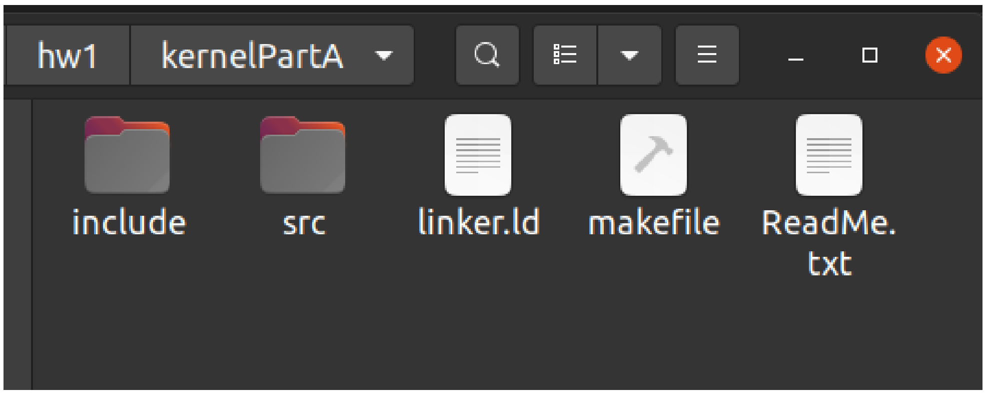
Cse312 HW1- PartA Report

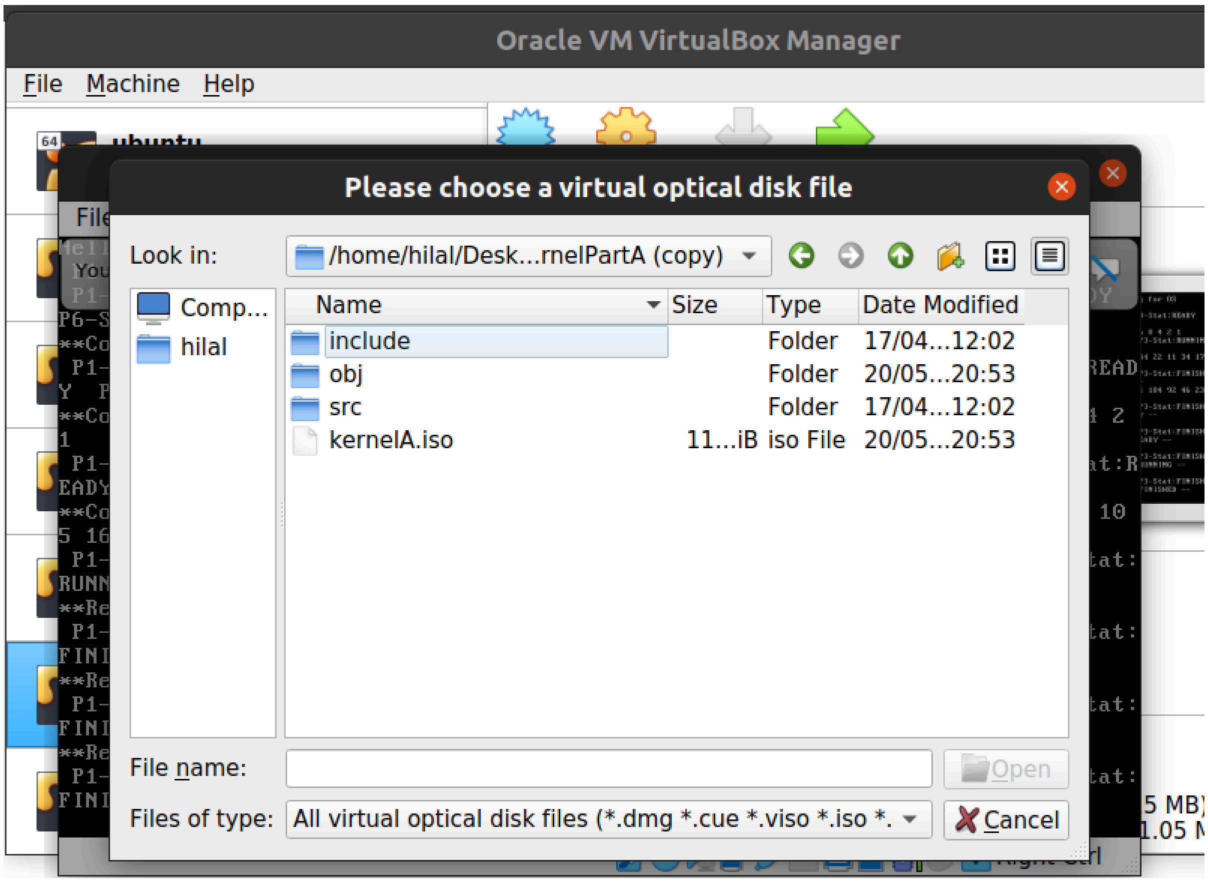
R.Hilal Saygin
200104004111

1- Install, Compilation and Run

Compile the program with ‘make run’ command in the main directory where Makefile placed. When the compilation completes ISO image kernelA.iso’ will be created alongside with the .bin file.

