Gebze Technical University

CSE312 - Operating Systems
Midterm Project Report

Selim AYNİGÜL

200104004004

Spring 2024

Program Structure

My homework includes Part A and Part B requirements of the homework. I will explain the essential parts briefly.

Syscalls

All requested functionalities in the homework are done by syscalls in the program. Not much difference from Engellman's original source code but just additions to it. When a syscall is made it calls the interrupt handler and creates an interrupt. Then interrupt handler calls the necessary function.

syscalls.cpp

```
uint32_t SyscallHandler::HandleInterrupt(uint32_t esp)
{
    CPUState* cpu = (CPUState*)esp;
    switch(cpu->eax)
    {
        case Syscalls::PRINTF:
            printf((char*)cpu->ebx);
            break;
        case Syscalls::EXEC:
            esp = InterruptHandler::exec(cpu->ebx);
            break;
        case Syscalls::FORK:
            cpu->ecx = InterruptHandler::fork(cpu);
            return InterruptHandler::HandleInterrupt(esp);
            break;
        case Syscalls::EXIT:
           if(InterruptHandler::exit()) {
              return InterruptHandler::Reschedule(esp);
           }
            break;
        case Syscalls::GETPID:
            cpu->ecx = InterruptHandler::getpid();
            break;
        case Syscalls::WAITPID:
            if(InterruptHandler::waitpid(esp)) {
                return InterruptHandler::Reschedule(esp);
            }
            break;
        default:
            break;
    }
    return esp;
}
```

interrupts.cpp

```
bool InterruptHandler::waitpid(common::uint32_t pid) {
    return interruptManager->taskManager->WaitTask(pid);
}
```

waitpid example for interrupt handler. Syscall for wait is calling this function to handle.

Initialization

Program starting initializing the init process. In InitTask function the init process is added to tasks as the initial process. initProcess() selects the necessary function according the given strategy input to test different parts of the homework.

kernel.cpp

```
    Task initTask(&gdt, initProcess);
    taskManager.InitTask(&initTask);

```

multitasking.cpp

```
// add initial process to tasks
bool TaskManager::InitTask(Task* task) {
    AddTask(task, &tasks[numTasks]);
    task->priority = 1;
    numTasks++;
    return true;
}
```

Fork Task

Shared fork function by the assistant's logic is used for forking. This creates a copy of the parent process in the tasks array. Calculates the posisition on the cpustate in the stack, returns pid for parent and puts 0 to ecx register for child process itself to distinct between parent and child in the main.

multitasking.cpp

```
common::uint32_t TaskManager::ForkTask(CPUState* cpustate) {

    // fork for part a, b1, b2
    if (numTasks >= 256)
        return -1;

    tasks[numTasks].state = ProcessState::READY;
    tasks[numTasks].pid = numTasks;
    tasks[numTasks].ppid = getpid();

    common::uint32_t currentTaskOffset = (((common::uint32_t)cpustate - (common::uint32_t) tasks[currentTask].stack));
    tasks[numTasks].cpustate = (CPUState*)(((common::uint32_t) tasks[numTasks].stack) + currentTaskOffset);

    for (int i = 0; i < sizeof(tasks[currentTask].stack); i++)
        tasks[numTasks].stack[i] = tasks[currentTask].stack[i];

    tasks[numTasks].cpustate->ecx = 0;
    numTasks++;

    return tasks[numTasks - 1].pid;
}
```

Waiting and Exiting Processes

waitpid

Wait function returns false if the process try to wait itself or the prcoess it waits is already finished. If the process still continues then it blocks itself and increases the counter for the how many process it waits.

multitasking.cpp

```
bool TaskManager::WaitTask(common::uint32_t esp) {
    CPUState* cpustate = (CPUState*)esp;
    common::uint32_t pid = cpustate->ebx;

    printf("PID ");
    printfInt(tasks[currentTask].pid);
    printf(" is waiting for PID ");
    printfInt(pid);
    printf("...\n");

    // if the waited process already terminated return false without waiting
    if (tasks[pid].state == ProcessState::TERMINATED || pid == tasks[currentTask].pid) {
        tasks[currentTask].cpustate->ecx = 0;
        return false;
    }

    tasks[pid].waitparent = true;
    tasks[currentTask].state = ProcessState::BLOCKED;
    tasks[currentTask].waitnum++;
    tasks[currentTask].cpustate->ecx = 1;

    return true;
}      You, 6 days ago • fixes
```

exit

Exit tasks makes the state of the process terminated. If waitparent value is true that means another process is waiting for this process to finish. So in that condition it changes necessary state info of the waiting process and if it is the last process that the other waits it removes blocked status.

