

GEBZE TECHNICAL UNIVERSITY  
CSE312 – HW1 DOCUMENTATION

Student Name: Emre Oytun  
Student Number: 200104004099

## 1) DESIGN DECISIONS and STRUCTURES:

### 1.a) Multitasking:

Priority – State – Constants (include/multitasking.h) :

```
namespace myos
{
    const common::uint32_t MAX_STACK_SIZE = 4096;
    const common::uint32_t MAX_NUM_TASKS = 256;

    typedef enum { High, Medium, Low } Priority; // The highest priority has the minimum value
    typedef enum { Ready, Running, Blocked, Terminated } State;
```

**MAX\_STACK\_SIZE:** This constant is for specifying a limited number for stack size of each processes.

**MAX\_NUM\_TASKS:** This constant is for specifying a limited number for max num of tasks since we keep a task array.

**Priority Enum:** This enum is for keeping the priorities of each processes. The high priority has the lowest number (0), and low priority has the highest number (2).

**State Enum:** This enum is for keeping the states of each processes.

CPUState Class (include/multitasking.h) :

```
class Task;

struct CPUState
{
    /* Pushed by interruptstubs.s */
    common::uint32_t eax;
    common::uint32_t ebx; // base register
    common::uint32_t ecx; // counting register
    common::uint32_t edx; // data register

    common::uint32_t esi; // stack index
    common::uint32_t edi; // data index
    common::uint32_t ebp; // stack base pointer

    /*
    ///////////////////////////////////
    common::uint32_t gs;
    common::uint32_t fs;
    common::uint32_t es;
    common::uint32_t ds;
    ///////////////////////////////////
    */

    common::uint32_t error; // for error code

    /* Pushed by processor */
    common::uint32_t eip; // instruction pointer
    common::uint32_t cs; // code segment
    common::uint32_t eflags; // flags
    common::uint32_t esp; // stack pointer s
    common::uint32_t ss; // stack segment
} __attribute__((packed));
```

This struct is for keeping the registers of a process. This is necessary because we need to save the registers of a process in context switch and restore them when it starts working again to keep track of the current state of the process.

The important fields:

**eax**: Keeps the return value when a function is called.

**ebx-ecx**: General purpose registers. Ecx is generally used for counting like in loops.

**ebx**: Stack base pointer which points to the current stack.

**eip**: Instruction pointer which keeps track of the instructions.

**cs**: Code segment

**ss**: Stack segment

Task Class: (include/multitasking.h)

```
class Task
{
    friend class TaskManager;
private:
    common::uint8_t stack[MAX_STACK_SIZE]; // 4 KiB
    CPUState* cpustate;

    common::int32_t pid;
    common::int32_t ppid;

    Priority priority;
    State state;

    bool waitingChild; // Indicates if this task is waiting any child
    int waitingChildId; // Indicates which child this task is waiting (-1 or child pid)

    bool parentTookInWait; // If parent has taken this in waitpid already

    //common::int32_t forkPid;
    common::int32_t arrivalOrder;

public:
    common::int32_t forkPid;

    Task();
    Task(GlobalDescriptorTable *gdt, void entrypoint());
    Task(const Task& task);
    ~Task();
    void Copy(const Task* oth);
    void CopyCpuState(CPUState* cpustate);
    void Reset(GlobalDescriptorTable* gdt, void entrypoint());

    CPUState* GetCPUState();
    void SetCPUState(CPUState* cpustate);

    Priority GetPriority();
    void SetPriority(Priority priority);

    common::int32_t GetPid();
    void SetPid(common::int32_t pid);

    common::int32_t GetPPid();
    void SetPPid(common::int32_t ppid);

    State GetState();
    void SetState(State state);

    bool GetParentTookInWait();
    void SetParentTookInWait(bool parentTookInWait);

    common::int32_t GetArrivalOrder();
    void SetArrivalOrder(common::int32_t arrivalOrder);
```

In this class, I keep all of the necessary information for a single process. You can find the descriptions for each field below:

### Task Class Fields:

**stack:** This array keeps the stack of the process. It is filled up as variables added in the process. It also contains the CPUState data. It has MAX\_STACK\_SIZE capacity which is 4KB.

**cpustate:** This keeps a reference to the cpustate of the process. This reference is an address inside the stack of the process. It keeps the registers of the process inside such as stack pointer, instruction pointer, stack segment, code segment etc.

**pid:** This keeps the process id which is assigned when the process is initialized. It is actually the index of tasks array in the TaskManager.

**ppid:** This keeps the parent process id which is assigned in the fork and used for some purposes such as waitpid.

**priority:** This keeps the priority of the task.

**state:** This keeps the state of the process such as terminated, ready, running and blocked.

**waitingChild:** As in the comment, this indicates if this process is waiting for any child. This is necessary for implementing waitpid since we should know if this process is waiting for any child.

**waitingChildId:** As in the comment, this indicates the child pid which this process is waiting for. This is necessary for implementing waitpid since we should know which child this process is waiting for.

**parentTookInWait:** As in the comment, this indicates if the parent has reaped up this child process or not. This is necessary for implementing waitpid since we should know if parent has already taken this process.

**arrivalOrder:** This indicates the arrival order of the process among all process. This information is necessary to make decision in preemptive priority based scheduling.

### Task Class Important Methods:

#### Reset:

```
void Task::Reset(GlobalDescriptorTable* gdt, void entrypoint()) {  
    // Allocate stack memory for the CPUState of this task.  
    cpustate = (CPUState*)(stack + MAX_STACK_SIZE - sizeof(CPUState));  
  
    // Initialize register of this task/process.  
    cpustate->eax = 0;  
    cpustate->ebx = 0;  
    cpustate->ecx = 0;  
    cpustate->edx = 0;  
  
    cpustate->esi = 0;  
    cpustate->edi = 0;  
    cpustate->ebp = 0;  
  
    cpustate->eip = (uint32_t)entrypoint;  
    cpustate->cs = gdt->CodeSegmentSelector();  
    cpustate->eflags = 0x202;  
}
```

This method resets the process's registers and sets the eip to given entrypoint. When this process is scheduled, it will start running from entrypoint.

Copy:

```
void Task::Copy(const Task* oth)
{
    // task.stack + 4096 - task.cpushate = this->stack + 4096 - this->cpustate
    this->cpustate = (CPUState*) (this->stack - (oth->stack - (uint8_t*) oth->cpustate));

    for (int i = 0; i < MAX_STACK_SIZE; ++i) {
        this->stack[i] = oth->stack[i];
    }
}
```

This method copies the given process. It actually does what fork does. It copies the stack from the given process to this process's stack. It also sets the cpustate to the proper position according to the other process's relative position of cpustate to its stack. It finds the difference of other task's stack and other task's cpustate then it subtracts it from this stack. As a result, cpustate is set to the proper location.

TaskManager Class:

```
class TaskManager
{
private:
    SchedulerType schedulerType;
    LifeCycleType lifeCycleType;
    ProcessTablePrintType processTablePrintType;

    Task tasks[MAX_NUM_TASKS];
    int currentTask;
    int numTasks;
    int nextArrivalOrder;

    int interruptNumAfterCollatz;
    Task* collatzTask;
    Task* blockedTaskForCollatz;

    Task* readyQueue[MAX_NUM_TASKS];
    int queueLen;

    bool ignoreSchedule;
    bool useDelayInPrintingProcessTable;

    GlobalDescriptorTable *gdt;

    void PrintProcessInfo(Task* task);
    void PrintProcessTable();

    CPUState* RoundRobinSchedule();
    CPUState* PreemptivePrioritySchedule();

    void AddToReadyQueue(Task* task);
    void AddToReadyQueueRoundRobin(Task* task);
    void AddToReadyQueuePreemptivePriority(Task* task);
    Task* PopFromReadyQueue();

public:
    TaskManager(GlobalDescriptorTable* gdt, SchedulerType schedulerType, LifeCycleType lifeCycleType, ProcessTablePrintType processTablePrintType, bool useDelayInPrintingProcessTable);
    ~TaskManager();
    Task* AddTask(Task* newTask, Priority priority, common::int32_t ppid);
    void CollatzAdded();
    CPUState* Schedule(CPUState* cpustate);
    CPUState* Schedule();
}
```

```

void Fork(CPUState* cpustate);
common::uint32_t Execve(void (*entrypoint)());
common::uint32_t Waitpid(common::uint32_t pid, CPUState* cpustate);
common::uint32_t Exit();
common::uint32_t BlockForCollatz(CPUState* cpustate);
void RemoveFromReadyQueue(int pid);

Task* GetCurrentTask();

void SetIgnoreSchedule(bool ignoreSchedule);

void SetLastTaskPriority(common::uint32_t priority);
};

```

### TaskManager Class Fields:

**schedulerType:** Indicates the scheduler type as the name says. It schedules the processes accordingly.

**lifeCycleType:** Indicates for which lifecycle this kernel is working now. It is a parameter to work for the lifecycles accordingly.