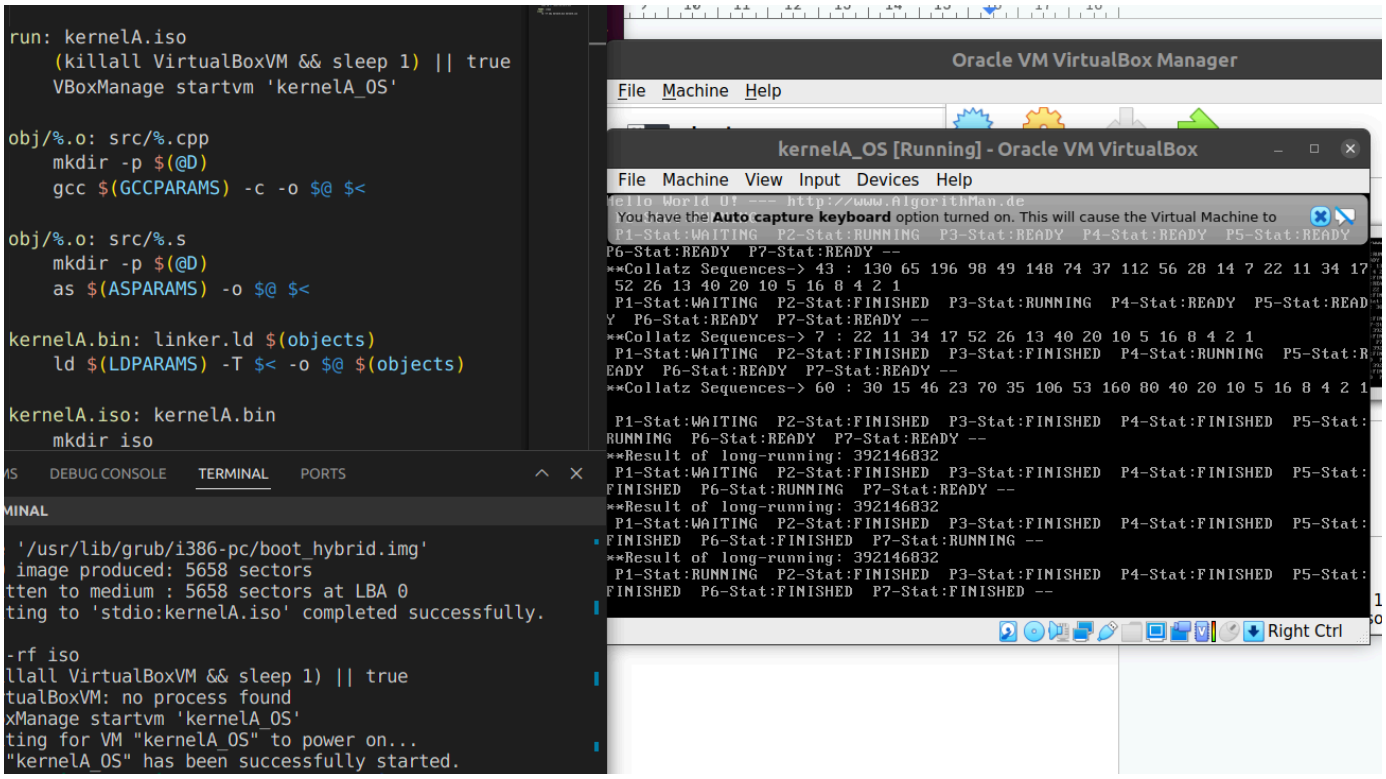


A virtual machine with the name ”`kernel_A_OS`” should be created on VirtualBox before further proceeding. Then this created “kernelA.iso” image file should be installed as a disk file onto the machine created in the Virtualbox.
Choose the iso file directory like:


Oracle VM VirtualBox Manager
File Machine Help
Please choose a virtual optical disk file
Look in:
/home/hilal/Desk...rnelPartA (copy)
Comp...
hilal
File name:
Files of type:
All virtual optical disk files (*.dmg *.cue *.viso *.iso *.
Open
Cancel

Name	Size	Type	Date Modified
include		Folder	17/04...12:02
obj		Folder	20/05...20:53
src		Folder	17/04...12:02
kernelA.iso	11...iB	iso File	20/05...20:53

After installing the disk file to the VM in the virtualBOx, go to the main directory again and now run the ‘make’ command to start up the machine installed with kernelA.iso image file. Or You can start the machine manually from the VirtualBox Manager Interface.



```
run: kernelA.iso
	(killall VirtualBoxVM && sleep 1) || true
	VBoxManage startvm 'kernelA_OS'

obj/%.o: src/%.cpp
	mkdir -p $(@D)
	gcc $(GCCPARAMS) -c -o $@ $<

obj/%.o: src/%.s
	mkdir -p $(@D)
	as $(ASPARAMS) -o $@ $<

kernelA.bin: linker.ld $(objects)
	ld $(LDPARAMS) -T $< -o $@ $(objects)

kernelA.iso: kernelA.bin
	mkdir iso
```

```
 '/usr/lib/grub/i386-pc/boot_hybrid.img'
 image produced: 5658 sectors
tten to medium : 5658 sectors at LBA 0
ting to 'stdio:kernelA.iso' completed successfully.

-rf iso
llall VirtualBoxVM && sleep 1) || true
tualBoxVM: no process found
xManage startvm 'kernelA_OS'
ting for VM "kernelA_OS" to power on...
"kernelA_OS" has been successfully started.
```

Oracle VM VirtualBox Manager
File Machine Help
kernelA_OS [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Ctrl

You have the **Auto capture keyboard** option turned on. This will cause the Virtual Machine to

```
Hello World U! --- http://www.AlgorithMan.de
 P1-Stat:WAITING  P2-Stat:RUNNING  P3-Stat:READY  P4-Stat:READY  P5-Stat:READY
P6-Stat:READY  P7-Stat:READY --
**Collatz Sequences-> 43 : 130 65 196 98 49 148 74 37 112 56 28 14 7 22 11 34 17
 52 26 13 40 20 10 5 16 8 4 2 1
 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:RUNNING  P4-Stat:READY  P5-Stat:READ
Y  P6-Stat:READY  P7-Stat:READY --
**Collatz Sequences-> 7 : 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:RUNNING  P5-Stat:R
EADY  P6-Stat:READY  P7-Stat:READY --
**Collatz Sequences-> 60 : 30 15 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1

 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:FINISHED  P5-Stat:
RUNNING  P6-Stat:READY  P7-Stat:READY --
**Result of long-running: 392146832
 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:FINISHED  P5-Stat:
FINISHED  P6-Stat:RUNNING  P7-Stat:READY --
**Result of long-running: 392146832
 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:FINISHED  P5-Stat:
FINISHED  P6-Stat:FINISHED  P7-Stat:RUNNING --
**Result of long-running: 392146832
 P1-Stat:RUNNING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:FINISHED  P5-Stat:
FINISHED  P6-Stat:FINISHED  P7-Stat:FINISHED --
```

If it is tested on ubuntu environment, 'make' will compiles, then starts vm.

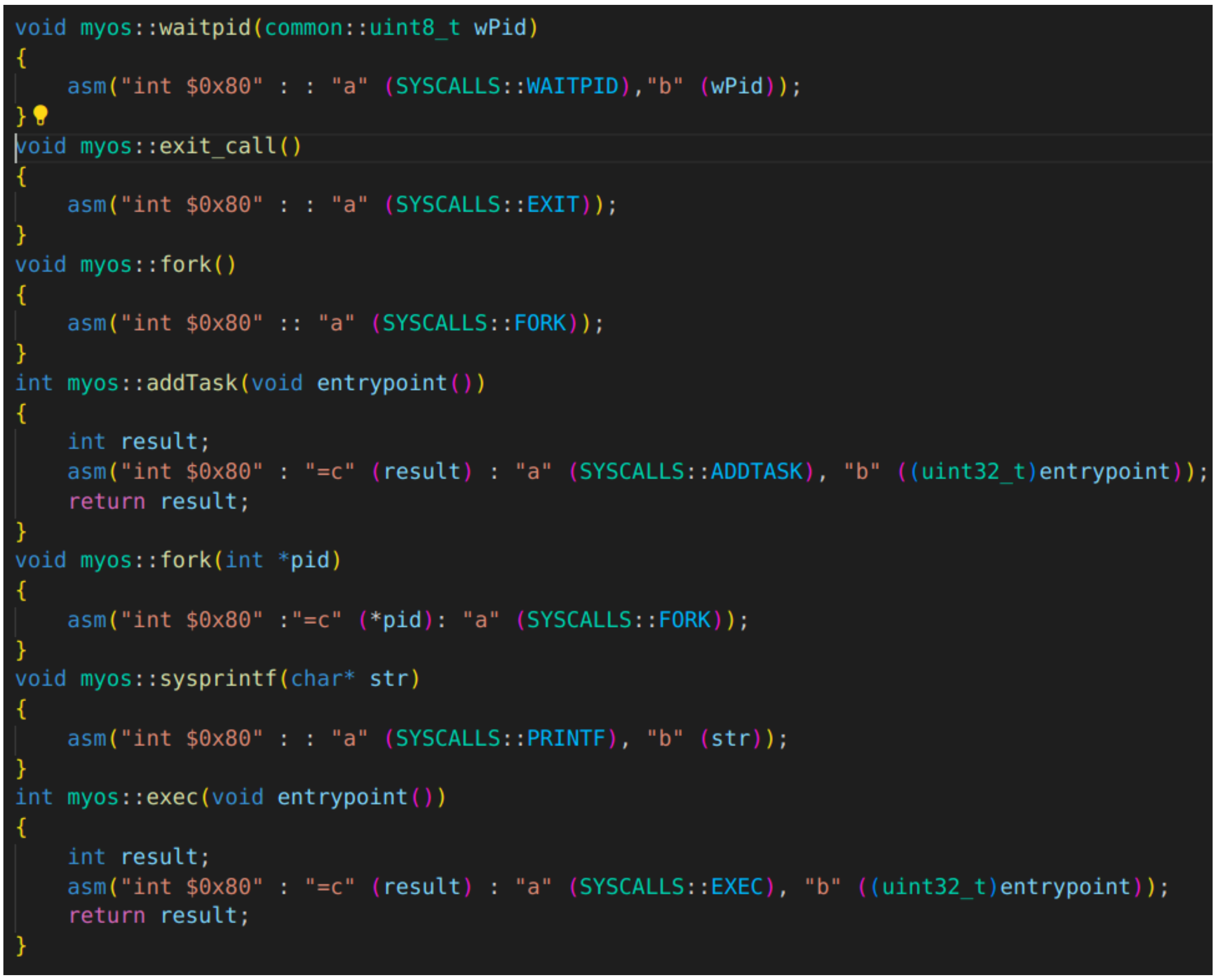
2- General Design of kernelA.iso

System Calls:

