I have two folders: one is named RealTime and the other is Tests. In RealTime folder everything happens like it would happen in a real os. At each timer interrupt scheduling occurs and etc. In Tests folder, I changed scheduling in a way that when ‘m’ key is pressed in keyboard, scheduling occurs and when space is pressed, the program on the cpu runs for a time between timer interrupts.

```
void collatz()
{
    for (int i = 2; i < 100; i++)
    {
        int n = i;

        printf("collatz sequence for ");
        printInteger(n);
        printf(":");
        while (n != 1)
        {
            if (n % 2 == 1)
                n = (3 * n) + 1;
            else
                n /= 2;

            printInteger(n);
            printf(",");
        }
        printf("\n");
        while (!CanRun::canRun)
            ;
    }
    CanRun::canRun = false;
    while (!CanRun::canRun);
    // while (1);
    sysexit();
}
```

For example above is updated version of collatz for testing. When space key is pressed canRun variable becomes true until the next timer interrupt. And when m key is pressed Schedule function runs and scheduling occurs. I added Test folder for the demo, it would be easier to show how my operating system works.

Also for PARTB_3 and PARTB_4, i updated collatz function to execute for each number less than 300 to make sure it runs long enough, otherwise it sometimes execute so fast.

PART A

I start with implementation of fork this is the basic of this homework in my opinion.

```
uint32_t sysfork()
{

    printf("sysfork called\n");
    uint32_t childPid;

    // system call fork and put the ebx register value into childpid (it will be 0 for child)
    asm("int $0x80" : "=b"(childPid) : "a"(2));

    printf("sysfork finished1\n");

    return childPid;
}
```

Sysfork function is used by init process to create process. When this function is called is causes to system call fork.

```
case 2: // fork

    child_pid = interruptManager->processManager->AddProcess(cpu);
    printf("child process created\n");
    ((CPUState *)esp)->ebx = child_pid; // write child pid as return value for parent
```

And this part in system call handler runs. It adds the process with AddProcess function.

```
common::uint32_t ProcessManager::AddProcess(CPUState *cpustate)
{

    if (numProcess >= 256)
        return 0;

    Process *parent = &processTable[currentProcess];
    Process *child = &processTable[numProcess];

    uint8_t *stackpPtr = parent->stack;
    child->CopyStack(stackpPtr);

    child->Init(cpustate->cs, cpustate->eip, numProcess, parent->pid);

    child->CopyCpuState(cpustate);
    // update necessary values
    // child will continue from where parent left, so make cpu state point to same place but in child stack
    child->cpustate = (CPUState*)((uint32_t)child->stack + (uint32_t) cpustate - (uint32_t) parent->stack);
    // ebp should be updated as well because we are changing to child stack
    child->cpustate->ebp = ((uint32_t)child->stack + (uint32_t) cpustate->ebp - (uint32_t) parent->stack);
    child->cpustate->eip = cpustate->eip;
    child->cpustate->ebx = 0; // child return value will be 0 for fork

    numProcess++;
    return child->pid;
}
```

Add process function simply copies the parent completely registers and stack as well. And then sets child's cpustate accordingly to child stack. Also ebp should be adjusted same way otherwise base pointer would point to some point in parent's stack and that would cause problems. Lastly ebx register is set to 0 to return 0 from fork to child.

Now we are ready to create processes with fork. So init function creates the processes.

```
void init()
{
    printf("init created\n");
    uint32_t pidArray[6];
    uint32_t pid;
    for (int i = 0; i < 3; i++)
    {
        pid = sysfork();
        if (pid == 0)
        {
            sysexecve(longRunningProgram);
        }
        else
        {
            pidArray[i] = pid;
        }
    }
    for (int i = 0; i < 3; i++)
    {
        pid = sysfork();
        if (pid == 0)
        {
            sysexecve(collatz);
        }
        else
        {
            pidArray[3 + i] = pid;
        }
    }
    // Wait for all children
    for (int i = 0; i < 6; i++)
    {
        syswaitpid(pidArray[i]);
        printf("pid terminated:");
        printInteger((int)pidArray[i]);
        printf("\n");
    }
    sysexit(); // after all children terminated, exit.
}
```