**processTablePrintType:** It is a parameter for printing the process table. According to that, table is printed either only in context switch, in process termination, in every time interrupt, or is not printed at all.

**tasks:** It is an array of processes. It keeps the active processes inside. We keep it statically here to allocate memory for the processes.

**currentTask:** It keeps the index of the current task.

**numTasks:** It keeps total number of tasks.

**nextArrivalOrder:** It keeps the next arrival number for the incoming process to keep track of the arrival order.

**interruptNumAfterCollatz:** It keeps interrupt number after collatz to implement lifecycles properly, especially for part-B 3<sup>rd</sup> and 4<sup>th</sup> strategies.

**collatzTask:** It keeps the collatz tasks's pointer if any, especially for part-B 3<sup>rd</sup> and 4<sup>th</sup> strategies.

**blockedTaskForCollatz:** It keeps the init task pointer if it is blocked for collatz, especially for part-B 3<sup>rd</sup> and 4<sup>th</sup> strategies.

**readyQueue:** It keeps queue of processes for scheduling. It is an ordered queue according to the scheduling strategy.

**queueLen:** It keeps the ready queue length.

**ignoreSchedule:** Boolean variable indicating if we need to ignore time interrupt to schedule. It is used in 4<sup>th</sup> lifecycle to block scheduling before all processes are ready.



**gseDelayInPrintingProcessTable:** It is a parameter to add or not add a delay while printing table.

**gdt:** It keeps global descriptor table which is for aligning memory segments properly.

### Important Methods of TaskManager for Multitasking:

#### AddTask:

```
Task* TaskManager::AddTask(Task* newTask, Priority priority, common::int32_t ppid)
{
    // Check Tasks array size. Return false, if it is full.
    if(numTasks >= MAX_NUM_TASKS)
        return 0; // null

    Task* task = &tasks[numTasks];
    task->Copy(newTask);
    task->SetPriority(priority);
    task->SetPid(numTasks);
    task->SetPPid(ppid);
    task->SetArrivalOrder(nextArrivalOrder);

    AddToReadyQueue(task);

    ++numTasks;
    ++nextArrivalOrder;

    return task;
}
```

This method adds a new task to the task manager.

It finds the next available process first.

Then copies the given task to this process. It does this because it needs to use the allocated space for the tasks. The copy method is described above in the methods of Task class.

It sets the pid with next available index and ppid with given ppid. It also sets the arrival order with next arrival order.

It adds the task to ready queue.

#### CollatzAdded:

```
void TaskManager::CollatzAdded()
{
    if (numTasks > 0) {
        collatzTask = &tasks[numTasks - 1];
        interruptNumAfterCollatz = 0;
    }
}
```

It understand that the last added process was collatz so it takes it as collatzTask reference add sets the interruptNumAfterCollatz counter to 0.

### SetLastTaskPriority:

```
void TaskManager::SetLastTaskPriority(common::uint32_t priority)
{
    Priority newPriority = (Priority) priority;
    if (numTasks > 0) {
        Task* task = &tasks[numTasks - 1];
        RemoveFromReadyQueue(task->GetPid());
        task->SetPriority(newPriority);
        AddToReadyQueue(task);
    }
}
```

It sets the last task's priority to the given priority. This method is called from syscall handler since it is a syscall that init process uses to give priority to the processes after fork.

### BlockForCollatz:

```
common::uint32_t TaskManager::BlockForCollatz(CPUState* cpustate)
{
    Task* runningTask = &tasks[currentTask];
    runningTask->SetCPUState(cpustate);
    runningTask->SetState(State::Blocked);

    blockedTaskForCollatz = runningTask;

    return (common::uint32_t) Schedule();
}
```

It sets the currently running process's state to Blocked. It is used in init process as syscall to block the init process until 5<sup>th</sup> interrupt after collatz task.

### Fork:

```
void TaskManager::Fork(CPUState* cpustate)
{
    Task* parent = GetCurrentTask();
    parent->SetCPUState(cpustate); // Save the current cpustate to the currently running process which is parent

    Task* child = AddTask(parent, parent->GetPriority(), parent->GetPid());
    if (child != 0) {
        child->CopyCpuState(cpustate); // Copy the cpustate in any case
    }

    // Set the fork pids
    parent->forkPid = child->GetPid();
    child->forkPid = 0;

    cpustate->ecx = child->GetPid();
    child->GetCPUState()->ecx = 0;
}
```

Fork firstly saves the cpustate to the parent process. Then copies the parent to the child process using AddTask method which described above. This method copies the whole stack and sets the necessary informations. Later, it also copies the current cpustate to the child process. Then, it sets forkPid's of processes to the parent and child processes. Later, these fork pids are used to determine if it is a parent or child process.

### Waitpid:

```
common::uint32_t TaskManager::Waitpid(common::uint32_t pid, CPUState* cpustate)
{
    Task* runningTask = GetCurrentTask();
    runningTask->SetCPUState(cpustate);

    // Waiting for any child of the current process
    if (pid == -1) {
        // Search through processes to find if there is a child which has been terminated
        uint32_t childId = 0;
        uint32_t childNum = 0;
        bool isFound = false;
        Task* task = 0; // null
        for (int i = 0; i < numTasks && !isFound; ++i) {
            task = &tasks[i];
            if (!task->GetParentTookInWait() && task->GetPPid() == runningTask->GetPid()) {
                ++childNum;
                if (task->GetState() == State::Terminated) {
                    childId = task->GetPid();
                    isFound = true;
                }
            }
        }

        // A terminated child is found
        if (isFound) {
            task->SetParentTookInWait(true);
            cpustate->eax = childId;
            return (common::uint32_t) cpustate;
        }

        // Not found any child
        if (childNum == 0) {
            cpustate->eax = childId = -1;
            runningTask->waitingChild = false;
            return (common::uint32_t) cpustate;
        }

        // Not found any terminated child
        runningTask->SetState(State::Blocked);
        runningTask->waitingChild = true;
        runningTask->waitingChildId = -1;

        // Schedule new process since this process is blocked
        return (common::uint32_t) Schedule();
    }
}
```



```

// Waiting for specific pid if it is actually the child of the current process
if (pid >= numTasks) {
    cpustate->eax = -1;
    runningTask->waitingChild = false;
    return (common::uint32_t) cpustate;
}

Task* task = &tasks[pid];

// The process with given pid is not child of this process
if (task->GetParentTookInWait() || task->GetPPid() != runningTask->GetPid()) {
    cpustate->eax = -1;
    runningTask->waitingChild = false;
    return (common::uint32_t) cpustate;
}

// The task is the child of the process as we check above, and its state is Terminated.
if (task->GetState() == State::Terminated) {
    task->SetParentTookInWait(true);
    cpustate->eax = pid;
    return (common::uint32_t) cpustate;
}

runningTask->SetState(State::Blocked);
runningTask->waitingChild = true;
runningTask->waitingChildId = pid;

// Schedule new process since this process is blocked
return (common::uint32_t) Schedule();
}

```

Waitpid checks if the searched child id is -1 or not. If it is -1, it search for all processes to find a child process which is terminated. If a terminated child process is found, then the eax register is set to its pid and it is done. If it is not found, but a child is found then it blocks the process until any child process is terminated. If there are no child process, then it sets eax to -1 to return -1 indicating there is no child process. If the pid is not -1, then it looks for the process with given pid, checks if it is a child process. If it is child process and is terminated then it returns, otherwise it blocks. If it is not child of this process it returns -1 with setting eax to -1.

### Execve:

```

common::uint32_t TaskManager::Execve(void (*entrypoint)())
{
    Task* runningTask = GetCurrentTask();
    runningTask->Reset(gdt, entrypoint);

    return (common::uint32_t) runningTask->GetCPUState();
}

```

It resets the running process with the given entry point and runs the scheduler. After that this running process will start from scratch with the given entry point.

## Exit:

```
common::uint32_t TaskManager::Exit()
{
    Task* runningTask = GetCurrentTask();
    runningTask->SetState(State::Terminated);

    // First check if the runningTask has a parent (init process does not have parent)
    if (runningTask->GetPPid() != -1) {
        Task* parent = &tasks[runningTask->GetPPid()];

        // Check if parent is waiting for a child, and the pid is either -1 or the pid of this task
        if (parent->waitingChild && (parent->waitingChildId == -1 || parent->waitingChildId == runningTask->GetPid()))
        {
            runningTask->SetParentTookInWait(true);

            parent->waitingChild = false;
            AddToReadyQueue(parent);

            CPUState* parentCpuState = parent->GetCPUState();
            parentCpuState->eax = runningTask->GetPid();
        }
    }

    // If it is collatz, make ready if there is a blocked process for collatz like init process
    if (collatzTask != 0 && runningTask->GetPid() == collatzTask->GetPid() && blockedTaskForCollatz != 0) {
        AddToReadyQueue(blockedTaskForCollatz);
        collatzTask = 0;
    }

    // Print process table if it is set to only in termination
    if (processTablePrintType == ProcessTablePrintType::PrintOnlyTermination) {
        PrintProcessTable();
    }

    return (common::uint32_t) Schedule();
}
```

It sets the current task's state to Terminated and schedules the next process. Meanwhile, it checks if any parent process is waiting for this child process so that the waiting parent process can continue. Also, it checks if the terminated process is collatz process and if there is a process waiting for collatz(which is init process of course); if the init process is waiting for collatz process and it terminates before 5<sup>th</sup> interrupt, it sets init process's state to Ready so that it can continue and there is no deadlock.