I designed the system calls to be invoked using a software interrupt (int $0x80) implemented with inline assembly asm(), and the specific call is identified using an enum value in the eax register.

Implemented System Calls

- **getPid:** Returns the process ID of the calling process.
- **waitpid:** Waits for the specified child process to change state.
- **exit_call:** Terminates the calling process.
- **sysprintf:** Outputs a string to the console.
- **fork:** Creates a copy of the calling process.
- **exec:** Replaces the process image with a new program.
- **addTask:** Adds a new task to the process table.



```
void myos::waitpid(common::uint8_t wPid)
{
    asm("int $0x80" : : "a" (SYSCALLS::WAITPID),"b" (wPid));
}
void myos::exit_call()
{
    asm("int $0x80" : : "a" (SYSCALLS::EXIT));
}
void myos::fork()
{
    asm("int $0x80" :: "a" (SYSCALLS::FORK));
}
int myos::addTask(void entrypoint())
{
    int result;
    asm("int $0x80" : "=c" (result) : "a" (SYSCALLS::ADDTASK), "b" ((uint32_t)entrypoint));
    return result;
}
void myos::fork(int *pid)
{
    asm("int $0x80" :"=c" (*pid): "a" (SYSCALLS::FORK));
}
void myos::sysprintf(char* str)
{
    asm("int $0x80" : : "a" (SYSCALLS::PRINTF), "b" (str));
}
int myos::exec(void entrypoint())
{
    int result;
    asm("int $0x80" : "=c" (result) : "a" (SYSCALLS::EXEC), "b" ((uint32_t)entrypoint));
    return result;
}
```

Process Management

Processes are managed using a class named TASK that stores information about each process. The Task objects contains the process ID, state, memory address, CPU registers, and other metadata.
The process table is an array of Task objects. The kernel uses the process table to keep track of all active processes in the system.
Process States: Processes can be in one of several states:
TaskStates {READY, WAITING, FINISHED};
 It supports task creation through direct addition and forking, employs a round-robin scheduling algorithm for fair CPU time distribution, handles context switching to allow tasks to pause and resume, and manages task termination.

Interrupt Handling

The kernel handles hardware and software interrupts. The software interrupt *0x80* is used for system calls. The interrupt handler saves the CPU state, determines the system call to execute, and dispatches the call to the appropriate handler function.

Multitasking

Multitasking is achieved through context switching, where the CPU state of the current process is saved, and the state of the next process is restored. The **Round Robin scheduling** algorithm is used to determine the next process to run.

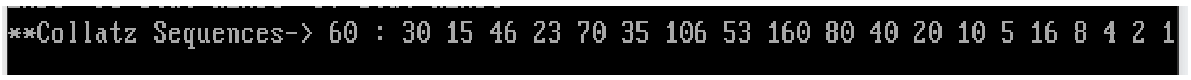
Context Switching

During a timer interrupt, the kernel performs a context switch by:
Saving the current process state.
Selecting the next process from the process table using implemented Robin-scheduler algorithm. The details of the function will be discussed in further Code explanation title of the report.
Then the state of the selected process restored.

Testing Programs in Part A

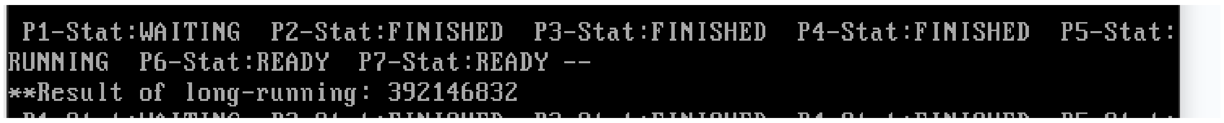
Two test programs are implemented to validate the OS functionality:

Collatz Conjecture: Calculates the Collatz sequence for numbers less than 100.
The expected results of this task is as:



```
**Collatz Sequences-> 60 : 30 15 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
```

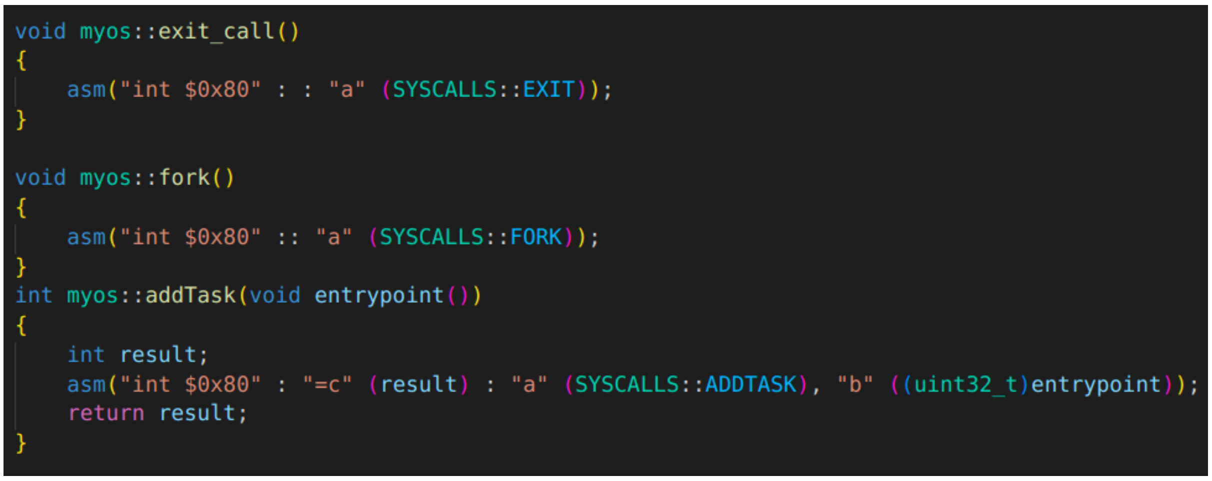
Long Running Program: Performs a computationally task described as in the pdf.
The output of this task are printed on the terminal of OS as:



```
 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:FINISHED  P5-Stat:
RUNNING  P6-Stat:READY  P7-Stat:READY --
**Result of long-running: 392146832
```

3- Code Explanations

Syscalls triggers the interrupt 0x80 to switch to the kernel mode, which the SyscallHandler handles, simulating a system call for exiting a process.