multitasking.cpp

```
bool TaskManager::ExitTask() {
    printf("Process with PID ");
    printfInt(currentTask);
    printf(" exited.\n");
    tasks[currentTask].state = ProcessState::TERMINATED;

    // if it's parent waits its child to terminate adjust necessary parts in the parent
    if (tasks[currentTask].waitparent) {
        int ppid = tasks[currentTask].ppid;
        if (tasks[ppid].state == BLOCKED) {
            // if there are other childs that the parent waits, just decrease waitnum
            if (tasks[ppid].waitnum > 1) {
                tasks[ppid].waitnum--;
            }
            // if this is the last child that parent waits make parent's state ready
            else {
                tasks[ppid].waitnum--;
                tasks[ppid].state = ProcessState::READY;
            }
        }
    }
    return true;
}
```

Scheduling

Robin Round Scheduling

Everytime a timer interrupt occurs schedule function executes. Robin round scheduling takes the next ready process in the circular queue.

multitasking.cpp

```cpp
//schedule for part a, b.1, b.2
CPUState* TaskManager::ScheduleRobinRound(CPUState* cpustate, int interruptCount) {
    if (numTasks <= 0)
        return cpustate;

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

    int nextTask = currentTask;

    do {
        nextTask = (nextTask + 1) % numTasks;
    } while (tasks[nextTask].state != ProcessState::READY);

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

Preemptive Scheduling

In preemptive scheduling the higher priority processes are executing first. If the priority of mulitple processes are equeal then the first comes will be executing until it finishes. The schedule function checks the highest indexed task in the tasks array and switch to it.

multitasking.cpp

```cpp
CPUState* TaskManager::SchedulePreemptive(CPUState* cpustate, int interruptCount) {

    // schedule for part b.3
    // schedule priority

    if (numTasks <= 0)
        return cpustate;

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

    // other processes after first one will arrive after 5th interrupt
    // so here we change them from blocked to ready
     if (interruptCount == 5) {
        for (int i = 0; i < numTasks; i++) {
            if (i > 1 && tasks[i].state == ProcessState::BLOCKED) {
                tasks[i].state = ProcessState::READY;
            }
        }
    }

    int highestPriorityTask = -1;
    for (int i = 0; i < numTasks; ++i) {
        if (tasks[i].state == ProcessState::READY || tasks[i].state == ProcessState::RUNNING) {
            if (highestPriorityTask == -1 || tasks[i].priority > tasks[highestPriorityTask].priority) {
                highestPriorityTask = i;
            }
        }
    }

    if (highestPriorityTask == -1) {
        // No READY task found, return the current CPU state
        return cpustate;
    }

     // save current cputstate of the current task
    if (currentTask >= 0 && tasks[currentTask].state == ProcessState::RUNNING) {
        tasks[currentTask].cpustate = cpustate;
        tasks[currentTask].state = ProcessState::READY;
    }

    currentTask = highestPriorityTask;
    tasks[currentTask].state = ProcessState::RUNNING;
    return tasks[currentTask].cpustate;
}
```

Dynamic Preemptive Scheduling

Dynamic scheduling is nearly the same with the static priority scheuduling except one main difference. It checks the first process every five timer interrupt as requested in the homework and if it is not terminated it increases its priority by one.

multitasking.cpp

```
CPUState* TaskManager::ScheduleDynamic(CPUState* cpustate, int interruptCount) {

    // schedule for part b.4

    if (numTasks <= 0)
        return cpustate;

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

    // if the first process is not terminated increase it's priority every 5 interrupt
    if (interruptCount % 5 == 0) {
        if (tasks[1].state != ProcessState::TERMINATED) {
            tasks[1].priority++;
        }
    }

    // save current cputstate of the current task
    if (currentTask >= 0 && tasks[currentTask].state == ProcessState::RUNNING) {
        tasks[currentTask].cpustate = cpustate;
        tasks[currentTask].state = ProcessState::READY;
    }

    int highestPriorityTask = -1;
    for (int i = 0; i < numTasks; ++i) {
        if (tasks[i].state == ProcessState::READY) {
            if (highestPriorityTask == -1 || tasks[i].priority > tasks[highestPriorityTask].priority) {
                highestPriorityTask = i;
            }
        }
    }

    if (highestPriorityTask == -1) {
        // No READY task found, return the current CPU state
        return cpustate;
    }