## 2.a) Scheduling:

I want to mention about important methods that do that scheduling according to the given scheduling parameters. You can find the scheduling parameters and lifecycles later on.

### AddToReadyQueue:

```
void TaskManager::AddToReadyQueue(Task* task)
{
    task->SetState(State::Ready);

    if (schedulerType == SchedulerType::RoundRobin) {
        AddToReadyQueueRoundRobin(task);
    }
    else {
        AddToReadyQueuePreemptivePriority(task);
    }
}

void TaskManager::AddToReadyQueueRoundRobin(Task* task)
{
    // Add the element at the tail
    readyQueue[queueLen] = task;
    ++queueLen;
}

void TaskManager::AddToReadyQueuePreemptivePriority(Task* task)
{
    // Insertion sort since it is an online sorting algorithm
    readyQueue[queueLen] = task; // Put it at the end firstly

    int priority = task->GetPriority();
    int arrivalOrder = task->GetArrivalOrder();

    bool isDone = false;
    int i = queueLen - 1;
    while (i >= 0 && !isDone) {
        int othPriority = readyQueue[i] -> GetPriority();
        int othArrivalOrder = readyQueue[i] -> GetArrivalOrder();

        if (priority < othPriority || (priority == othPriority && arrivalOrder < othArrivalOrder)) {
            readyQueue[i + 1] = readyQueue[i];
            --i;
        }
        else isDone = true;
    }
    readyQueue[i + 1] = task;
    ++queueLen;
}
```

These methods are used to add a task to ready queue according to the scheduling strategy. If it is round robin, it adds the process to end of the queue. If it is preemptive priority scheduling, it inserts the task using insertion sort. If the priority is high, or priorities are equal but arrival order is prior then this task should run first.

## Schedule Methods:

```
Task* TaskManager::PopFromReadyQueue()
{
    // Remove and return head
    Task* task = readyQueue[0]; // We know there is at least one element before calling this method already, so no need to extra check
    for (int i = 1; i < queueLen; ++i) {
        readyQueue[i - 1] = readyQueue[i];
    }
    --queueLen;
    return task;
}

// Round robin schedule
CPUState* TaskManager::RoundRobinSchedule()
{
    // Ready process'ler arasından bulup verilen strategy degiskenine gore next process'i bulup running yapacak
    int oldTask = currentTask;

    Task* task = PopFromReadyQueue();
    task->SetState(State::Running);
    currentTask = task->GetPid();

    // If there is a context switch, then print the process table
    if (processTablePrintType == ProcessTablePrintType::PrintEveryTimeInterrupt || (oldTask != currentTask && processTablePrintType == ProcessTablePrintType::PrintEverySwitch))
    {
        PrintProcessTable();
    }

    return tasks[currentTask].cpustate;
}

// Preemptive priority schedule
CPUState* TaskManager::PreemptivePrioritySchedule()
{
    // Ready process'ler arasından bulup verilen strategy degiskenine gore next process'i bulup running yapacak
    int oldTask = currentTask;

    Task* task = PopFromReadyQueue();
    task->SetState(State::Running);
    currentTask = task->GetPid();

    // If there is a context switch, then print the process table
    if (processTablePrintType == ProcessTablePrintType::PrintEveryTimeInterrupt || (oldTask != currentTask && processTablePrintType == ProcessTablePrintType::PrintEverySwitch))
    {
        PrintProcessTable();
    }

    return tasks[currentTask].cpustate;
}

// Schedules the next process according to the scheduler type
CPUState* TaskManager::Schedule()
{
    if (schedulerType == SchedulerType::RoundRobin) {
        return RoundRobinSchedule();
    }
    return PreemptivePrioritySchedule();
}
```

```

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

    // In strategy 4, the init task sets the scheduler to ignore schedule so that all processes are added to queue before scheduling
    if (lifeCycleType == LifeCycleType::LifeCycleB4 && ignoreSchedule) {
        return cpustate;
    }

    // If collatz task added, then increment the "interruptNumAfterCollatz" counter
    if (interruptNumAfterCollatz != -1) ++interruptNumAfterCollatz;

    // In strategy 4, the collatz's priority is set to High after 5th interrupt
    if (lifeCycleType == LifeCycleType::LifeCycleB4 && interruptNumAfterCollatz == 5) {
        printf("\nBEFORE 5th INTERRUPT (SEE TABLE BELOW) : \n");
        --interruptNumAfterCollatz;
        PrintProcessTable();
        Helper::Delay();

        RemoveFromReadyQueue(collatzTask->GetPid());
        collatzTask->SetPriority(Priority::High);
        AddToReadyQueue(collatzTask);

        ++interruptNumAfterCollatz;
        PrintProcessTable();
        Helper::Delay();
    }

    // If 5th interrupt after collatz tasks started, then unblock the init process which is blocked after adding collatz task to ready queue.
    if (lifeCycleType == LifeCycleType::LifeCycleB3 && interruptNumAfterCollatz != -1) {
        if (interruptNumAfterCollatz == 5 && blockedTaskForCollatz != 0) {
            printf("\nBEFORE 5th INTERRUPT (SEE TABLE BELOW) : \n");
            --interruptNumAfterCollatz;
            PrintProcessTable();
            Helper::Delay();

            // Start again the blocked init process
            AddToReadyQueue(blockedTaskForCollatz);
            blockedTaskForCollatz = 0; // null

            ++interruptNumAfterCollatz;
        }
    }

    if(currentTask >= 0) {
        tasks[currentTask].cpustate = cpustate;
        AddToReadyQueue(&tasks[currentTask]);
    }

    return Schedule();
}

```

In the Schedule method at the last screenshot, it checks if there is no process, then it returns the cpustate directly. If there is a process, then it checks if it needs to ignore it according to the parameters. If not ignore, it increments the interruptNumAfterCollatzCounter if collatz process added. If it is 4<sup>th</sup> lifecycle and interrupt number is 5 after collatz, it increments collatz processes priority. If it is 3<sup>th</sup> lifecycle and interrupt number is 5 after collatz, it makes the init process ready to work so that it can add other processes. In the end, it saves the stack registers of currently running process and calls another Schedule method which does the actual scheduling.

In the Schedule method at the 3<sup>rd</sup> screenshot, it uses proper methods according to parameters. Actually, both of them are popping from the queue and runs the next process. I was thinking if I add something later on, I can change them properly therefore they are separate processes. The important thing here is to add to ready queue method.



## DoHandleInterrupt Method (src/hardwarecommunication/interrupts.cpp) :

```
uint32_t InterruptManager::DoHandleInterrupt(uint8_t interrupt, uint32_t esp)
{
    CPUState* cpustate = (CPUState*)esp;

    //if (interrupt != 0x20) { // timer interrupt }

    // If we have handler for this interrupt, then forward the interrupt to this handler's HandleInterrupt method.
    if(handlers[interrupt] != 0)
    {
        esp = handlers[interrupt]->HandleInterrupt(esp);
    }
    else if(interrupt != hardwareInterruptOffset)
    {
        // If we don't have handler for this interrupt, print this message.
        printf("UNHANDLED INTERRUPT 0x");
        printfHex(interrupt);
        printf("\n");
    }

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

    // hardware interrupts must be acknowledged, otherwise next interrupts cannot be caught.
    if(hardwareInterruptOffset <= interrupt && interrupt < hardwareInterruptOffset+16)
    {
        programmableInterruptControllerMasterCommandPort.Write(0x20);
        if(hardwareInterruptOffset + 8 <= interrupt)
            programmableInterruptControllerSlaveCommandPort.Write(0x20);
    }

    return esp;
}
```

This method handles the interrupts. If it is timer interrupt, it calls the schedule method to schedule the next task. At the end, it acknowledges the interrupt to take another interrupts.

### 3) Syscalls: (kernel.cpp)

The syscalls are registered as 0x80 interrupt, when this interrupt happens the syscall handler starts working and runs the proper method from TaskManager.

