CSE312 OPERATING SYSTEMS
HOMEWORK 1

Implementing System calls and Scheduling Algorithms

Omer Kaan Uslu
1801042642
Department of Computer Engineering, Gebze Technical University

Contents

1 Introduction **3**

2 The TaskManager and Task Classes **3**

3 System Calls **4**

3.1 The fork system call 5

3.2 The execve system call 6

3.3 The waitpid system call 7

3.4 The exit system call 7

3.5 Tests and Results 8

4 Lifecycles, Strategies and All Other Things **9**

4.1 The Schedule Function 9

4.2 The Random Function 9

4.3 The Functions of tests 10

4.4 Part A, Lifecycle and Strategy 1 11

4.5 Part B, Strategy 1 13

4.6 Part B, Strategy 2 14

4.7 Part B, Strategy 3 15

1 Introduction

In this homework, we are wanted to understand the Viktor Engellman's Operating System and implement the system calls fork, execve, waitpid and exit. Firstly, I began the homework with trying to understand the operating system by watching all the videos. I watched all the videos to the 7th video. After that, I just skipped the graphics part because it is unnecasary to watch for us. Our goal is to implement system calls and run some programs on it. So after watching 7th video, I watched 15th video and it is very explaining. Before watching 15th video, I didn't know anything about what we can do and how we can do. After that, I watched 16th video(Dynamic Memory Management). And after that, I watched the 20th video which is very explaining about system calls implementation. After that, I started to write system calls.

2 The TaskManager and Task Classes

For the reasons for this homework, we are wanted to create processes instead of threads. So I have to edit TaskManager and Task classes accordingly.

For the Task class, I just added things for process like process id, parent process id, and it's state. The isParentWaiting variable is there for waitpid implementation which is explained in section 3.

The TaskManager class have new counters for process id etc. and have *ogullar* Task array to point child processes created and not to change original *tasks* array. And of course all other system calls are implemented in this class.

```
class TaskManager{
private:
    GlobalDescriptorTable* gdt;
    Task* tasks[256];
    Task ogullar[50];

    int numTasks;
    int currentTask;
    int pidCounter;
    int interruptCount;
    int ogulCount;
public:
    TaskManager(GlobalDescriptorTable* gdt);
    ~TaskManager();
    bool AddTask(Task* task,bool fromFork);
    CPUState* Schedule(CPUState* cpustate);
    void printProcessTable();

    //syscalls
    common::uint32_t fork(CPUState* cpu, int* child_pid);
    void exit();
    void waitpid(common::uint32_t child_pid);
    common::uint32_t execve(void (*entrypoint)());
};
```

Figure 1: The TaskManager Class

3 System Calls

For the system calls, I use the snippet Viktor Engellman created for printf system call. The SystemcallHandler's HandleInterrupt method handles the system calls with help of interrupts. When the interrupt comes with *0x80* interrupt index, the SystemcallHandler automatically handles the interrupt with it's HandleInterrupt method. Here it is;

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


    switch(cpu->eax){
    case 1:
        taskManager->exit();
        esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
        break;
    case 2:
        int child_pid;
        taskManager->fork(cpu,&child_pid);
        cpu->ecx = child_pid;
        printf("babu");
        break;
    case 4:
        printf((char*)cpu->ebx);
        break;
    case 7:
        taskManager->waitpid(cpu->ebx);
        esp = (uint32_t)taskManager->Schedule((CPUState*)esp);
        break;
    case 11:
        esp = taskManager->execve((void (*)()) cpu->ebx);
        break;
    default:
        break;
    }

    return esp;
}
```

Figure 2: The SystemcallHandler's HandleInterrupt method

The software interrupts comes here and where they are coming from is in Figure 4.

```
void sysfork(){
    asm("int $0x80" : : "a" (2));
}

void sysexit(){
    asm("int $0x80" : : "a" (1));
}

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

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

Figure 3: The system calls in kernel.cpp

I used the POSIX standarts for register values like in Figure 5.

```
void sysfork(){
    asm("int $0x80" : : "a" (2));
}

void sysexit(){
    asm("int $0x80" : : "a" (1));
}

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

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

Figure 4: POSIX system call Table

3.1 The fork system call

The fork system call is implemented like the Figure 6. Let's ignore the debugging protocols with print functions. It firstly sets up the pointers for child and parent process. After that, it copies the instruction pointer and code segment of parent process to child process. The stack duplication is handled afterwards. Since the cpustate values have to be same with parent process, it is copied. The base of the CPU state for the child is recalculated to ensure it points correctly within the newly copied stack. This involves calculating the offset from the base of the parent's CPU state and adjusting the child's CPU state pointer accordingly. After that, the child's ecx value is set to 0 to distinguish parent and child process. When the child process' pid returned to SystemcallHandler, it puts returned value to parent's ecx value. Finally, the childTask is added.