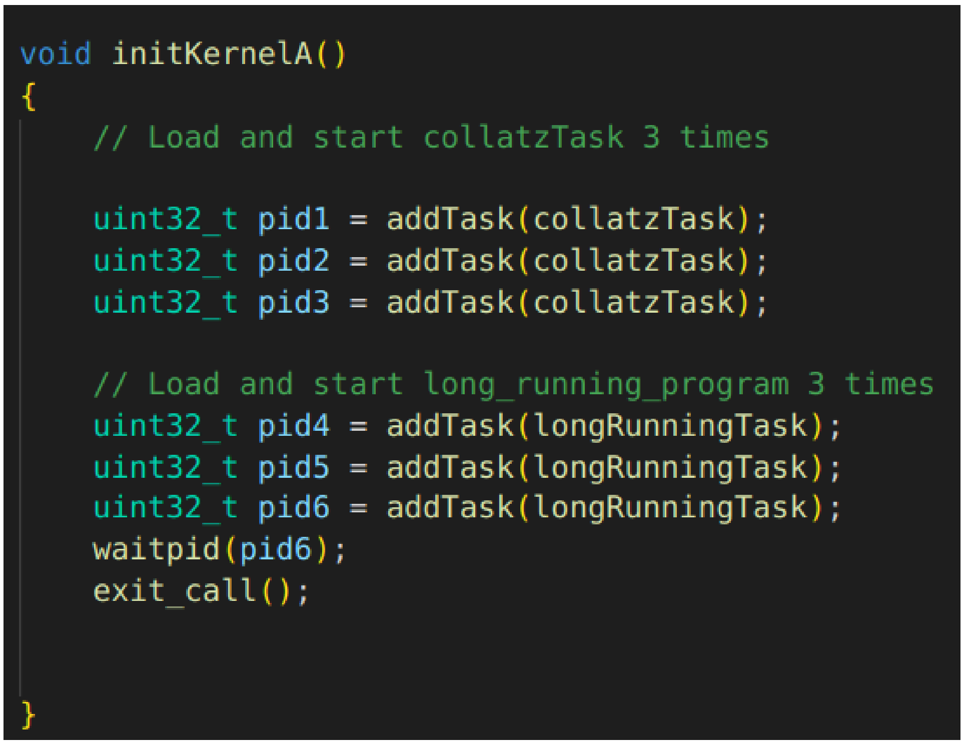


```
void myos::exit_call()
{
    asm("int $0x80" : : "a" (SYSCALLS::EXIT));
}

void myos::fork()
{
    asm("int $0x80" :: "a" (SYSCALLS::FORK));
}
int myos::addTask(void entrypoint())
{
    int result;
    asm("int $0x80" : "=c" (result) : "a" (SYSCALLS::ADDTASK), "b" ((uint32_t)entrypoint));
    return result;
}
```

Handling Multi-Programming and Managing Process Table

Loading Multiple processes into kernel:

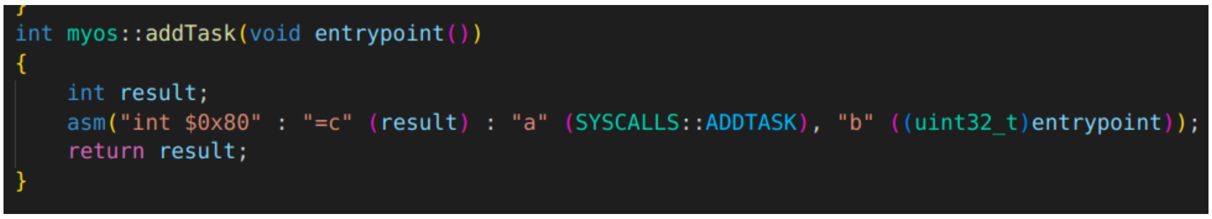
The kernel A functionality implemented in the initkernelA function.
addTask system call funtion is used to load tasks (programs) into memory and prepare them for execution. It adds new processes into the OS by invoking system call interrupt signal..



```
void initKernelA()
{
    // Load and start collatzTask 3 times

    uint32_t pid1 = addTask(collatzTask);
    uint32_t pid2 = addTask(collatzTask);
    uint32_t pid3 = addTask(collatzTask);

    // Load and start long_running_program 3 times
    uint32_t pid4 = addTask(longRunningTask);
    uint32_t pid5 = addTask(longRunningTask);
    uint32_t pid6 = addTask(longRunningTask);
    waitpid(pid6);
    exit_call();


}
```

myos::addTask is used to add a new task to the operating system's task manager. It achieves this by invoking a system call.

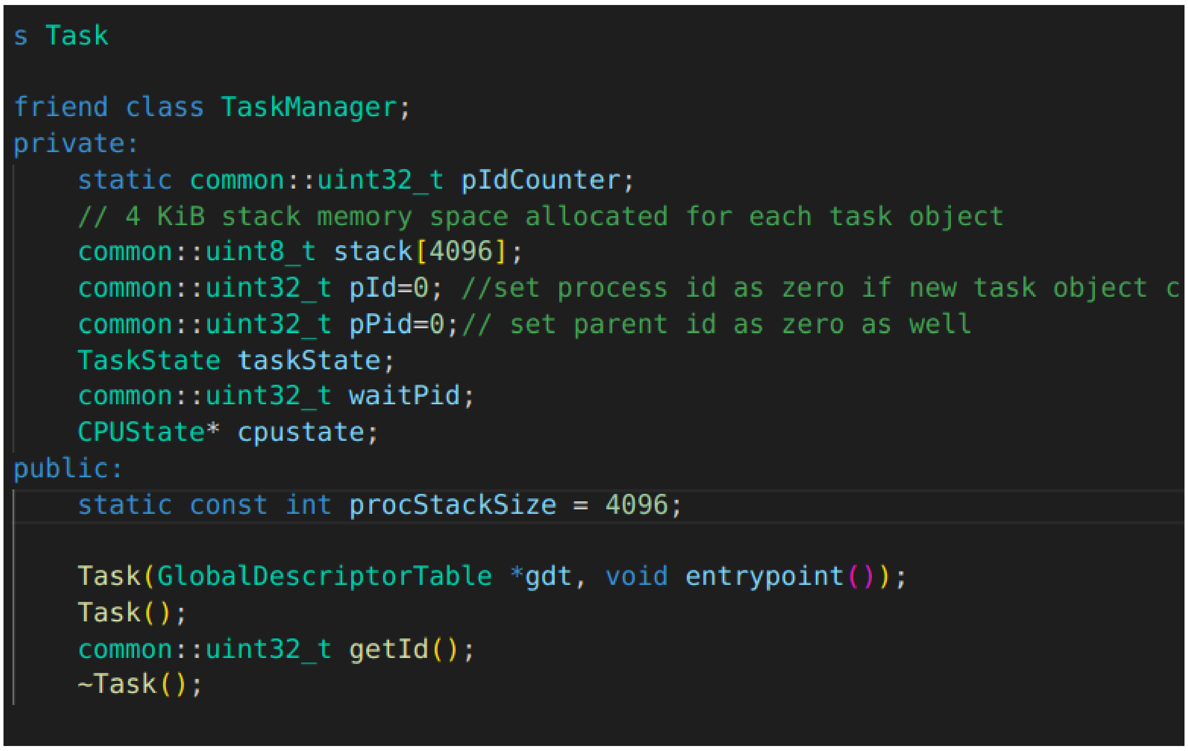
It takes a function pointer entrypoint which points to the entry point of the task that needs to be added. It returns an integer which is typically the process ID (PID) of the newly created task. It uses inline assembly to perform a system call.
"int $0x80": This is the assembly instruction to trigger a software interrupt with the interrupt vector 0x80. Passes the task's entry point to the kernel, which then creates the new task and returns its PID.



```
int myos::addTask(void entrypoint())
{
    int result;
    asm("int $0x80" : "=c" (result) : "a" (SYSCALLS::ADDTASK), "b" ((uint32_t)entrypoint));
    return result;
}
```

Task Class:

The Task class represents an individual task or process in the system. It includes the task's state, its stack, and its context. By adding tasks to the TaskManager, the kernel can switch between them, implementing multi-programming.