#### Making syscalls:

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

int sysfork()
{
    asm("int $0x80" : : "a" (57));
}

void sysexecve(void (*entrypoint)())
{
    asm("int $0x80" : : "a" (59), "b" (entrypoint));
}

void sysexit()
{
    asm("int $0x80" : : "a" (8));
}

uint32_t syswaitpid(uint32_t pid)
{
    uint32_t result;
    asm("int $0x80" : : "a" (7), "b" (pid));
    asm("" : "=a"(result));
    return result;
}

void sysblockforcollatz()
{
    asm("int $0x80" : : "a" (9));
}

void syssetlasttaskpriority(Priority priority)
{
    asm("int $0x80" : : "a" (10), "b" ((uint32_t) priority));
}

uint32_t sysforkpid()
{
    uint32_t forkPid;
    asm("int $0x80" : "=a"(forkPid) : "a" (58));
    return forkPid;
}
```

```

uint32_t sysforkpid()
{
    uint32_t forkPid;
    asm("int $0x80" : "=a"(forkPid) : "a" (58));
    return forkPid;
}

void syscollatzadded()
{
    asm("int $0x80" : : "a" (11));
}

int sysrand()
{
    uint64_t clockCounter;
    asm("rdtsc": "=A"(clockCounter));

    int num = (int) (clockCounter * 345834 + 123456) / 23415;
    if (num < 0) num *= -1;
    return num;
}

```

In these syscalls, I'm making an interrupt of 0x80 using "int \$0x80" so it traps into the syscall handler. In eax register, I'm giving the syscall code to discriminate the syscalls.

If I need to return result, I'm using "=a" syntax to read the values inside registers after making syscalls.

## Handling Syscalls: (syscalls.cpp)

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

    switch(cpu->eax)
    {
        case 1:
            // write syscall in linux (sysprintf)
            printf((char*)cpu->ebx);
            break;

        case 57:
            // fork syscall number in linux (sysfork)
            taskManager->Fork(cpu);
            break;

        case 58:
            cpu->eax = taskManager->GetCurrentTask()->forkPid;
            break;

        case 59:
            // execve syscall number in linux (sysexecve)
            esp = taskManager->Execve((void (*)()) cpu->ebx);
            break;

        case 7:
            // waitpid syscall number in linux (syswaitpid)
            esp = taskManager->Waitpid(cpu->ebx, cpu);
            break;

        case 8:
            // exit
            esp = taskManager->Exit();
            break;

        case 9:
            esp = taskManager->BlockForCollatz(cpu);
            break;

        case 10:
            taskManager->SetLastTaskPriority(cpu->ebx);
            break;

        case 11:
            taskManager->CollatzAdded();
            break;

        default:
            break;
    }

    return esp;
}
```

Here, it calls the proper syscall handler according to the given syscall code in the eax register.

Lifecycles:

For Part-A:

```
void initA()
{
    sysfork();
    if (sysforkpid() == 0) {
        sysexecve(collatz);
    }
    sysfork();
    if (sysforkpid() == 0) {
        sysexecve(collatz);
    }
    sysfork();
    if (sysforkpid() == 0) {
        sysexecve(collatz);
    }

    sysfork();
    if (sysforkpid() == 0) {
        sysexecve(longRunningProgram);
    }
    sysfork();
    if (sysforkpid() == 0) {
        sysexecve(longRunningProgram);
    }
    sysfork();
    if (sysforkpid() == 0) {
        sysexecve(longRunningProgram);
    }

    while (syswaitpid(-1) != -1);
    printf("All programs terminated \n");

    while(1);
}
```

It runs each process 3 times, waits them and runs an infinite loop.



### For Part-B Lifecycle 1:

```
void initB1()
{
    void (*entryPoints[]) (void) = {collatz, linearSearch, binarySearch, longRunningProgram};

    int randNumber = sysrand() % 4;
    printf("Random number: ");
    printInteger(randNumber);
    printf("\n");

    void (*entrypoint)() = entryPoints[randNumber];
    for (int i = 0; i < 10; ++i) {
        sysfork();
        if (sysforkpid() == 0) {
            sysexecve(entrypoint);
        }
    }

    while (syswaitpid(-1) != -1);
    printf("All programs terminated \n");

    while(1);
}
```

It selects one task randomly, and runs it 10 times.

### For Part-B Lifecycle 2:

```
void initB2()
{
    void (*entryPoints[]) (void) = {collatz, linearSearch, binarySearch, longRunningProgram};

    int randNumber1 = sysrand() % 4;
    printf("Random number1: ");
    printInteger(randNumber1);
    printf("\n");

    int randNumber2 = randNumber1;
    while (randNumber2 == randNumber1) {
        randNumber2 = sysrand() % 4;
    }
    printf("Random number2: ");
    printInteger(randNumber2);
    printf("\n");

    void (*entrypoint1)() = entryPoints[randNumber1];
    void (*entrypoint2)() = entryPoints[randNumber2];

    for (int i = 0; i < 3; ++i) {
        sysfork();
        if (sysforkpid() == 0) {
            sysexecve(entrypoint1);
        }
    }

    for (int i = 0; i < 3; ++i) {
        sysfork();
        if (sysforkpid() == 0) {
            sysexecve(entrypoint2);
        }
    }

    while (syswaitpid(-1) != -1);
    printf("All programs terminated \n");

    while(1);
}
```

It selects 2 process randomly, and loads them 3 times.

### For Part-B Lifecycle 3:

```
void initB3()
{
    sysfork();
    if (sysforkpid() == 0) {
        sysexecve(&collatz);
    }
    syssetlasttaskpriority(Priority::Low);
    syscollatzadded();
    sysblockforcollatz();

    // Init other tasks like that
    sysfork();
    if (sysforkpid() == 0) {
        sysexecve(&longRunningProgram);
    }
    syssetlasttaskpriority(Priority::Low);

    sysfork();
    if (sysforkpid() == 0) {
        sysexecve(&binarySearch);
    }
    syssetlasttaskpriority(Priority::Low);

    sysfork();
    if (sysforkpid() == 0) {
        sysexecve(&linearSearch);
    }
    syssetlasttaskpriority(Priority::Low);

    while (syswaitpid(-1) != -1);
    printf("All programs terminated \n");

    while(1);
}
```

It loads and runs the collatz process. When 5<sup>th</sup> interrupt comes this init process is waken up by the OS and the other processes are loaded into memory with the same priority low as collatz as specified in the homework PDF.

For Part-B Lifecycle 4:

```
void initB4()
{
    taskManager->SetIgnoreSchedule(true);

    // Critical region: Prepare ready queue before start scheduling
    sysfork();
    if (sysforkpid() == 0) {
        sysexecve(&collatz);
    }
    syssetlasttaskpriority(Priority::Low);
    syscollatzadded();

    sysfork();
    if (sysforkpid() == 0) {
        sysexecve(&longRunningProgram);
    }
    syssetlasttaskpriority(Priority::Medium);

    sysfork();
    if (sysforkpid() == 0) {
        sysexecve(&binarySearch);
    }
    syssetlasttaskpriority(Priority::Medium);

    sysfork();
    if (sysforkpid() == 0) {
        sysexecve(&linearSearch);
    }
    syssetlasttaskpriority(Priority::Medium);

    taskManager->SetIgnoreSchedule(false);

    while (syswaitpid(-1) != -1);
    printf("All programs terminated \n");

    while(1);
}
```

Here, it ignores timer interrupt to schedule until all processes are loaded into memory. Collatz task is initialized with lowest priority among other processes. After 5<sup>th</sup> interrupt, OS makes it priority highest among others as setting it to High priority in the TaskManager Schedule method as described pages above.

## Running the OS and Parameters:

### PARAMETERS AND INPUTS:

```
LifeCycleType lifeCycleType = LifeCycleType::LifeCycleB4; // A, B1, B2, B3, B4
SchedulerType schedulerType = SchedulerType::PreemptivePriority; // PreemptivePriority, RoundRobin
ProcessTablePrintType processTablePrintType = ProcessTablePrintType::PrintEverySwitch; // PrintEverySwitch, PrintEveryTimeInterrupt,
PrintOnlyTermination, DoNotPrint
bool useDelayInPrintingProcessTable = false;

int collatzInputs[] = {7, 7, 7, 7, 7, 7, 7, 7, 7, 7};
int binarySearchInputs[] = {110, 110, 110, 110, 110, 110, 110, 110, 110, 110};
int linearSearchInputs[] = {175, 110, 80, 175, 175, 175, 175, 175, 175, 175};
int longRunningProgramInputs[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 10};

int collatzNo = 0;
int binarySearchNo = 0;
int linearSearchNo = 0;
int longRunningNo = 0;
```

In kernel.cpp, you'll find parameters at the top of the code.

You should set this parameters according to the scheduling strategy and lifecycle that you want.

LifeCycleType:

LifeCycleType::LifeCycleA: For Part-A

LifeCycleType::LifeCycleB1: For Part-B lifecycle 1

LifeCycleType::LifeCycleB2: For Part-B lifecycle 2

LifeCycleType::LifeCycleB3: For Part-B lifecycle 3

LifeCycleType::LifeCycleB4: For Part-B lifecycle 4

SchedulerType:

SchedulerType::RoundRobin

SchedulerType::PreemptivePriority

ProcessTablePrintType: You can set here to see results well.

ProcessTablePrintType::PrintEverySwitch: Prints process table in every context switch

ProcessTablePrintType::PrintEveryTimeInterrupt: Prints process table in every time interrupts

ProcessTablePrintType::PrintOnlyTermination: Prints process table in every process termination

ProcessTablePrintType::DoNotPrint: Does not print process table at all so that you can examine the process results well

useDelayInPrintingProcessTable: You can set this to true if you want to see delay so that you can examine the results and printed values in the screen well. If you want to see the results immediately, set it to false.

