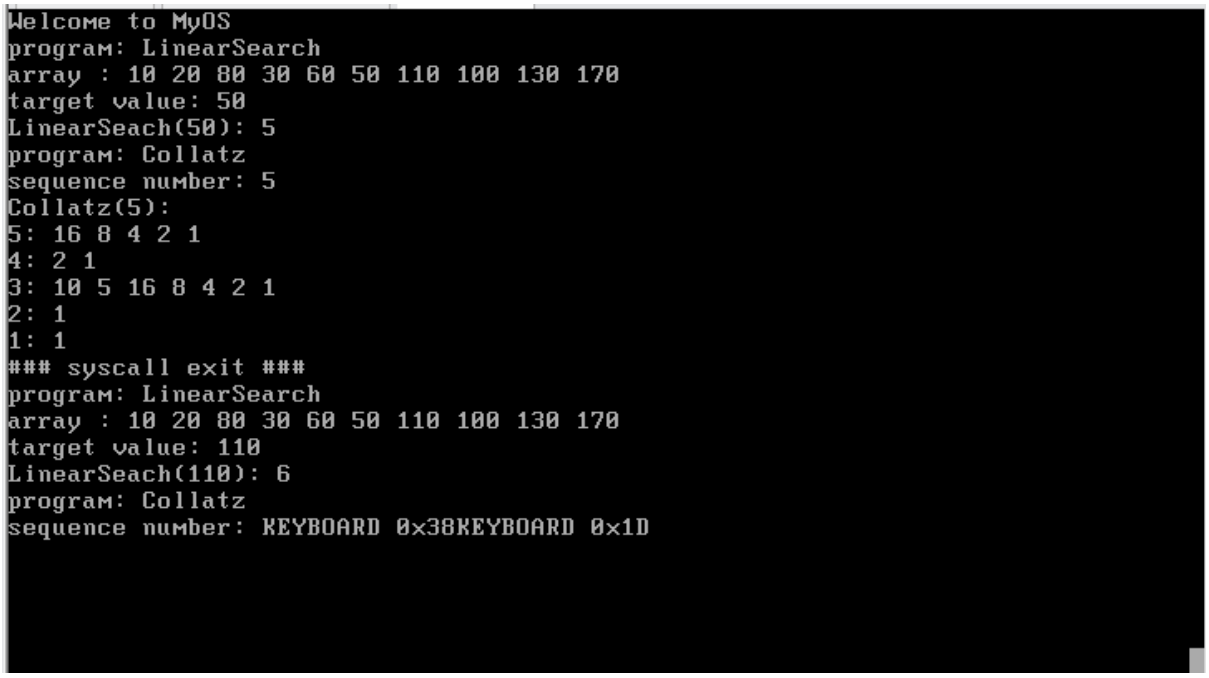
9.6 First Strategy

```
Welcome to MyOS
program: LinearSearch
array : 10 20 80 30 60 50 110 100 130 170
target value: 100
LinearSeach(100): 7
program: BinarySearch
array: 10 20 80 30 60 50 110 100 130 170
sorted array: 10 20 30 50 60 80 100 110 130 170
target value: fdfsdfbc
Invalid format error
target value: 50
BinarySeach(50): 3
program: Collatz
sequence number: 4
Collatz(4):
4: 2 1
3: 10 5 16 8 4 2 1
2: 1
1: 1
### syscall exit ###
### syscall exit ###
### syscall exit ###
KEYBOARD 0x38KEYBOARD 0x1D
```

9.7 Second Strategy

```
Welcome to MyOS
program: LinearSearch
array : 10 20 80 30 60 50 110 100 130 170
target value: 10
LinearSeach(10): 0
program: LinearSearch
array : 10 20 80 30 60 50 110 100 130 170
target value: 30
LinearSeach(30): 3
program: LinearSearch
array : 10 20 80 30 60 50 110 100 130 170
target value: 40
LinearSeach(40): -1
program: LinearSearch
array : 10 20 80 30 60 50 110 100 130 170
target value: 120
LinearSeach(120): -1
program: LinearSearch
array : 10 20 80 30 60 50 110 100 130 170
target value: 40
LinearSeach(40): -1
program: LinearSearch
array : 10 20 80 30 60 50 110 100 130 170
target value: KEYBOARD 0x38KEYBOARD 0x1D
```

9.8 Final Strategy



10 Postface

The expected requirements which are ignoring timer interrupt in a smooth way so that mouse and keyboard interrupts can be handled, taking the function parameters for the three tasks binary search, linear search, collatz sequence, and finally the test cases are done completely.