    currentTask = highestPriorityTask;
    tasks[currentTask].state = ProcessState::RUNNING;
    return tasks[currentTask].cpustate;
}
```

How to Test?

My source files include all Part A and Part B requirements. To test them:

Search for "int strategy" in the interrupts.h file. There are two of them in differenc classes. Change their values to test different parts.,

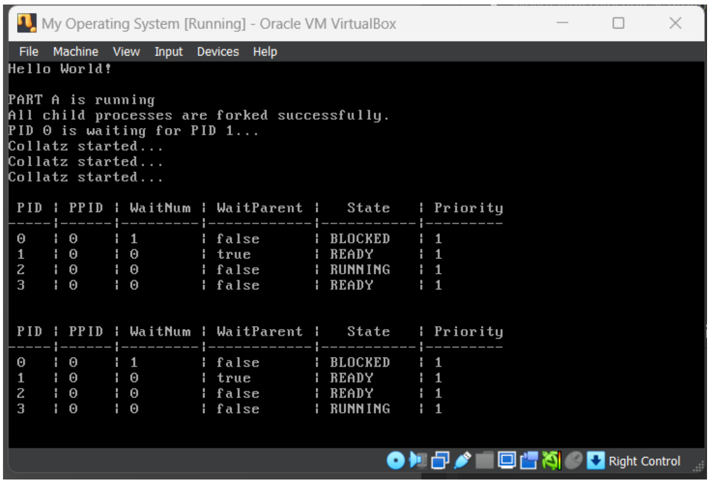
0 for part a,
1 for part b first strategy,
2 for part b second strategy,
3 for part b third strategy,
4 for part by dynamic strategy.

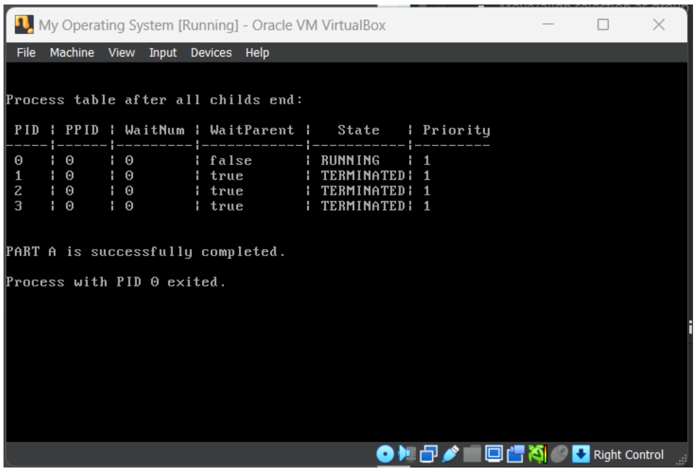
Example Otuputs

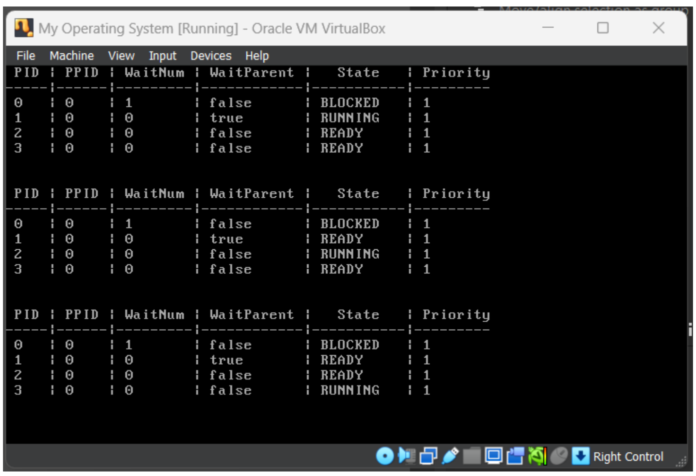
Part A

In part a, we use three nested forks and wait for the three child process to exit using waitpid.