**NOTE:** If you directly want to see the results or if the delay is too much for your machine so that the processes cannot continue then set it to false. You can take record from virtual box to examine the results at least in this case.

**NOTE:** You should not run RoundRobin with lifecycles B3 and B4 since these lifecycles are designed to work only with PreemptivePriority scheduling.

Inputs:

Each process can run at max 10 times in a lifecycle. So, you can set the inputs here accordingly by setting the inputs array.

```
void binarySearch()
{
    int arr[] = { 10, 20, 80, 30, 60, 50, 110, 100, 130, 170 };
    int len = sizeof(arr) / sizeof(int);
    int x = binarySearchInputs[binarySearchNo++];

    // Insertion sort
    for (int i = 1; i < len; ++i) {
        int item = arr[i];
        int j = i - 1;
        bool isDone = false;
        while (j >= 0 && !isDone) {

void linearSearch()
{
    int arr[] = { 10, 20, 80, 30, 60, 50, 110, 100, 130, 170 };
    int len = sizeof(arr) / sizeof(int);
    int x = linearSearchInputs[linearSearchNo++];

    int resIdx = -1;
    for (int i = 0; i < len && resIdx == -1; ++i) {
        if (arr[i] == x) {
            resIdx = i;
        }
    }
}
```

You can set search functions array inputs by changing the arrays from here.

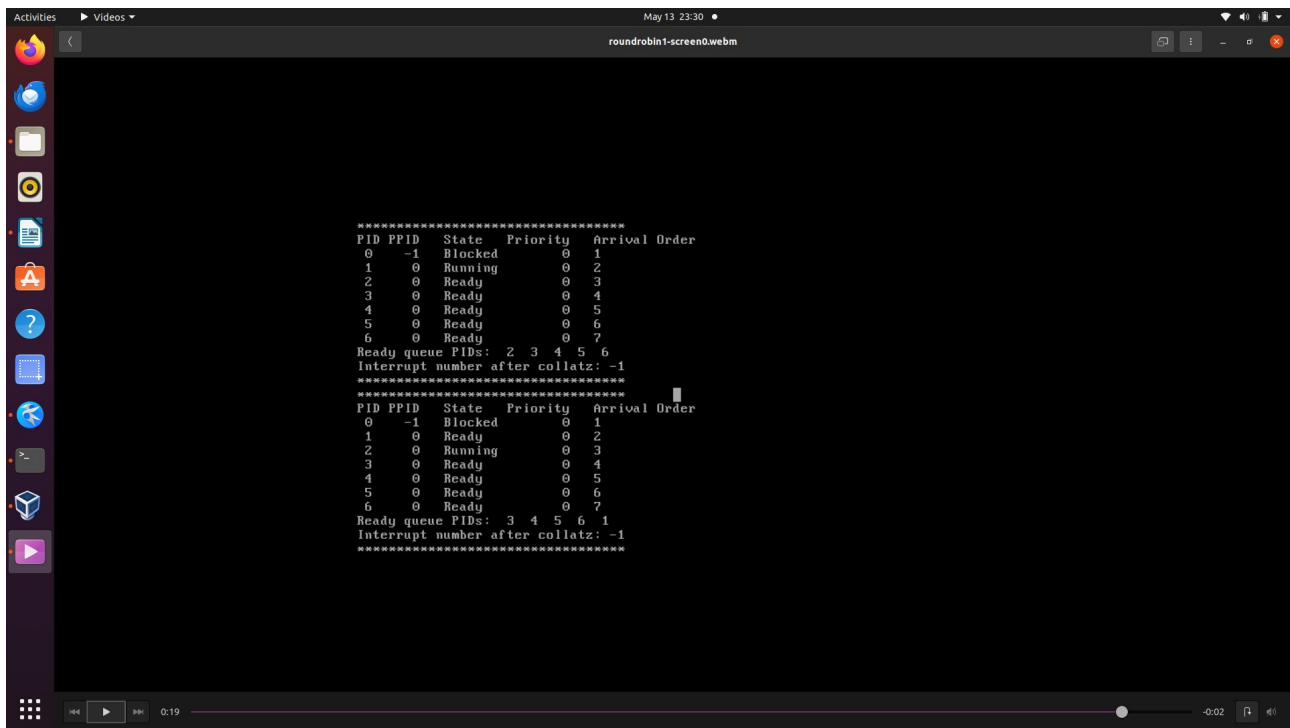
### HOW TO COMPILE AND RUN:

You can “make” to compile the project as a whole. You should add the iso to to the virtual box as Engellman describes and run it.



## SCREENSHOTS:

### PART-A:



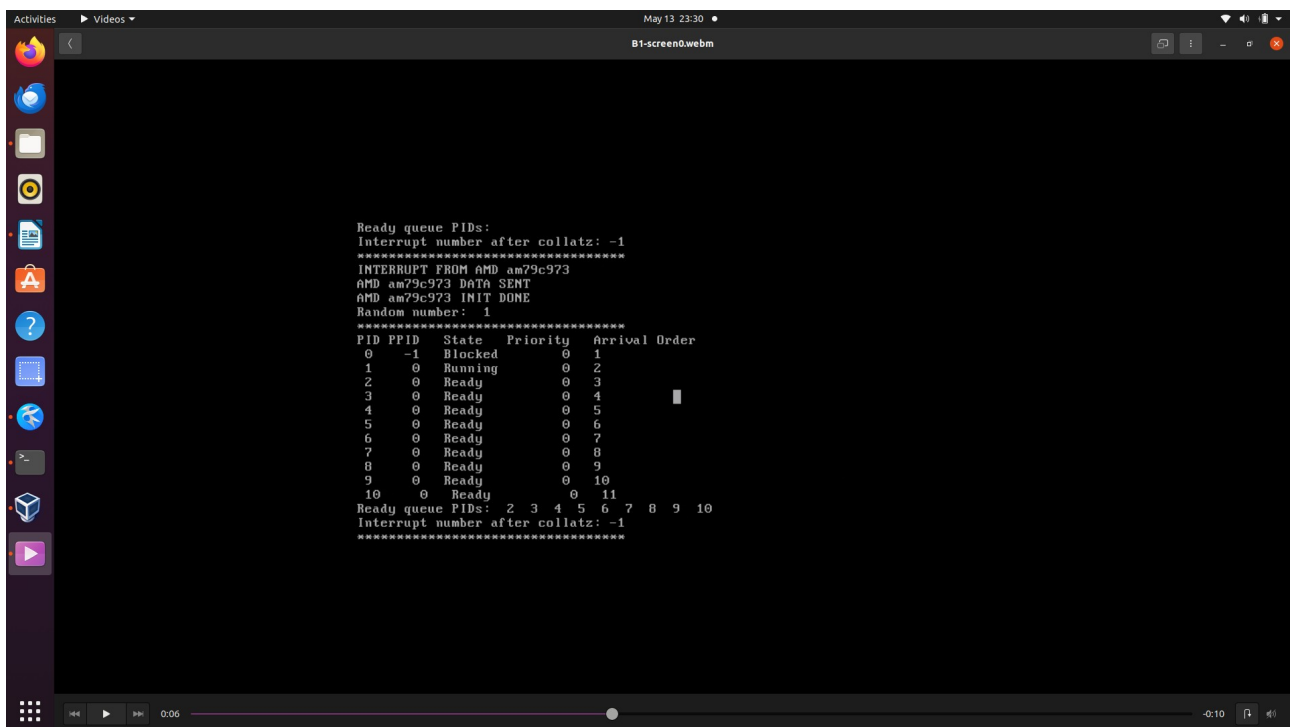
The screenshot shows a terminal window titled "roundrobin1-screen0.webm" with a dark background and white text. The terminal displays the output of a program that simulates round robin scheduling. It shows a table of process states (PID, PPID, State, Priority, Arrival, Order) and the sequence of processes in the ready queue. The processes are scheduled in a round robin manner, with the ready queue PIDs being 2, 3, 4, 5, 6, and 7. The interrupt number after collatz is -1.

```
=====
PID PPID  State  Priority  Arrival Order
0   -1   Blocked      0      1
1    0   Running      0      2
2    0   Ready       0      3
3    0   Ready       0      4
4    0   Ready       0      5
5    0   Ready       0      6
6    0   Ready       0      7
Ready queue PIDs: 2 3 4 5 6
Interrupt number after collatz: -1
=====
PID PPID  State  Priority  Arrival Order
0   -1   Blocked      0      1
1    0   Ready       0      2
2    0   Running      0      3
3    0   Ready       0      4
4    0   Ready       0      5
5    0   Ready       0      6
6    0   Ready       0      7
Ready queue PIDs: 3 4 5 6 1
Interrupt number after collatz: -1
=====
```

As you see, the tasks are scheduled in round robin manner and working accordingly.