Init creates 3 longRunningProgram and 3 collatz. As you can see each process is created by fork as being the same as parent and then they changed their core with sysexecve function.

```
void sysexecve(void entrypoint())
{
    // set ecx register to entrypoint that points to the program that the process will run
    // the process will set its eip to ecx value and will be able to load the program
    asm("int $0x80" : : "a"(11), "c"(entrypoint));
}
void syswaitpid(uint32_t pid)
```

Sysexecve simply changes instruction pointer of the process. Entrypoint given in the init function is stored in ecx register and syscall for execve is called.

```
    break;
case 11: // execve
    // change the instruction pointer
    cpu->eip = cpu->ecx;
    // reset base pointer
    cpu->ebp = (uint32_t)curProcess->GetCpuState();
```

And in syscall, instruction pointer is set to ecx register which is new entrypoint and ebp of the process is reset to forget about the previously stored things in stack.

After that the last thing remaining is the syswaitpid().

```
void syswaitpid(uint32_t pid)
{
    asm("int $0x80" : : "a"(7), "b"(pid));
}
```

Syscall for waitpid is called.

```
    case 7: // waitpid
    printf("waitpid \n");
        // set the state of current process as waiting
        curProcess->SetState(Process::Waiting);
        // set the waitpid to determine the waited process
        curProcess->SetWaitpid((uint32_t) cpu->ebx);
        // reschedule
        esp = (uint32_t)interruptManager->processManager->Schedule(cpu);
        break;
```

Process' state is changed as waiting. It's waitpid field is set to the given pid. And Schedule function is called to put a ready process in cpu.

Waiting processes are controlled in Schedule function so let's look at Schedule function.

```
CPUState *ProcessManager::Schedule(CPUState *cpustate)
{


    clearScreen();
    printf("*****************************SCHEDULING*****************************\n");
    if (numProcess <= 0)
        return cpustate;

    if (currentProcess >= 0)
    {
        // do not schedule terminated processes
        if (processTable[currentProcess].state != Process::Terminated)
        {
            Process *curProcess = &processTable[currentProcess];
            (curProcess->cpustate) = cpustate;
            // if it is waiting, then don't make it ready
            if(processTable[currentProcess].state == Process::Running)
                processTable[currentProcess].SetState(Process::Ready);
            // printf("Interrupted Process");
            // processTable[currentProcess].PrintProcess();
        }
    }
```

Screen is cleared to see process table and other informations.

If we have a running process it should be saved if it is not terminated and should be set as ready, cpu may run it again.

```
    }
    // traverse the list until a non terminated process is found
    do
    {
        if (++currentProcess >= numProcess)
        {
            currentProcess %= numProcess;
        }
        uint32_t waitpid = processTable[currentProcess].waitpid;
        // if waitpid parameter is
        if (processTable[currentProcess].state == Process::Waiting && waitpid <= 0)
        {
            // skip init, it cannot be child
            for (int i = 1; i < numProcess; i++)
            {
                if (processTable[i].ppid == processTable[currentProcess].pid && processTable[i].state == Process::Terminated)
                {
                    processTable[i].ppid = -1;
                    processTable[currentProcess].state = Process::Ready;
                }
            }
        }
        else if (processTable[currentProcess].state == Process::Waiting && waitpid > 0)
        {
            if (processTable[waitpid].state == Process::Terminated)
            {
                processTable[waitpid].ppid = -1;
                processTable[currentProcess].state = Process::Ready;
            }
        }
    } while (processTable[currentProcess].state == Process::Terminated || processTable[currentProcess].state == Process::Waiting);
```