Since robin round scheduling is used in this part, running process is switching on every timer interrupt as you can see in the outputs.

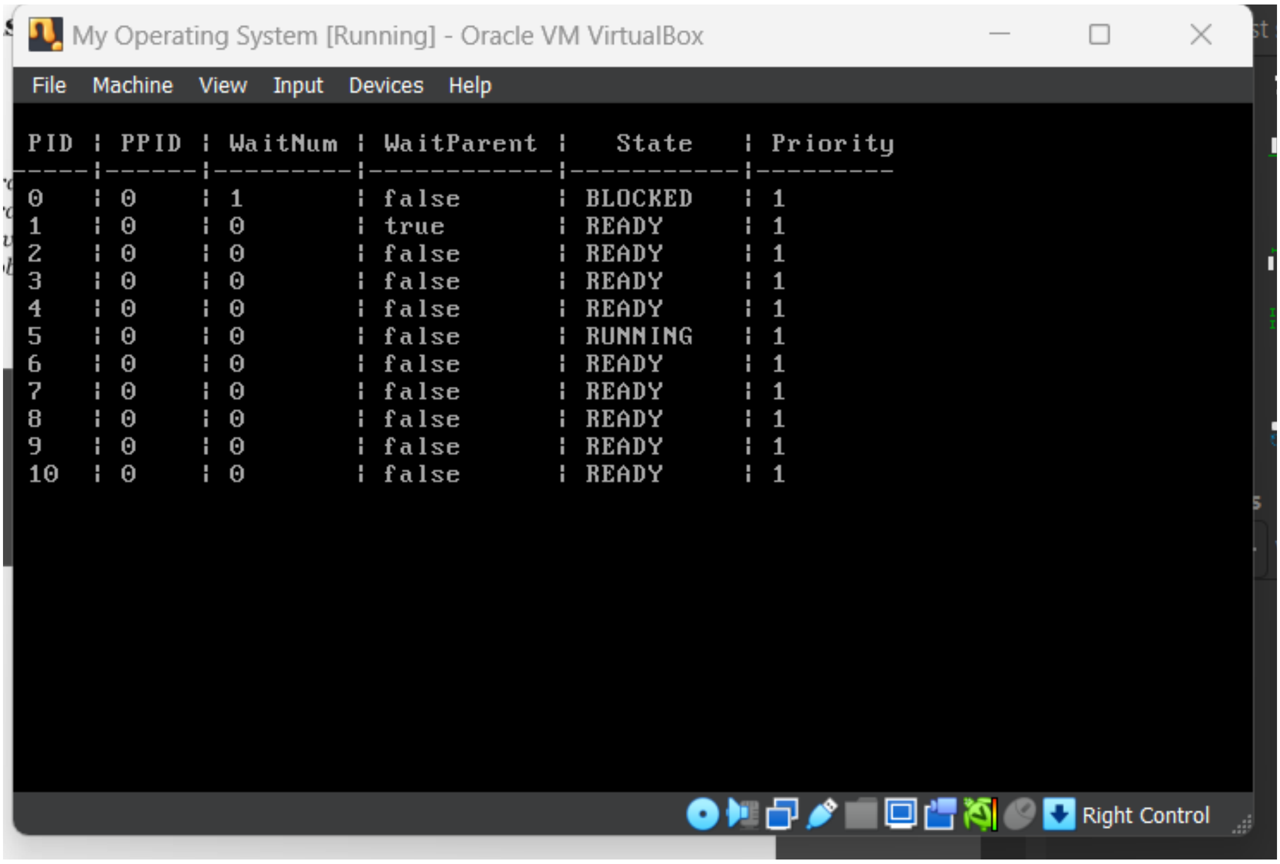




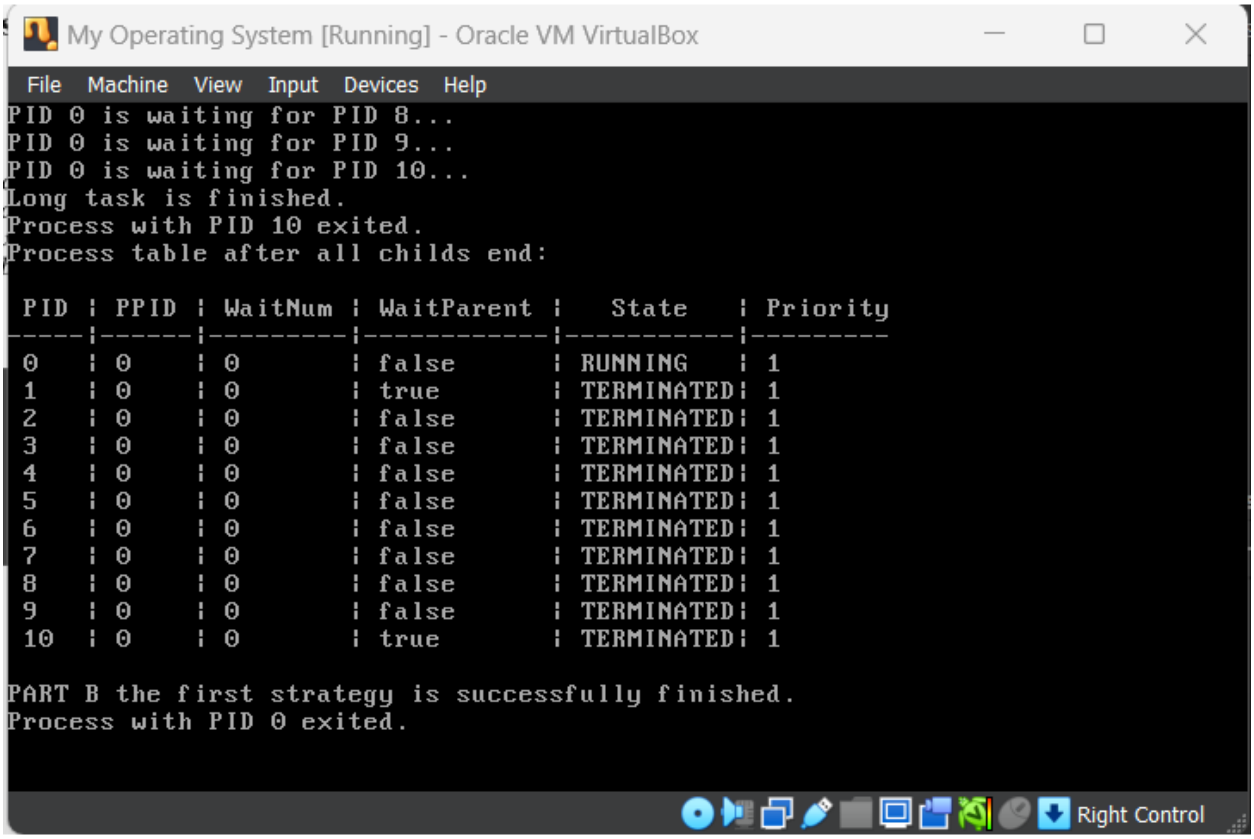


Part B First Strategy

In this part the only difference with part a is we fork the same task 10 times and wait them in the parent. waitpid is blockes parent process until all the childs exit. Exit makes the process status terminated.



PID	PPID	WaitNum	WaitParent	State	Priority
0	0	1	false	BLOCKED	1
1	0	0	true	READY	1
2	0	0	false	READY	1
3	0	0	false	READY	1
4	0	0	false	READY	1
5	0	0	false	RUNNING	1
6	0	0	false	READY	1
7	0	0	false	READY	1
8	0	0	false	READY	1
9	0	0	false	READY	1
10	0	0	false	READY	1