```
uint32_t TaskManager::fork(CPUState* cpu,int* child_pid){
    printf("LOW: ");
    print_int(currentTask);
    Task* baba = tasks[currentTask];
    Task *ogul = &ogullar[ogulCount];

    ogul->cpustate->eip = (uint32_t)baba->cpustate->eip;
    ogul->cpustate->cs = gdt->CodeSegmentSelector();


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

    ogul->cpustate = (CPUState*)(ogul->stack + (4096) - sizeof(CPUState));
    common::uint32_t offset = ((uint32_t)baba->cpustate) - ((uint32_t)cpu);
    ogul->cpustate = (CPUState*)(((uint32_t)ogul->cpustate) - offset);

    *(ogul->cpustate) = *cpu;

    ogul->cpustate->ecx = 0;


    AddTask(&(*ogul), true);


    int pidValue = ogul->pid;
    (*child_pid) = pidValue;
    printf("\n\n\n\n\n");

    ogulCount++;

    return (uint32_t) tasks[currentTask]->cpustate;
}
```

Figure 5: The Fork system call

3.2 The execve system call

The execve system call is more easy to understand, it basically changes the child process' instruction pointer to newly given entrypoint and just returns the current tasks cpustate. It returns esp to interrrupt handler in SystemcallHandler. When I didn't do it it just continues without doing execve.

```
uint32_t TaskManager::execve(void entrypoint()){
    printf("execve girdi");
    print_int(currentTask);
    int pid = tasks[currentTask]->pid;
    int pPid = tasks[currentTask]->pPid;

    tasks[currentTask] = new Task(gdt, entrypoint);
    tasks[currentTask]->pid = pid;
    tasks[currentTask]->pPid = pPid;
    tasks[currentTask]->state = READY;
    printf("execve cikis");
    return (uint32_t) tasks[currentTask]->cpustate;
}
```

Figure 6: The Execve system call

3.3 The waitpid system call

The waitpid system call is very basic. It just sets the child pid's *isParentWaiting* variable to true and sets parent processes status to *BLOCKED*. In the schedule system calls shown in section 4, it is guaranteed that the *BLOCKED* process is not scheduled.

```
void TaskManager::waitpid(uint32_t child_pid){
    printf(" waitpid girdi");
    print_int(child_pid);
    for(int i = 0; i < numTasks; i++){
        if(tasks[i]->pid == child_pid){
            tasks[currentTask]->state = BLOCKED;
            tasks[i]->isParentWaiting = true;
            return;
        }
    }
    // if there is no child, don't do anything..
}
```

Figure 7: The Waitpid system call

3.4 The exit system call

The exit system call is as simple as waitpid. It just controls there are any parent process waiting for this process it just awake the parent process up by setting it's state to READY. After that, it just sets it's status to *TERMINATED*. In the schedule system calls shown in section 4, it is guaranteed that the *TERMINATED* process is not scheduled.

```
void TaskManager::exit(){
    printf("exit");
    if(tasks[currentTask]->isParentWaiting){
        printf("dumen");
        for(int i = 0; i < numTasks; i++){
            if(tasks[i]->pid == tasks[currentTask]->pPid){
                tasks[i]->state = READY;
                return;
            }
        }
    }

    tasks[currentTask]->state = TERMINATED; // this task never w
}
```

Figure 8: The Exit system call

3.5 Tests and Results

The test case code is shown in Figure 10. The result is shown in Figure 11. It is shown that all the system calls are working well. The fork's return value got from ecx value with help of assembly.