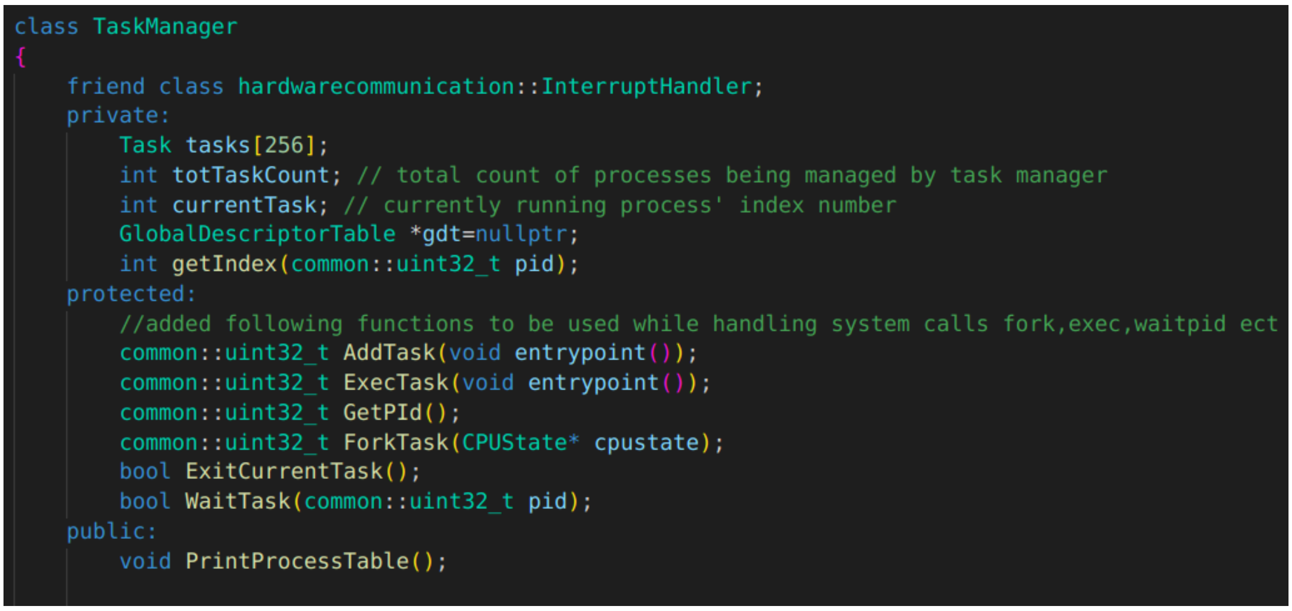


```
s Task

friend class TaskManager;
private:
    static common::uint32_t pIdCounter;
    // 4 KiB stack memory space allocated for each task object
    common::uint8_t stack[4096];
    common::uint32_t pId=0; //set process id as zero if new task object c
    common::uint32_t pPid=0;// set parent id as zero as well
    TaskState taskState;
    common::uint32_t waitPid;
    CPUState* cpustate;
public:
    static const int procStackSize = 4096;

    Task(GlobalDescriptorTable *gdt, void entrypoint());
    Task();
    common::uint32_t getId();
    ~Task();
```

TaskManager class:

It is responsible for managing multiple tasks. It holds the process table and manages task scheduling and switching

Proces Table management: processes currently running on the programs are being hold in the ***tasks*** array attribute of the TaskManager class. Each Task object contains all the necessary information about a process, including its state, stack, CPU state, and process IDs (PID and PPID).



```
class TaskManager
{
    friend class hardwarecommunication::InterruptHandler;
    private:
        Task tasks[256];
        int totTaskCount; // total count of processes being managed by task manager
        int currentTask; // currently running process' index number
        GlobalDescriptorTable *gdt=nullptr;
        int getIndex(common::uint32_t pid);
    protected:
        //added following functions to be used while handling system calls fork,exec,waitpid ect
        common::uint32_t AddTask(void entrypoint());
        common::uint32_t ExecTask(void entrypoint());
        common::uint32_t GetPId();
        common::uint32_t ForkTask(CPUState* cpustate);
        bool ExitCurrentTask();
        bool WaitTask(common::uint32_t pid);
    public:
        void PrintProcessTable();
```

- Attributes in TaskManager:

Task tasks[]: aAn array that holds up to 256 Task objects. Each entry in this array represents a single process in the system represented by Task class object.
int totTaskCount: keeps track of the total number of tasks currently managed by the TaskManager.
int currentTask: represents the index of the currently running task within the tasks array.

- Methods in TaskManager:
  AddTask(void entrypoint()): method adds a new task to the process table by initialising a Task object with the given entry point and assigning it a unique PID.
  ExecTask(void entrypoint()): method creates and executes a new task, similar to AddTask.

  ***ForkTask(CPUState cpustate)**:**

  This method duplicates the currently running task to create a new task. The implementa<tion of Operating system call in many operation systems like unix.
  It copies the relevant information, CPU state and stack of the current task to the new task, of currently running process which is the parent process for the task that is being created.
  - Static Process ID Counter: Uses a static variable pIdCounter to keep track of the last assigned process ID, ensuring each new task gets a unique ID.
  - Task Structure: Each task is represented by an instance of the Task structure, which includes fields for the task's state, process ID, parent process ID, CPU state, and more.
  - Memory Management: Directly manipulates memory addresses and sizes to manage stacks and CPU states, reflecting low-level control typical in operating system kernels.
  - Error Handling: Returns 0 if the maximum number of tasks is reached, providing a simple error indication mechanism.

ExitCurrentTask(): handles the termination of the current task. It updates the task state and removes the task from the scheduling rotation.

WaitTask(common::uint32_t pid): makes the current task wait for another task with the specified PID to finish execution.

PrintProcessTable(): prints the details of all tasks in the process table for debugging and monitoring purposes. For the current kernel after the test case completed, it prints all the info in the process table.