```
PID 0 is waiting for PID 8...
PID 0 is waiting for PID 9...
PID 0 is waiting for PID 10...
Long task is finished.
Process with PID 10 exited.
Process table after all childs end:
```

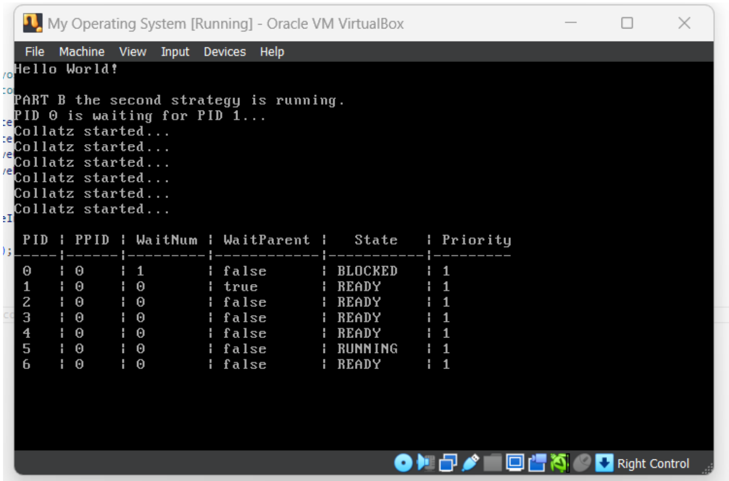
PID	PPID	WaitNum	WaitParent	State	Priority
0	0	0	false	RUNNING	1
1	0	0	true	TERMINATED	1
2	0	0	false	TERMINATED	1
3	0	0	false	TERMINATED	1
4	0	0	false	TERMINATED	1
5	0	0	false	TERMINATED	1
6	0	0	false	TERMINATED	1
7	0	0	false	TERMINATED	1
8	0	0	false	TERMINATED	1
9	0	0	false	TERMINATED	1
10	0	0	true	TERMINATED	1

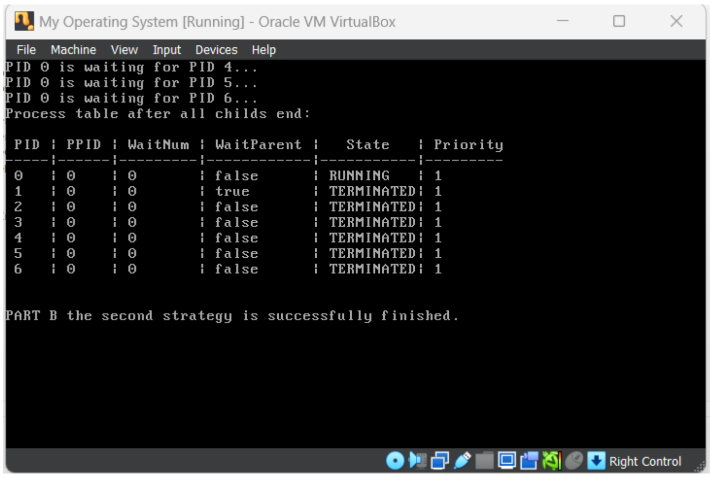
```
PART B the first strategy is successfully finished.
Process with PID 0 exited.
```

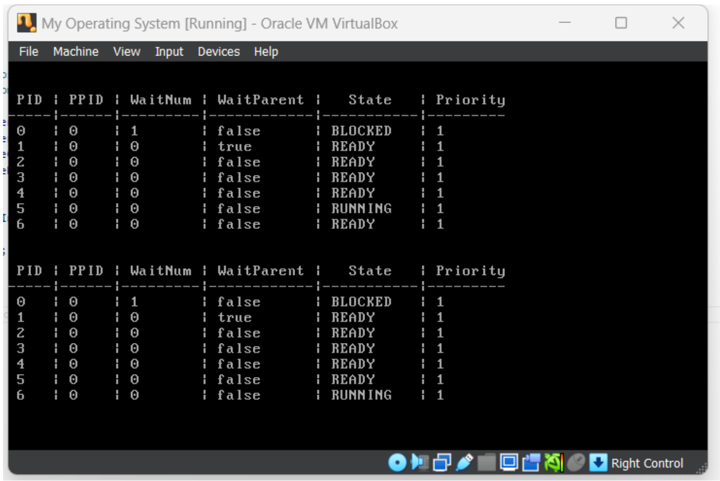
Part B Second Strategy

Same with the first strategy. Just different tasks are used.

Selected tasks switching on every interrupt since we use robin round scheduling.

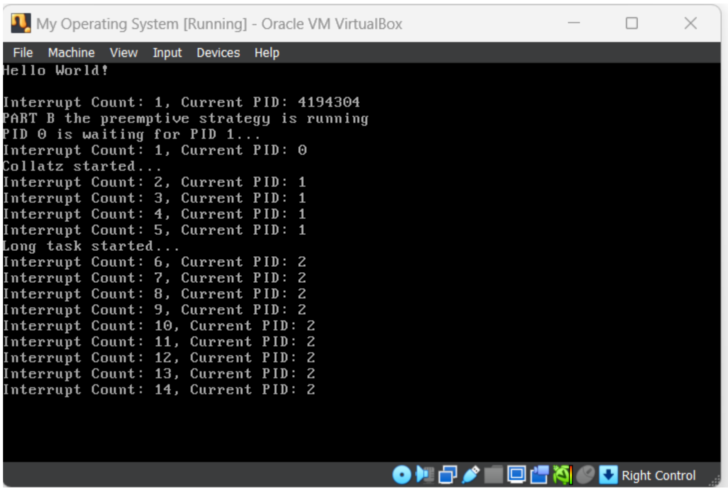




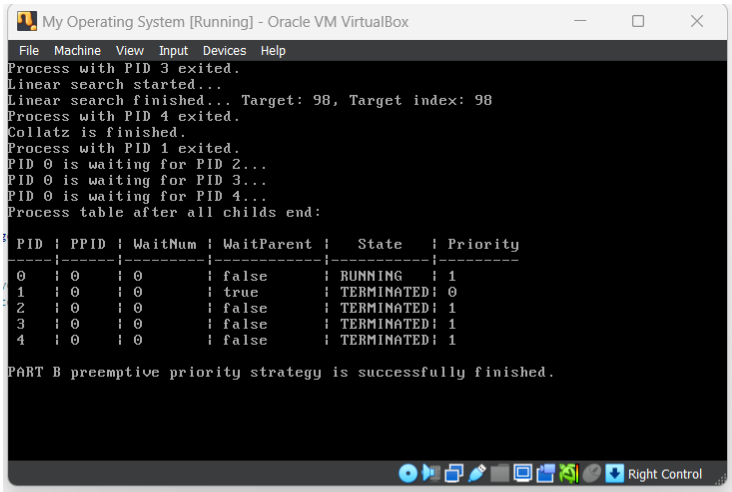