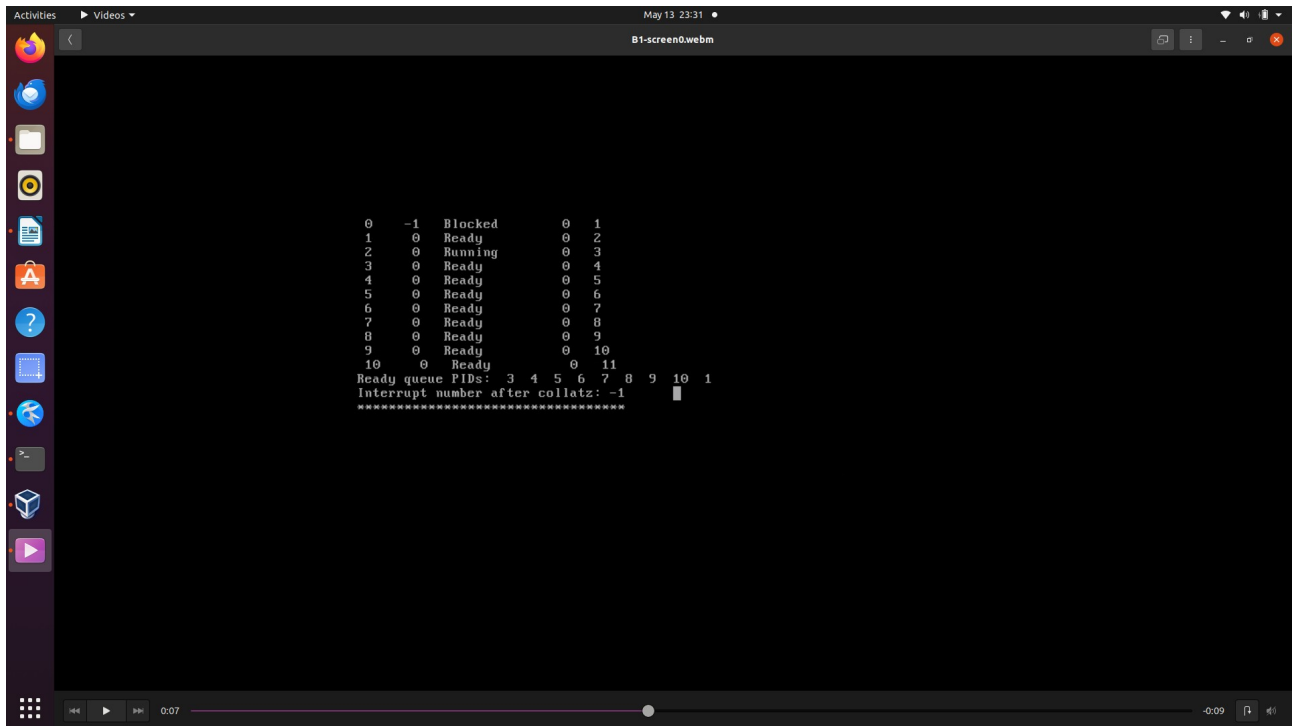
### PART-B:

#### Lifecycle1 – B1:



The screenshot shows a terminal window titled "B1-screen0.webm" with a dark background and white text. The terminal displays the output of a program that simulates round robin scheduling. It shows a table of process states (PID, PPID, State, Priority, Arrival, Order) and the sequence of processes in the ready queue. The processes are scheduled in a round robin manner, with the ready queue PIDs being 2, 3, 4, 5, 6, 7, 8, 9, and 10. The interrupt number after collatz is -1.

```
Ready queue PIDs:
Interrupt number after collatz: -1
=====
INTERRUPT FROM AMD am79c973
AMD am79c973 DATA SENT
AMD am79c973 INIT DONE
Random number: 1
=====
PID PPID  State  Priority  Arrival Order
0   -1   Blocked      0      1
1    0   Running      0      2
2    0   Ready       0      3
3    0   Ready       0      4
4    0   Ready       0      5
5    0   Ready       0      6
6    0   Ready       0      7
7    0   Ready       0      8
8    0   Ready       0      9
9    0   Ready       0     10
10   0   Ready       0     11
Ready queue PIDs: 2 3 4 5 6 7 8 9 10
Interrupt number after collatz: -1
=====
```

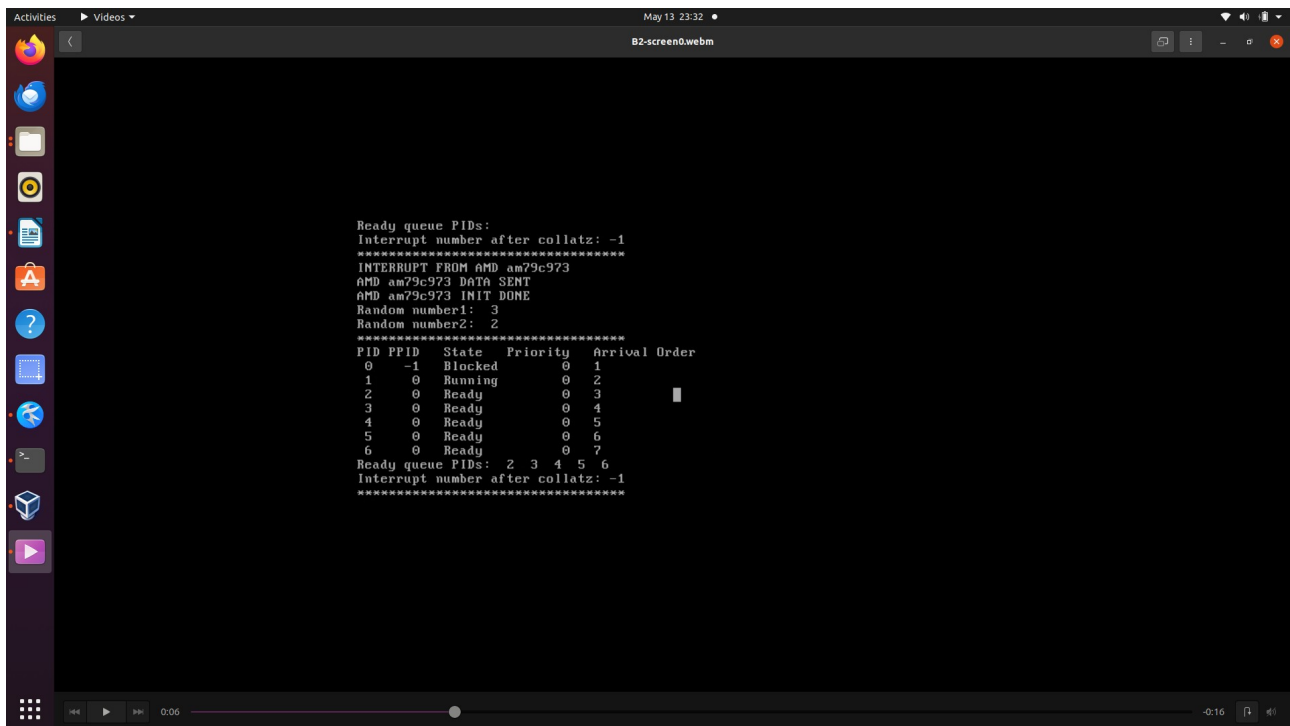


As you see, a random process is selected and loaded 10 times. The random number it prints in the screen indicates which process it selected among 4. The indexes are like that:

```
{collatz, linearSearch, binarySearch, longRunningProgram};  
0, 1, 2, 3
```

It runs the processes in RoundRobin manner now, if you wish it can work with PreemptivePriority scheduling also.