```
void oylesine(){
    printf("execve calisiyor");
    sysexit();
}
void taskA(){
    int i = 0;
    sysfork();

    int value_from_ecx;
    asm("movl %%ecx, %0": "=r" (value_from_ecx));
    print_int(value_from_ecx);

    if(value_from_ecx == 0){// child process

        printf("A");
        print_int(i);
        sysexit();
    }
    else{//parent process
        print_int(i);
        i = 1;
        printf("B");
        printf(" after waitpid");


    }
    sysfork();

    asm("movl %%ecx, %0": "=r" (value_from_ecx));
    print_int(value_from_ecx);

    if(value_from_ecx == 0)
        printf("fork yeni bitti1");
    else
        printf("fork yeni bitti2");
    while(1){
    }
}
```

Figure 9: The Systemcall Test Case

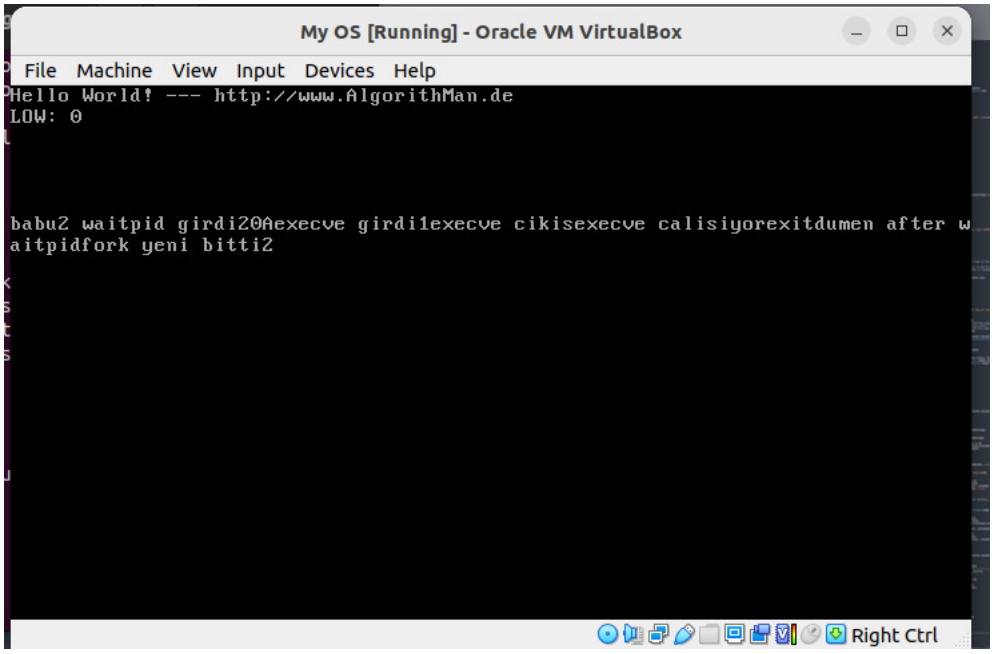

Figure 10: The Systemcall Test Case Result

4 Lifecycles, Strategies and All Other Things

4.1 The Schedule Function

The schedule function is implemented like this. If processes are TERMINATED or BLOCKED, the process is not scheduled. It works with standart round robin style.

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

    printProcessTable();

    if(currentTask >= 0){
        tasks[currentTask]->cpustate = cpustate;
    }
    if(tasks[currentTask]->state == RUNNING)
        tasks[currentTask]->state = READY;


if(++currentTask >= numTasks){
        currentTask %= numTasks;
} // if the task is blocked or terminated do not schedule it.

while(tasks[currentTask]->state == BLOCKED || tasks[currentTask]->state == TERMINATED){
    //print_int(currentTask);
    if(++currentTask >= numTasks){
        currentTask %= numTasks;
    }
}
    //print_int(currentTask);
    tasks[currentTask]->state = RUNNING;


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

Figure 11: The Schedule Function

4.2 The Random Function

The random function generated in kernel.cpp is implemented using Linear congruential generator[1] way. The implementation[2] is from stackoverflow answer. It first sets up the

seed and generating random numbers accordingly with modulo operations.

```
long a = 25214903917;
long c = 11;
long previous = 17;


int rand(){
    long r = a* previous + c;
    previous = r;

    if(r < 0) return r*(-1);
    return r;
}
```

Figure 12: The Random Function

4.3 The Functions of tests

All are written by ChatGPT[3]. I didn't add anything because they are just to test.