Part B Third Strategy

In this part you can see that first 5 interrupt collatz executes but after 5th interrupt other processes are arrived and they continues to execute. After all of them is finished collatz continues.



Final version of the process table. You can see that collatz exited as the last process since it has a lower priority.

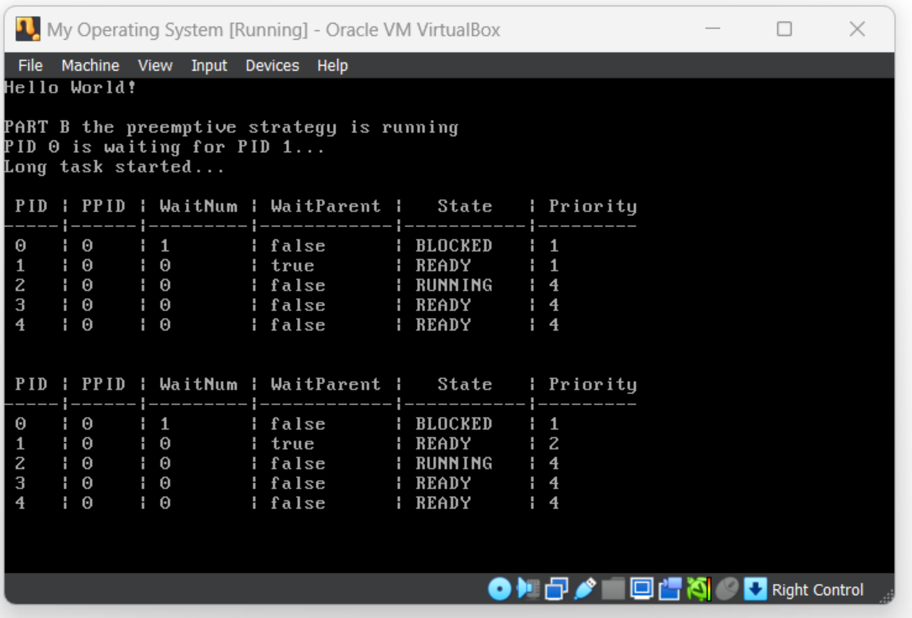


Part B Dynamic Strategy

As in the outputs, at the start the first process don run since it has a low priority, bu after some time it has higher priority since it's priority increased every 5 interrupt, so it starts running until finishing it's job. After that the next processes coninously executes their job.

At first the collatz program not executes since it has a lower priority. So other equal priority tasks executes in a first come first executes manner. So the second tasks continues without switching.

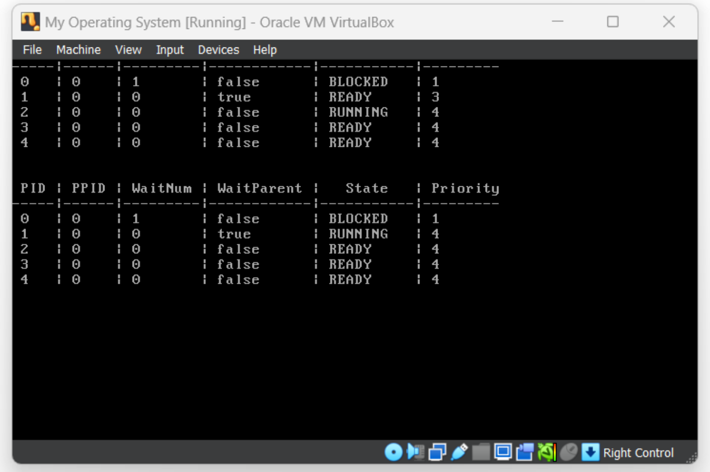
1)