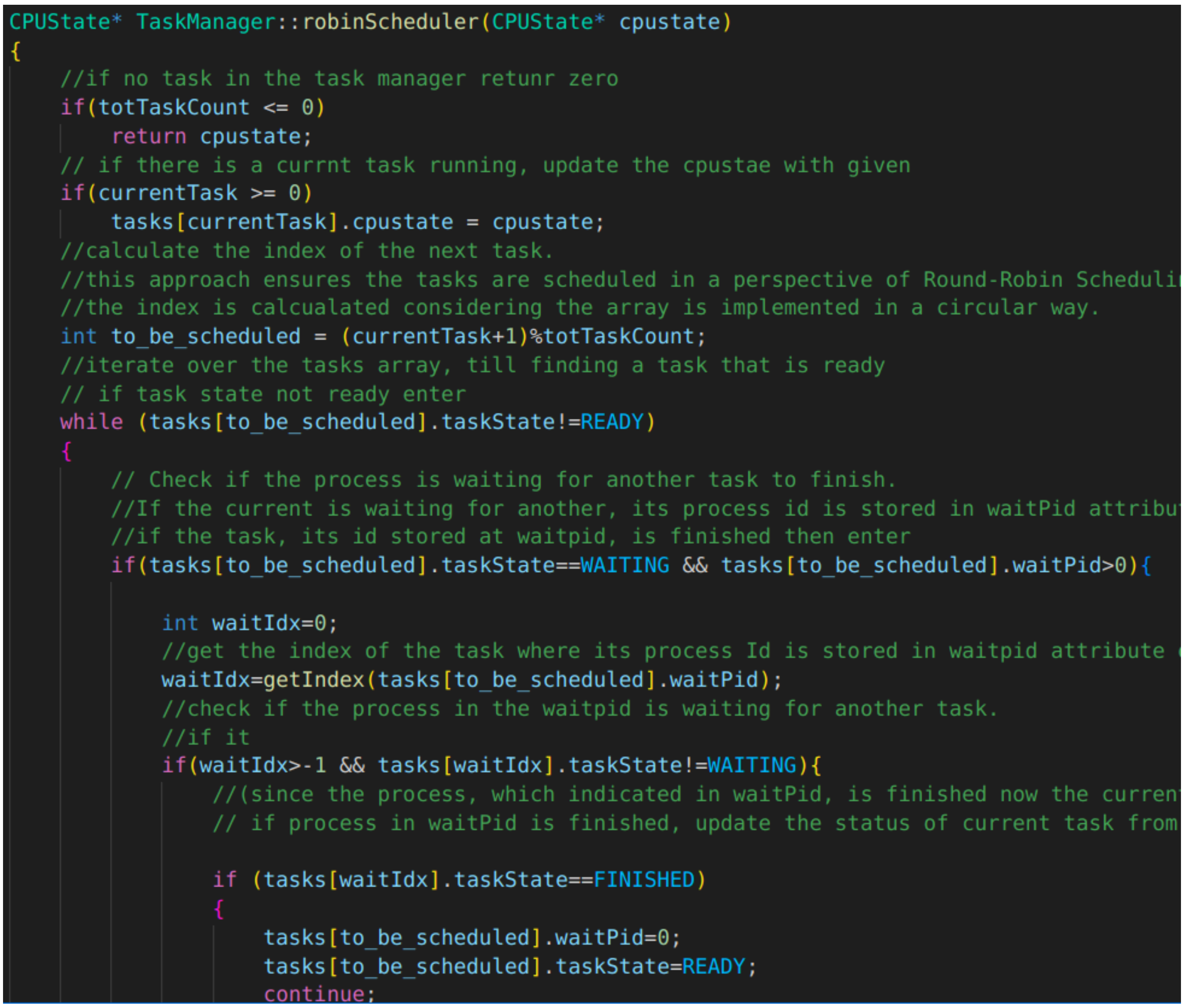
Scheduler:

Implemented using a round-robin scheduling algorithm. It handles task switching by saving the state of the current task, updating the task index, and restoring the state of the next task.
It is implemented as: **CPUState robinScheduler(CPUState cpustate)**:**
It is called during each timer interrupt to switch between tasks.
Implementation Details

- Check for Existing Tasks: First, it checks if there are any tasks currently managed by the TaskManager. If there are no tasks (totTaskCount <= 0), it simply returns the current CPU state without making any changes, indicating that there's nothing to schedule.
- Save Current Task State: If there is a currently running task (currentTask >= 0), it saves the current CPU state back into the corresponding task's cpustate. This is crucial for preserving the execution context of the task so it can resume later.
- Determine Next Task: It calculates the index of the next task to be scheduled. This calculation considers the current task index and wraps around to the beginning of the task list if necessary, ensuring a cyclic rotation through all tasks. This is the essence of round-robin scheduling.
- Find Ready Task: Starting from the calculated index, it iterates through the task list until it finds a task whose state is READY. If a task is found in the WAITING state and it's waiting for another task to complete (its waitPid is non-zero), it further checks if the task it's waiting for has completed. If so, it updates the waiting task's state to READY and continues the search from this point.
- Handle Waiting Tasks: If a task is waiting for another task that is still running or waiting itself, it skips this task and moves to the next one in the list. This prevents deadlocks and ensures that only tasks that are truly ready to run are considered.
- Schedule Found Task: Once a ready task is found, it sets this task as the current task and updates the currentTask index accordingly. Then prints the process table to provide visibility into the current state of all tasks currently in the process table.
- Return Scheduled Task's cpuState: it returns the CPU state of the scheduled task, which should be loaded into the CPU registers to resume execution of the task.

- Circular Queue Behavior: treats the task list as a circular queue, ensuring that tasks are considered for execution in a round-robin fashion.
  This functionality achieved by using modular arithmetic (% totTaskCount) when determining the next task index.

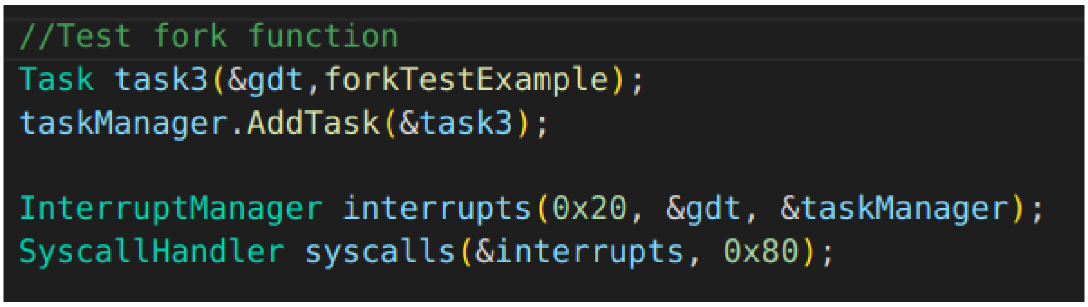
- Fairness and Preemption: By saving the current task's state before selecting the next task, and by cycling through all tasks regardless of their priority, the scheduler ensures fairness and preemption. Therefore, every task gets an equal chance to make progress.
- Efficiency: Quickly moving through tasks to find the next one to run. It avoids unnecessary work by directly manipulating task states and indices.



```cpp
CPUState* TaskManager::robinScheduler(CPUState* cpustate)
{
    //if no task in the task manager retunr zero
    if(totTaskCount <= 0)
        return cpustate;
    // if there is a currnt task running, update the cpustae with given
    if(currentTask >= 0)
        tasks[currentTask].cpustate = cpustate;
    //calculate the index of the next task.
    //this approach ensures the tasks are scheduled in a perspective of Round-Robin Schedulin
    //the index is calcualted considering the array is implemented in a circular way.
    int to_be_scheduled = (currentTask+1)%totTaskCount;
    //iterate over the tasks array, till finding a task that is ready
    // if task state not ready enter
    while (tasks[to_be_scheduled].taskState!=READY)
    {
        // Check if the process is waiting for another task to finish.
        //If the current is waiting for another, its process id is stored in waitPid attribu
        //if the task, its id stored at waitpid, is finished then enter
        if(tasks[to_be_scheduled].taskState==WAITING && tasks[to_be_scheduled].waitPid>0){

            int waitIdx=0;
            //get the index of the task where its process Id is stored in waitpid attribute
            waitIdx=getIndex(tasks[to_be_scheduled].waitPid);
            //check if the process in the waitpid is waiting for another task.
            //if it
            if(waitIdx>-1 && tasks[waitIdx].taskState!=WAITING){
                //(since the process, which indicated in waitPid, is finished now the curren
                // if process in waitPid is finished, update the status of current task from

                if (tasks[waitIdx].taskState==FINISHED)
                {
                    tasks[to_be_scheduled].waitPid=0;
                    tasks[to_be_scheduled].taskState=READY;
                    continue;
```