```
void printCollatz() { // chat gpt
    for(int i = 0;i<100;i++){
    int n = 7;
    printf("Collatz sequence for ");
    print_int(n);
    printf("\n");
    while (n != 1) {
        print_int(n);
        printf("  ");
        if (n % 2 == 0) {
            n /= 2;
        } else {
            n = 3 * n + 1;
        }
    }
    print_int(n);  // Print the last number in the sequence (which is 1)
 }
    sysexit();
    while(1);
}
```

Figure 13: The Collatz Function

```
void long_running_program() { // chat gpt
    int n = 1000;
    int result = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            result += i * j;
        }
    }
    printf("result long running: ");
    print_int(result);
    sysexit();

    while(1);
}
```

Figure 14: The Long Running program Function

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

    for (int i = 0; i < n; i++) {
        if (arr[i] == x) {
            print_int(i);
            sysexit(); // Eğer değer bulunursa indeksini döndür
        }
    }
    print_int(-1);
    sysexit();  // Eğer değer bulunamazsa -1 döndür
}
```

Figure 15: The Linear Search Function

```
void binarySearch() {
        printf("binary");

    int arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170};
    int x = 110;
    int n = sizeof(arr) / sizeof(arr[0]);
    int low = 0, high = n - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        // Eğer x orta değerse, ortanın indeksini döndür
        if (arr[mid] == x){
            print_int(mid);
            sysexit();
        }

        // Eğer x orta değerden büyükse, sol yarıyı atla
        if (arr[mid] < x) low = mid + 1;

        // Eğer x orta değerden küçükse, sağ yarıyı atla
        else high = mid - 1;
    }

    // Eğer element bulunamazsa -1 döndür
    print_int(-1);
    sysexit();
}
```

Figure 16: The Binary Search Function

4.4 Part A, Lifecycle and Strategy 1

The part A has 1 strategy and it is loading 3 programs 3 times and doing round robin. I written it and tested it. Unfortunately, with all other tests, I can't see the programs actual part any time because it flows so fast. But the result process table is shown and working properly.

```
void partA(){
    int a;
    for(int i = 0;i < 3;i++){ // run print Collatz 3 times and long running 3 ties
        sysfork();
        int value_from_ecx;
        asm("movl %%ecx, %0": "=r" (value_from_ecx));
        print_int(value_from_ecx);

        if(value_from_ecx == 0){
            printCollatz();
            sysexit();
        }
    }

    for(int i = 0;i < 3;i++){ // run print Collatz 3 times and long running 3 ties
        sysfork();
        int value_from_ecx;
        asm("movl %%ecx, %0": "=r" (value_from_ecx));
        print_int(value_from_ecx);

        if(value_from_ecx == 0){
            long_running_program();
            sysexit();
        }
    }
    printf("fordan cikti baba");
    while(1);
}
```