```
Hello World!

PART B the preemptive strategy is running
PID 0 is waiting for PID 1...
Long task started...

 PID ¦ PPID ¦ WaitNum ¦ WaitParent ¦   State    ¦ Priority
-----¦------¦---------¦------------¦------------¦---------
 0   ¦ 0    ¦ 1       ¦ false      ¦ BLOCKED    ¦ 1
 1   ¦ 0    ¦ 0       ¦ true       ¦ READY      ¦ 1
 2   ¦ 0    ¦ 0       ¦ false      ¦ RUNNING    ¦ 4
 3   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
 4   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4

 PID ¦ PPID ¦ WaitNum ¦ WaitParent ¦   State    ¦ Priority
-----¦------¦---------¦------------¦------------¦---------
 0   ¦ 0    ¦ 1       ¦ false      ¦ BLOCKED    ¦ 1
 1   ¦ 0    ¦ 0       ¦ true       ¦ READY      ¦ 2
 2   ¦ 0    ¦ 0       ¦ false      ¦ RUNNING    ¦ 4
 3   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
 4   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
```

Priority of the first task is increasing by one every five interrupt.

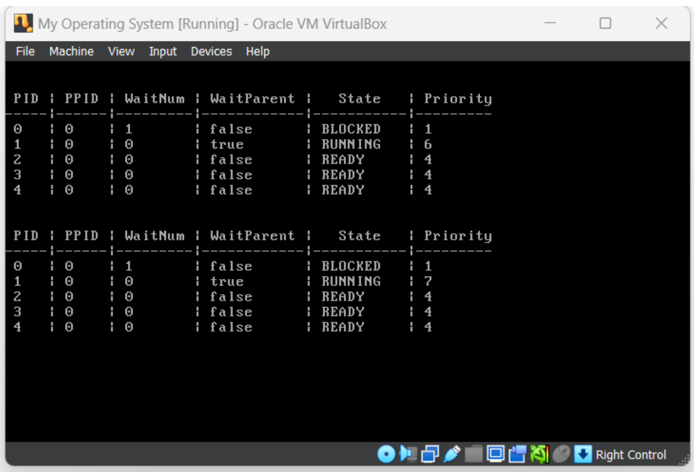
2)



```
-----¦------¦---------¦------------¦------------¦---------
 0   ¦ 0    ¦ 1       ¦ false      ¦ BLOCKED    ¦ 1
 1   ¦ 0    ¦ 0       ¦ true       ¦ READY      ¦ 3
 2   ¦ 0    ¦ 0       ¦ false      ¦ RUNNING    ¦ 4
 3   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
 4   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4