Handling Interrupts

The kernel demonstrates interrupt handling through the InterruptManager:
Interrupt manager being set up in the main kernel function after task objects are created.



```
//Test fork function
Task task3(&gdt,forkTestExample);
taskManager.AddTask(&task3);

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

InterruptManager

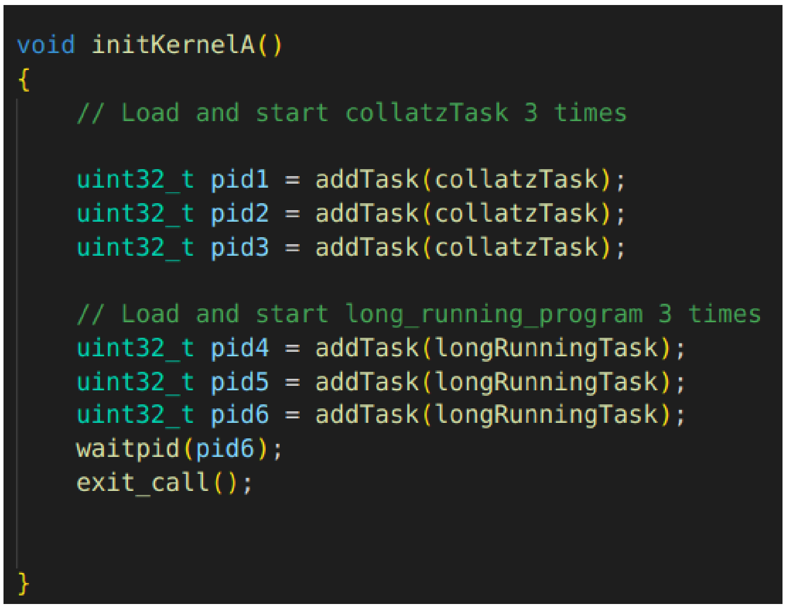
I defined this class defined for setting up and managing interrupts within the system. Constructor takes two arguments: an interrupt number and pointers to a GlobalDescriptorTable and a TaskManager.
When kernel started, an instance of InterruptManager is created, initializing it with the interrupt number 0x20.

Syscall Handler

I created this class is initialized with references to the InterruptManager and an interrupt number (0x80). This is assigned for software interrupts (interrupt 0x80) to handle system calls made by user-mode applications.

Execution Flow

The kernelMain function initializes critical components he GDT, task manager, and interrupt manager.
It adds tasks to the task manager, which include examples of process creation, execution, and specific operations. For the test cases longRunningTask, collatzTask are added in the kernel execution function *initkernalA.* In the intKernelA request scenario is implemented.



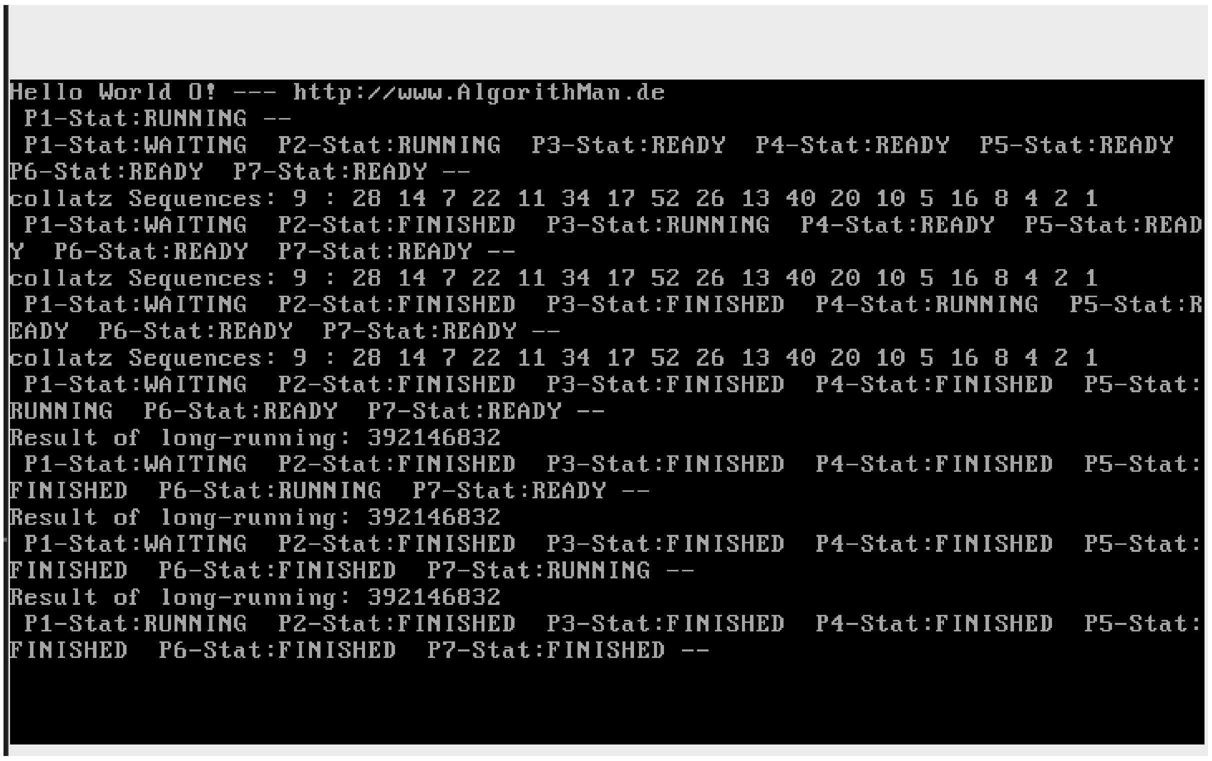
```
void initKernelA()
{
    // Load and start collatzTask 3 times

    uint32_t pid1 = addTask(collatzTask);
    uint32_t pid2 = addTask(collatzTask);
    uint32_t pid3 = addTask(collatzTask);

    // Load and start long_running_program 3 times
    uint32_t pid4 = addTask(longRunningTask);
    uint32_t pid5 = addTask(longRunningTask);
    uint32_t pid6 = addTask(longRunningTask);
    waitpid(pid6);
    exit_call();


}
```

The interrupt manager is activated, enabling the handling of hardware interrupts.

The system enters an infinite loop, where it waits for interrupts and handles them accordingly, allowing tasks to run and interact with the system.

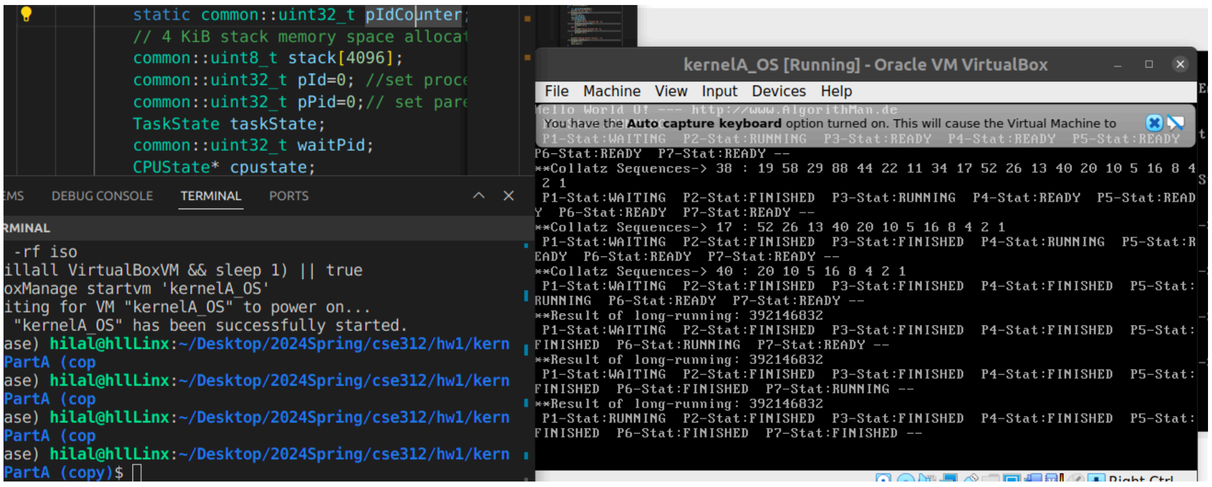
4- Test Cases and Results

Print he processes meta data stored in the process table for observation of multi-process handling.