This is the part where cpu selects a process to run. Because it should be round robin, cpu traverse the process table until a ready process is found. If a process' state is waiting then if its waitpid is <=0 then if any child of the process is terminated, then

process is set as ready. But if waitpid is >0 then the process having pid == waitpid is searched. And if it is terminated, then the process is set as ready and terminated child's ppid is set as -1 to make sure it is not waited again.

```
processTable[currentProcess].SetState(Process::Running);
printf("|-----------------PROCESS TABLE-----------------|\n");
for (int i = 0; i < numProcess; i++)
{
    processTable[i].PrintProcess();
}
printf("|-----------------------------------------------|\n");

return processTable[currentProcess].cpustate;
```

Once proper process is found its state is set as running and it is loaded into cpu.

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

This is the sysexit function, it simply calls exit system call

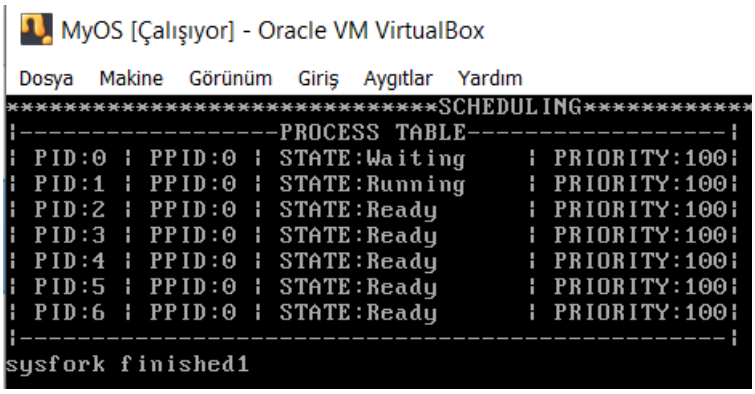
```
case 1: // exit
    printf("exit called\n");
    curProcess->SetState(Process::Terminated);
    // schedule the next process
    esp = (uint32_t)interruptManager->processManager->Schedule(nullptr);
    break;
```

In exit system call, process's state is set as terminated and scheduler is called to put another process in cpu.

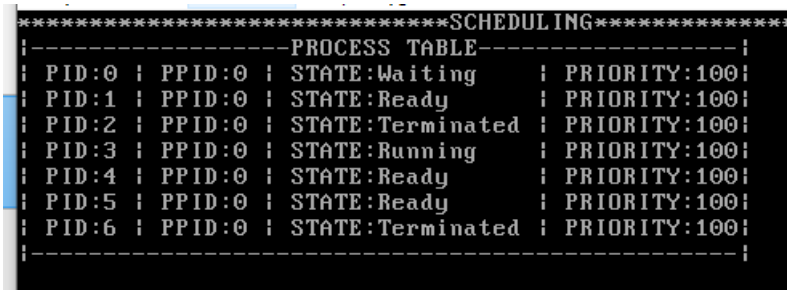
RESULTS:

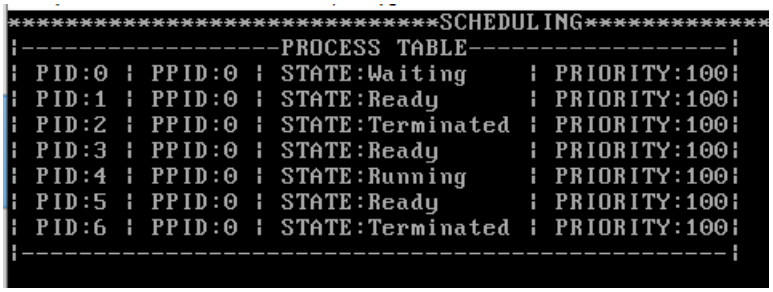
It is not easy to show results with screen shots but i will do my best.