Figure 17: Part A, Strategy 1

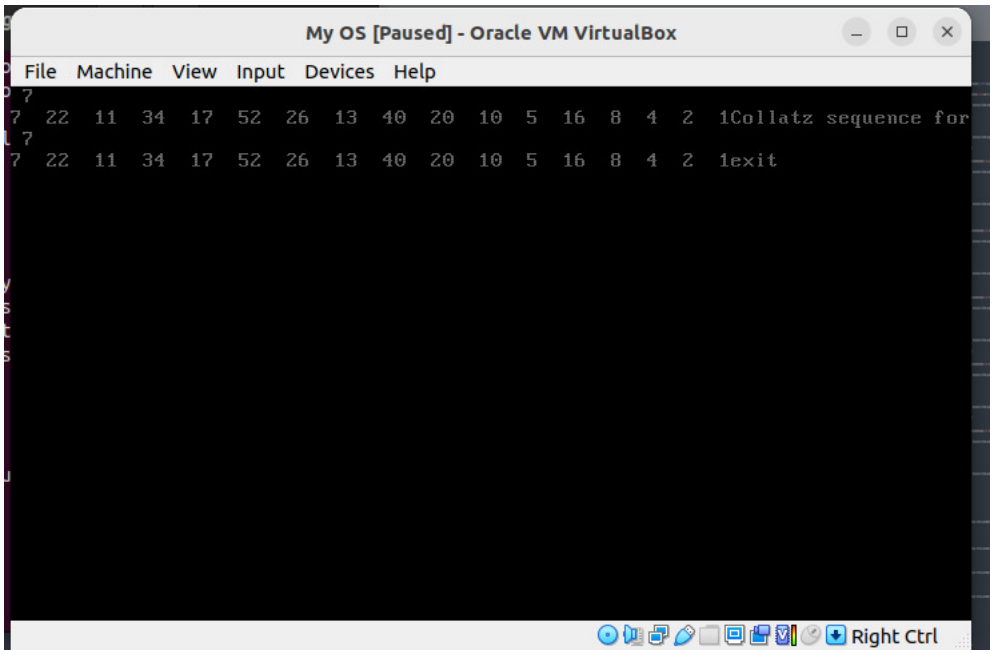

Figure 18: Part A, Strategy 1 Result 1

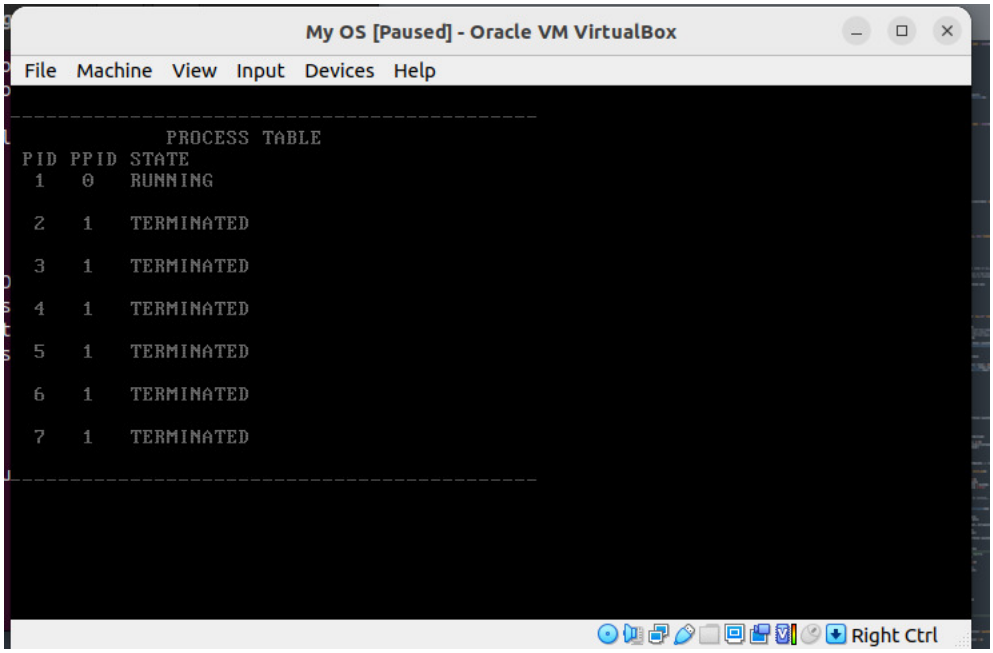

Figure 19: Part A, Strategy 1 Result 2

4.5 Part B, Strategy 1

I used random function and for loops.

```
void bStartegyOne(){
    int random = rand() % 4;

    if(random == 0){
        for(int i = 0;i < 10;i++){ // run print Collatz 3 times and long running 3 ties
        sysfork();
        int value_from_ecx;
        asm("movl %%ecx, %0": "=r" (value_from_ecx));
        print_int(value_from_ecx);

        if(value_from_ecx == 0){
            printCollatz();
            sysexit();
            }
        }
    }
    else if(random == 1){
        for(int i = 0;i < 10;i++){ // run print Collatz 3 times and long running 3 ties
        sysfork();
        int value_from_ecx;
        asm("movl %%ecx, %0": "=r" (value_from_ecx));
        print_int(value_from_ecx);

        if(value_from_ecx == 0){
            long_running_program();
            sysexit();
            }
        }
    }
    else if(random == 2){
        for(int i = 0;i < 10;i++){ // run print Collatz 3 times and long running 3 ties
        sysfork();
        int value_from_ecx;
        asm("movl %%ecx, %0": "=r" (value_from_ecx));
        print_int(value_from_ecx);

        if(value_from_ecx == 0){
            binarySearch();
```