```
Hello World O! --- http://www.AlgorithMan.de
 P1-Stat:RUNNING --
 P1-Stat:WAITING  P2-Stat:RUNNING  P3-Stat:READY  P4-Stat:READY  P5-Stat:READY
P6-Stat:READY  P7-Stat:READY --
collatz Sequences: 9 : 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:RUNNING  P4-Stat:READY  P5-Stat:READ
Y  P6-Stat:READY  P7-Stat:READY --
collatz Sequences: 9 : 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:RUNNING  P5-Stat:R
EADY  P6-Stat:READY  P7-Stat:READY --
collatz Sequences: 9 : 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:FINISHED  P5-Stat:
RUNNING  P6-Stat:READY  P7-Stat:READY --
Result of long-running: 392146832
 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:FINISHED  P5-Stat:
FINISHED  P6-Stat:RUNNING  P7-Stat:READY --
Result of long-running: 392146832
 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:FINISHED  P5-Stat:
FINISHED  P6-Stat:FINISHED  P7-Stat:RUNNING --
Result of long-running: 392146832
 P1-Stat:RUNNING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:FINISHED  P5-Stat:
FINISHED  P6-Stat:FINISHED  P7-Stat:FINISHED --
```

Tes result of kernelA for Collatz and Long running program,
The results of the calculations and the processes data are displayed for this execution.



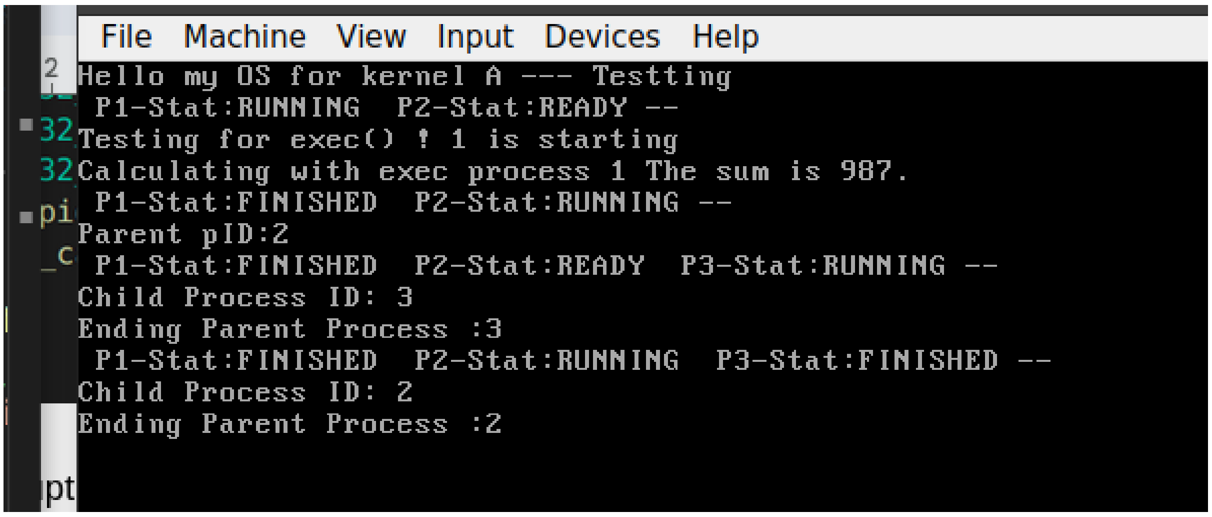
```
static common::uint32_t pIdCounter;
// 4 KiB stack memory space allocat
common::uint8_t stack[4096];
common::uint32_t pId=0; //set proce
common::uint32_t pPid=0;// set pare
TaskState taskState;
common::uint32_t waitPid;
CPUState* cpustate;
```

```
 -rf iso
illall VirtualBoxVM && sleep 1) || true
oxManage startvm 'kernelA_OS'
iting for VM "kernelA_OS" to power on...
 "kernelA_OS" has been successfully started.
ase) hilal@hllLinx:~/Desktop/2024Spring/cse312/hw1/kern
PartA (cop
ase) hilal@hllLinx:~/Desktop/2024Spring/cse312/hw1/kern
PartA (cop
ase) hilal@hllLinx:~/Desktop/2024Spring/cse312/hw1/kern
PartA (cop
ase) hilal@hllLinx:~/Desktop/2024Spring/cse312/hw1/kern
PartA (copy)$
```

You have the **Auto capture keyboard** option turned on. This will cause the Virtual Machine to

```
Hello World O! --- http://www.AlgorithMan.de
 P1-Stat:WAITING  P2-Stat:RUNNING  P3-Stat:READY  P4-Stat:READY  P5-Stat:READY
P6-Stat:READY  P7-Stat:READY --
**Collatz Sequences-> 38 : 19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4
 2 1
 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:RUNNING  P4-Stat:READY  P5-Stat:READ
Y  P6-Stat:READY  P7-Stat:READY --
**Collatz Sequences-> 17 : 52 26 13 40 20 10 5 16 8 4 2 1
 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:RUNNING  P5-Stat:R
EADY  P6-Stat:READY  P7-Stat:READY --
**Collatz Sequences-> 40 : 20 10 5 16 8 4 2 1
 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:FINISHED  P5-Stat:
RUNNING  P6-Stat:READY  P7-Stat:READY --
**Result of long-running: 392146832
 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:FINISHED  P5-Stat:
FINISHED  P6-Stat:RUNNING  P7-Stat:READY --
**Result of long-running: 392146832
 P1-Stat:WAITING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:FINISHED  P5-Stat:
FINISHED  P6-Stat:FINISHED  P7-Stat:RUNNING --
**Result of long-running: 392146832
 P1-Stat:RUNNING  P2-Stat:FINISHED  P3-Stat:FINISHED  P4-Stat:FINISHED  P5-Stat:
FINISHED  P6-Stat:FINISHED  P7-Stat:FINISHED --
```

Test result of exec task and fork task system calls are displayed alongside with the process table.



```
Hello my OS for kernel A --- Testting
 P1-Stat:RUNNING  P2-Stat:READY --
Testing for exec() ! 1 is starting
Calculating with exec process 1 The sum is 987.
 P1-Stat:FINISHED  P2-Stat:RUNNING --
Parent pID:2
 P1-Stat:FINISHED  P2-Stat:READY  P3-Stat:RUNNING --
Child Process ID: 3
Ending Parent Process :3
 P1-Stat:FINISHED  P2-Stat:RUNNING  P3-Stat:FINISHED --
Child Process ID: 2
Ending Parent Process :2
```