 PID ¦ PPID ¦ WaitNum ¦ WaitParent ¦   State    ¦ Priority
-----¦------¦---------¦------------¦------------¦---------
 0   ¦ 0    ¦ 1       ¦ false      ¦ BLOCKED    ¦ 1
 1   ¦ 0    ¦ 0       ¦ true       ¦ RUNNING    ¦ 4
 2   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
 3   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
 4   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
```

After having equal priority with others the first task started executing until finish.

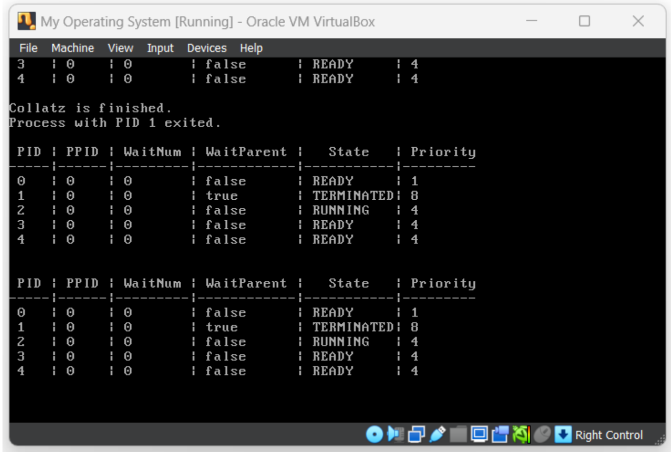
3)



```
 PID ¦ PPID ¦ WaitNum ¦ WaitParent ¦   State    ¦ Priority
-----¦------¦---------¦------------¦------------¦---------
 0   ¦ 0    ¦ 1       ¦ false      ¦ BLOCKED    ¦ 1
 1   ¦ 0    ¦ 0       ¦ true       ¦ RUNNING    ¦ 6
 2   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
 3   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
 4   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4

 PID ¦ PPID ¦ WaitNum ¦ WaitParent ¦   State    ¦ Priority
-----¦------¦---------¦------------¦------------¦---------
 0   ¦ 0    ¦ 1       ¦ false      ¦ BLOCKED    ¦ 1
 1   ¦ 0    ¦ 0       ¦ true       ¦ RUNNING    ¦ 7
 2   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
 3   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
 4   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
```

When the first task is finished it's job other tasks continues execution.

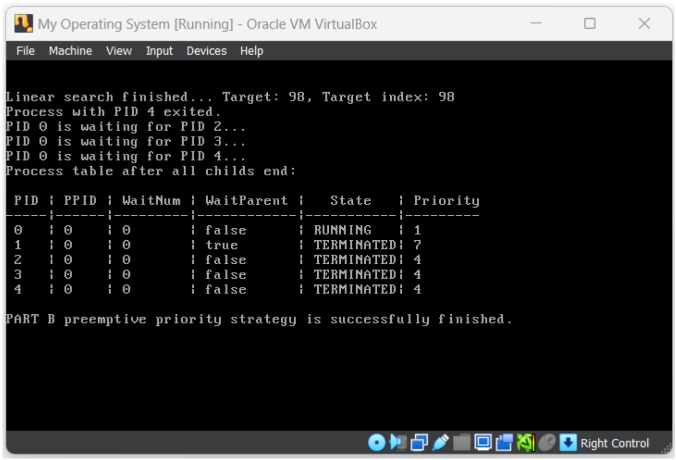
4)



```
 3   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
 4   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4

Collatz is finished.
Process with PID 1 exited.

 PID ¦ PPID ¦ WaitNum ¦ WaitParent ¦   State    ¦ Priority
-----¦------¦---------¦------------¦------------¦---------
 0   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 1
 1   ¦ 0    ¦ 0       ¦ true       ¦ TERMINATED¦ 8
 2   ¦ 0    ¦ 0       ¦ false      ¦ RUNNING    ¦ 4
 3   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
 4   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4

 PID ¦ PPID ¦ WaitNum ¦ WaitParent ¦   State    ¦ Priority
-----¦------¦---------¦------------¦------------¦---------
 0   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 1
 1   ¦ 0    ¦ 0       ¦ true       ¦ TERMINATED¦ 8
 2   ¦ 0    ¦ 0       ¦ false      ¦ RUNNING    ¦ 4
 3   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
 4   ¦ 0    ¦ 0       ¦ false      ¦ READY      ¦ 4
```

Final version of the process table.

5)



```
Linear search finished... Target: 98, Target index: 98
Process with PID 4 exited.
PID 0 is waiting for PID 2...
PID 0 is waiting for PID 3...
PID 0 is waiting for PID 4...
Process table after all childs end:

 PID ¦ PPID ¦ WaitNum ¦ WaitParent ¦   State    ¦ Priority
-----¦------¦---------¦------------¦------------¦---------
 0   ¦ 0    ¦ 0       ¦ false      ¦ RUNNING    ¦ 1
 1   ¦ 0    ¦ 0       ¦ true       ¦ TERMINATED¦ 7
 2   ¦ 0    ¦ 0       ¦ false      ¦ TERMINATED¦ 4
 3   ¦ 0    ¦ 0       ¦ false      ¦ TERMINATED¦ 4
 4   ¦ 0    ¦ 0       ¦ false      ¦ TERMINATED¦ 4

PART B preemptive priority strategy is successfully finished.
```