Figure 20: Part B, Strategy 1

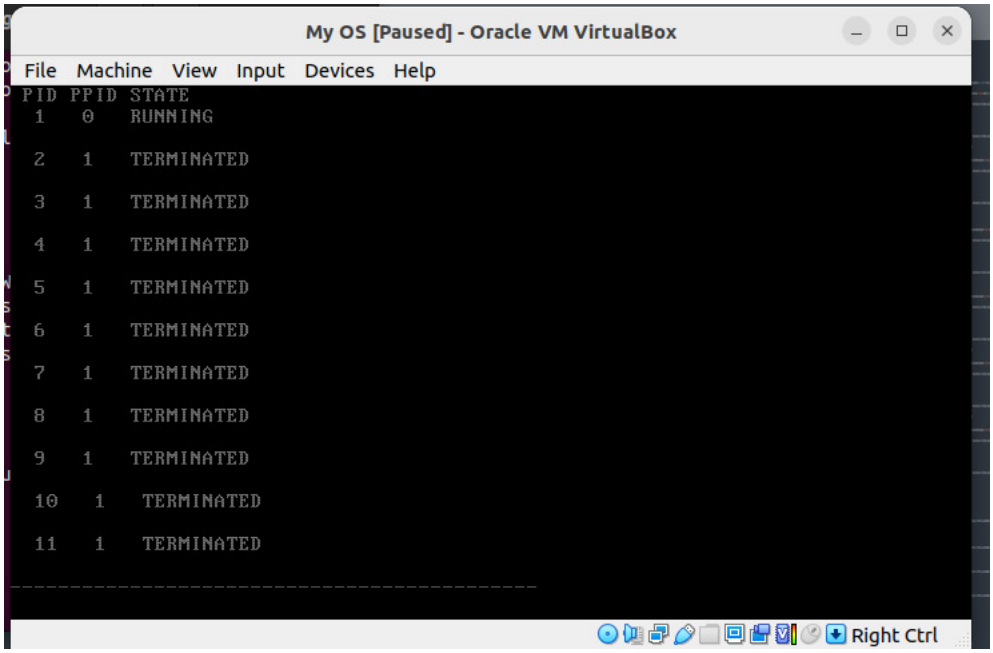

Figure 21: Part B, Strategy 1 Result

4.6 Part B, Strategy 2

I used random function and for loops.

```
void bStrategyTwo(){
    int firstRandom = rand() % 4;
    int secondRandom = rand() % 4;

    while(firstRandom == secondRandom){// if they are same, change second random value
        secondRandom = rand();
    }

    if(firstRandom == 0 && secondRandom == 1){//long running and print collatz
        for(int i = 0;i < 3;i++){
        sysfork();
        int value_from_ecx;
        asm("movl %%ecx, %0": "=r" (value_from_ecx));
        print_int(value_from_ecx);

        if(value_from_ecx == 0){
            printCollatz();
            sysexit();
            }
        }

        for(int i = 0;i < 3;i++){
        sysfork();
        int value_from_ecx;
        asm("movl %%ecx, %0": "=r" (value_from_ecx));
        print_int(value_from_ecx);

        if(value_from_ecx == 0){
            long_running_program();
            sysexit();
            }
        }
    }
```