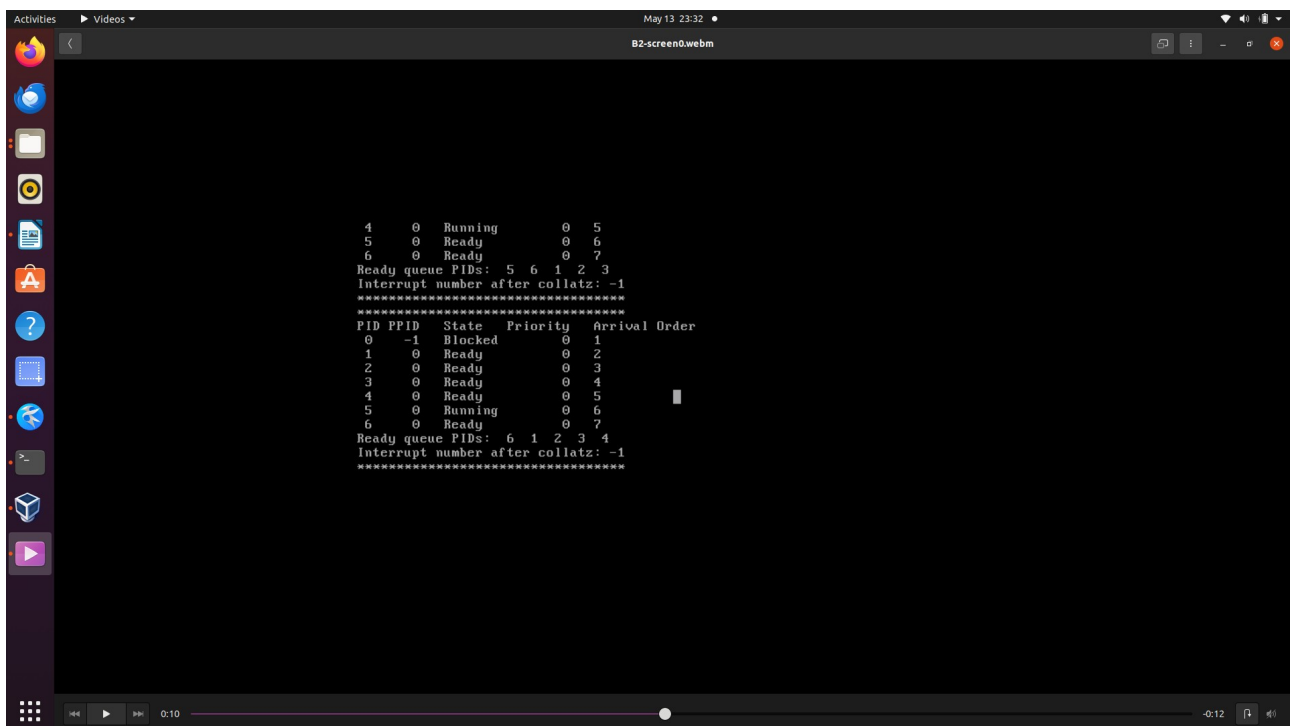
## Lifecycle2 – B2:



Activities Videos May 13 23:32 B2-screen0.webm

```
Ready queue PIDs:
Interrupt number after collatz: -1
*****
INTERRUPT FROM AMD am79c973
AMD am79c973 DATA SENT
AMD am79c973 INIT DONE
Random number1: 3
Random number2: 2
*****
PID PPID State Priority Arrival Order
0 -1 Blocked 0 1
1 0 Running 0 2
2 0 Ready 0 3
3 0 Ready 0 4
4 0 Ready 0 5
5 0 Ready 0 6
6 0 Ready 0 7
Ready queue PIDs: 2 3 4 5 6
Interrupt number after collatz: -1
*****
```

0:06 -0:16



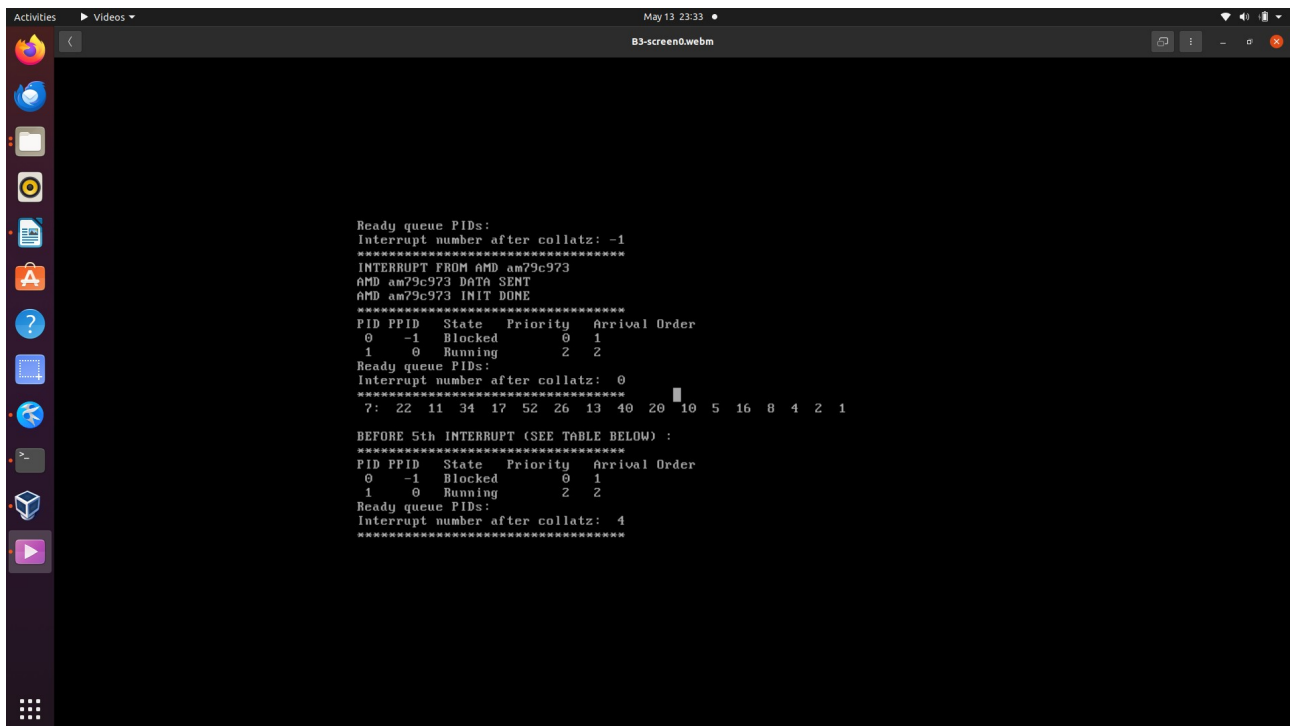
Activities Videos May 13 23:32 B2-screen0.webm

```
4 0 Running 0 5
5 0 Ready 0 6
6 0 Ready 0 7
Ready queue PIDs: 5 6 1 2 3
Interrupt number after collatz: -1
*****
PID PPID State Priority Arrival Order
0 -1 Blocked 0 1
1 0 Ready 0 2
2 0 Ready 0 3
3 0 Ready 0 4
4 0 Ready 0 5
5 0 Running 0 6
6 0 Ready 0 7
Ready queue PIDs: 6 1 2 3 4
Interrupt number after collatz: -1
*****
```

0:10 -0:12

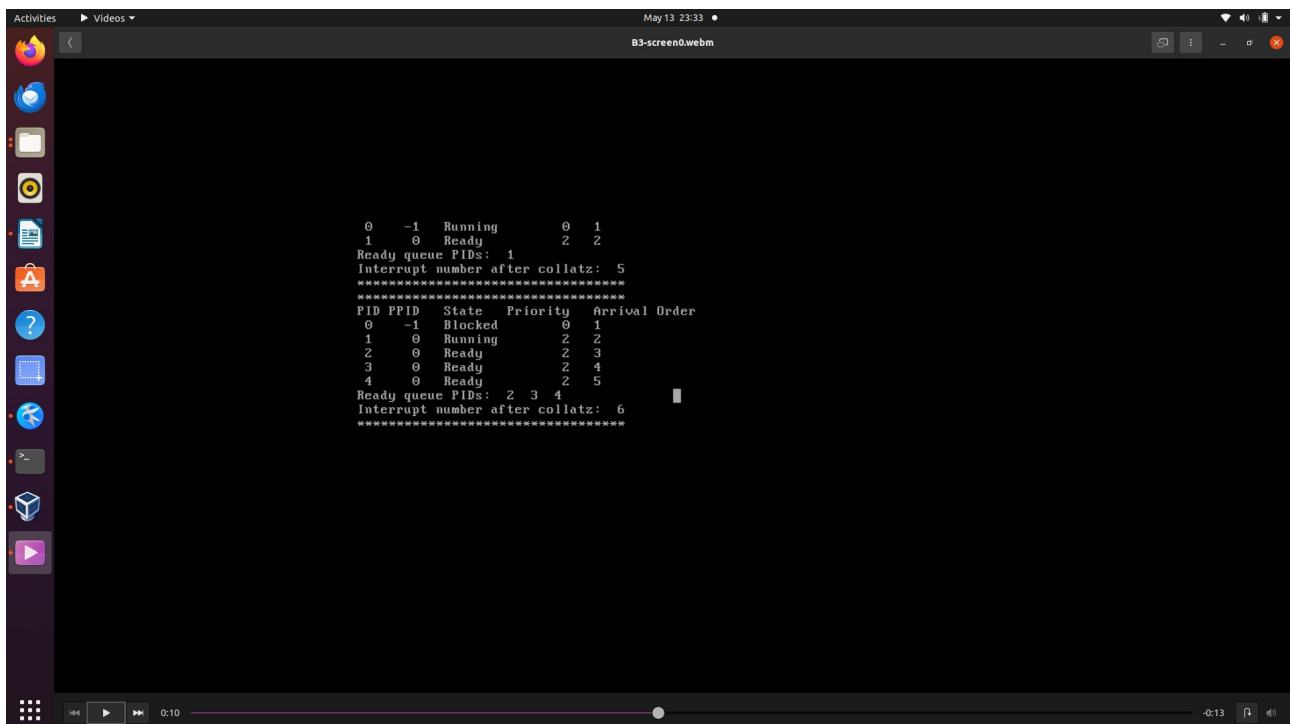
Here, it selects 2 process among 4 and loads them 3 times.  
It runs the processes in RoundRobin manner now, if you wish it can work with PreemptivePriority scheduling also.