Figure 22: Part B, Strategy 2

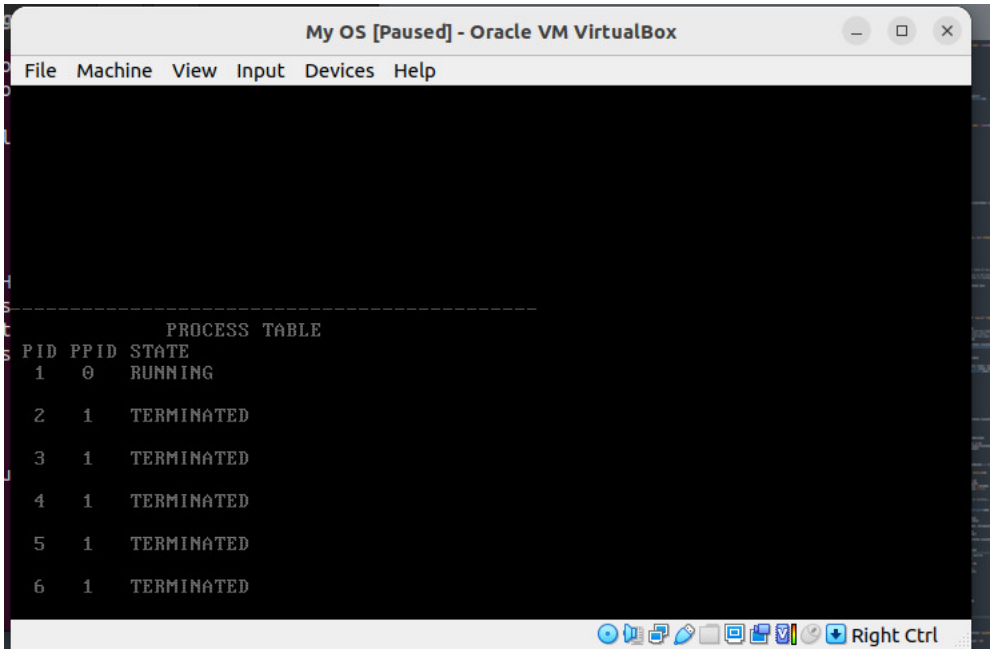

Figure 23: Part B, Strategy 2 Result

4.7 Part B, Strategy 3

For this one, I just edited the Schedule function of 3 lines. Aging is implemented this way.

```
void bStrategyThree(){
    sysfork();
    int value_from_ecx;
    asm("movl %%ecx, %0": "=r" (value_from_ecx));
    print_int(value_from_ecx);

    if(value_from_ecx == 0){
        //printCollatz();
        sysexit();
    }


    sysfork();
    asm("movl %%ecx, %0": "=r" (value_from_ecx));
    print_int(value_from_ecx);

    if(value_from_ecx == 0){
        long_running_program();
        sysexit();
    }

    sysfork();
    asm("movl %%ecx, %0": "=r" (value_from_ecx));
    print_int(value_from_ecx);

    if(value_from_ecx == 0){
        binarySearch();
        sysexit();
    }

    sysfork();
    asm("movl %%ecx, %0": "=r" (value_from_ecx));
    print_int(value_from_ecx);

    if(value_from_ecx == 0){
        linearSearch();
        sysexit();
    }

    printf("Fork bitti baba");
    while(1);
}
```

Figure 24: Part B, Strategy 3

```
interruptCount++;
if(numTasks <= 0 || interruptCount < 5)
    return cpustate;
```

Figure 25: The edited part in schedule

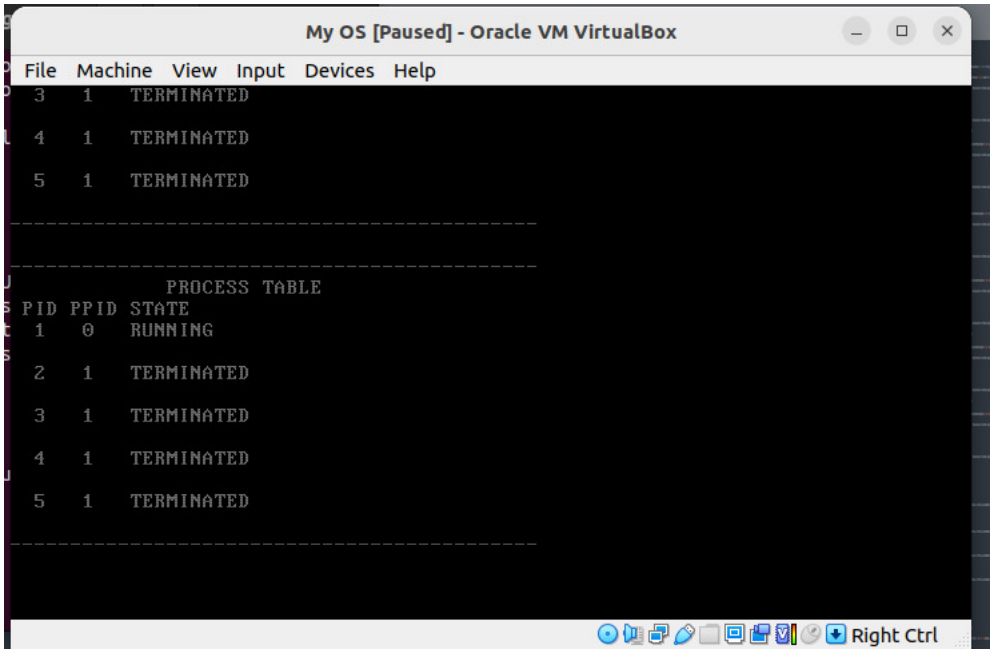

Figure 26: Part B, Strategy 3 Result

References

[1] Wikipedia, "Linear congruential generator," https://en.wikipedia.org/wiki/Linear_congruential_generator, accessed: [2024, May].

[2] S. Overflow, "Generate random numbers without using any external functions," https://stackoverflow.com/questions/15038174/generate-random-numbers-without-using-any-external-functions, accessed: [2024, May].

[3] OpenAI, "Collatz, long running program, linear search, binary search," https://chatgpt.com, accessed: [2024, May].