## Lifecycle3 – B3:



```
Ready queue PIDs:
Interrupt number after collatz: -1
=====
INTERRUPT FROM AMD am79c973
AMD am79c973 DATA SENT
AMD am79c973 INIT DONE
=====
PID PPID State Priority Arrival Order
0 -1 Blocked 0 1
1 0 Running 2 2
Ready queue PIDs:
Interrupt number after collatz: 0
7: 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

BEFORE 5th INTERRUPT (SEE TABLE BELOW) :
=====
PID PPID State Priority Arrival Order
0 -1 Blocked 0 1
1 0 Running 2 2
Ready queue PIDs:
Interrupt number after collatz: 4
=====
```

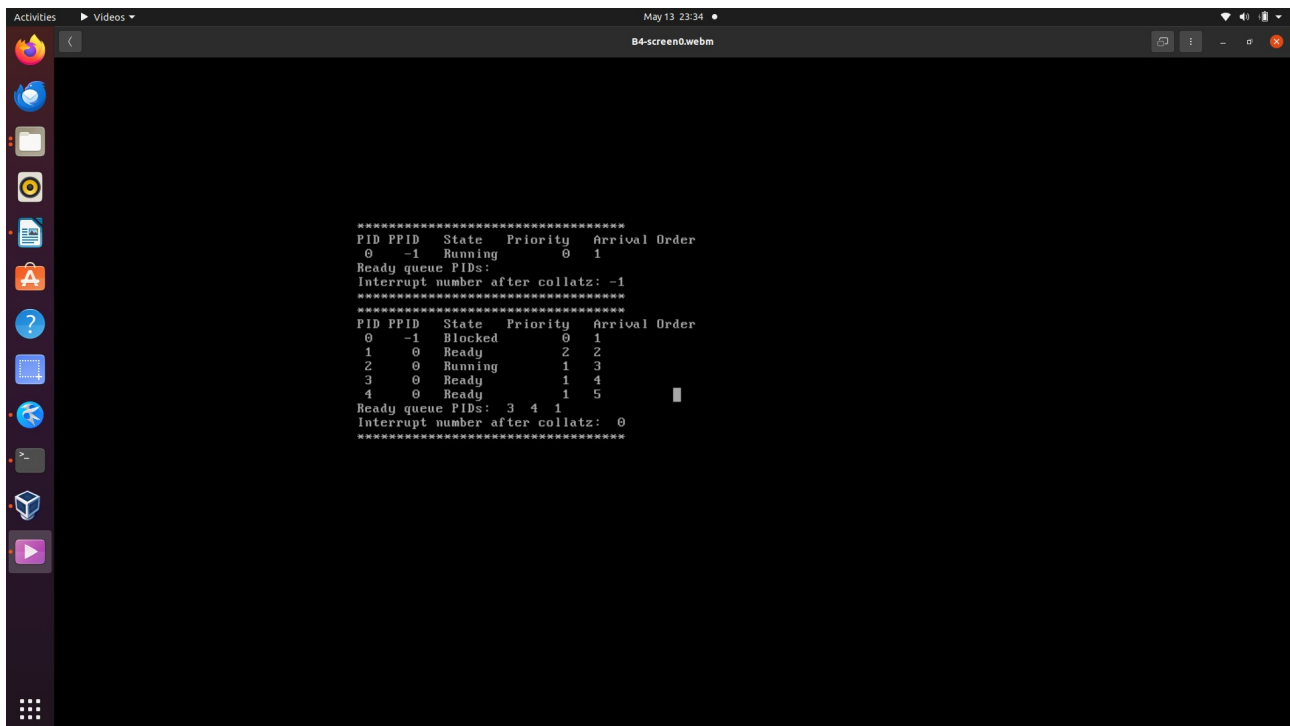


```
0 -1 Running 0 1
1 0 Ready 2 2
Ready queue PIDs: 1
Interrupt number after collatz: 5
=====
PID PPID State Priority Arrival Order
0 -1 Blocked 0 1
1 0 Running 2 2
2 0 Ready 2 3
3 0 Ready 2 4
4 0 Ready 2 5
Ready queue PIDs: 2 3 4
Interrupt number after collatz: 6
=====
```

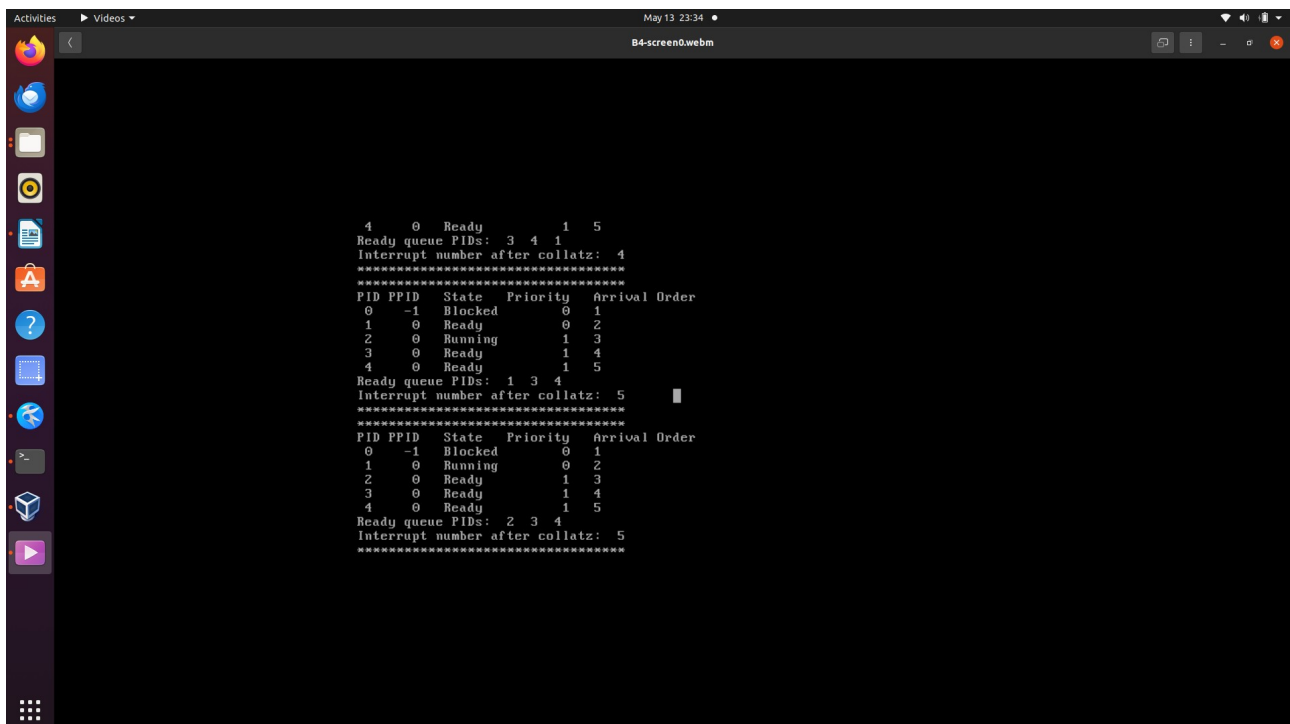
Here, firstly it runs collatz task for 5 interrupt after that it adds the other processes with the same priority as you can see from the screenshots. After 5<sup>th</sup> interrupt another tasks are added, but collatz task continues to run since it arrives first among other processes with the same priority.

It works with PreemptivePriority scheduling manner.

## Lifecycle4 – B4:



```
*****
PID PPID State Priority Arrival Order
0 -1 Running 0 1
Ready queue PIDs:
Interrupt number after collatz: -1
*****
PID PPID State Priority Arrival Order
0 -1 Blocked 0 1
1 0 Ready 2 2
2 0 Running 1 3
3 0 Ready 1 4
4 0 Ready 1 5
Ready queue PIDs: 3 4 1
Interrupt number after collatz: 0
*****
```



```
4 0 Ready 1 5
Ready queue PIDs: 3 4 1
Interrupt number after collatz: 4
*****
PID PPID State Priority Arrival Order
0 -1 Blocked 0 1
1 0 Ready 0 2
2 0 Running 1 3
3 0 Ready 1 4
4 0 Ready 1 5
Ready queue PIDs: 1 3 4
Interrupt number after collatz: 5
*****
PID PPID State Priority Arrival Order
0 -1 Blocked 0 1
1 0 Running 0 2
2 0 Ready 1 3
3 0 Ready 1 4
4 0 Ready 1 5
Ready queue PIDs: 2 3 4
Interrupt number after collatz: 5
*****
```

Here, it starts collatz process with priority 2(lowest priority), after 5<sup>th</sup> interrupt its priority is made 0(highest priority), so it continues to work until it terminates since it has the higher priority.

NOTE: I used the record feature of VirtualBox to take the screenshots and examine the results properly. You can use the same way to examine the results if the results are printed into the screen too quickly or too slowly. If too slow, then set useDelayInPrintingProcessTable to false.