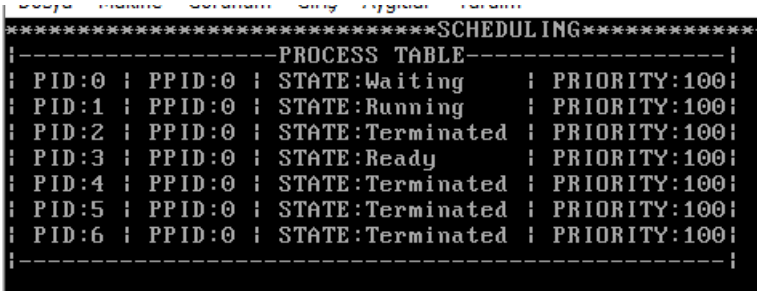
This is the process table when all the processes are loaded by init and init is waiting for them to terminate.



At each timer interrupt the next process runs







When all of them are terminated, init starts to run

At the end init terminates.

PART B_1 First Strategy

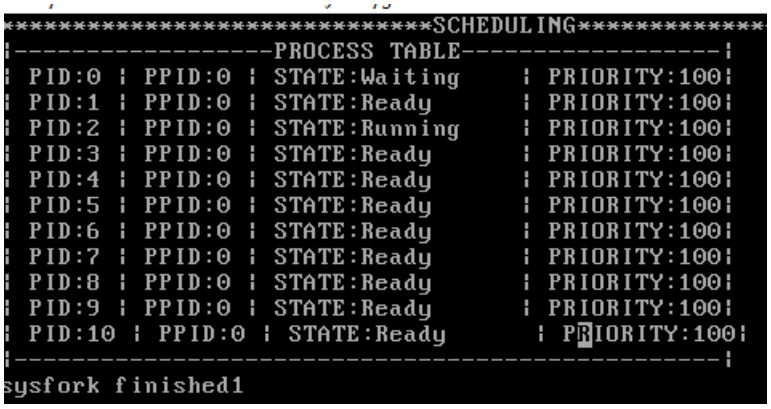
All the things for part A is valid for this part as well. The only difference is init function. Because in this strategy, round robin scheduling is used just like in part A.

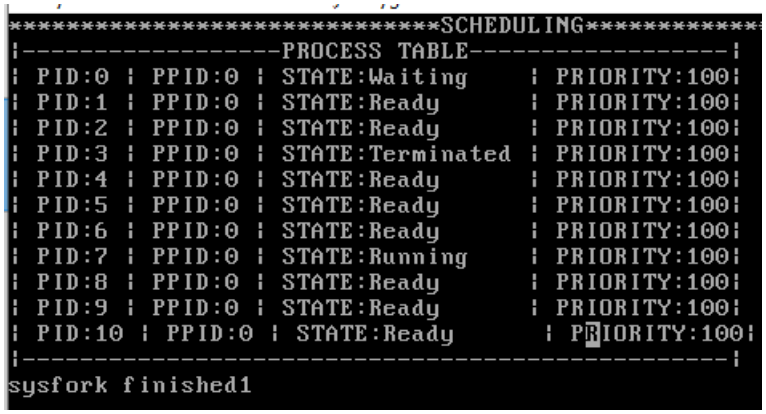
```c
void init()
{
    printf("init created\n");
    uint32_t pidArray[10];
    void (*programs[])() = {collatz,longRunningProgram,binarySearch,linearSearch};
    void (*selectedProgram)() = programs[rand(0,4)];
    uint32_t pid;
    for (int i = 0; i < 10; i++)
    {
        pid = sysfork();
        if (pid == 0)
        {
            sysexecve(selectedProgram);
        }
        else
        {
            pidArray[i] = pid;
        }
    }

    // Wait for all children
    for (int i = 0; i < 10; i++)
    {
        syswaitpid(pidArray[i]);
        printf("pid terminated:");
        printInteger((int)pidArray[i]);
        printf("\n");
    }

    sysexit(); // after all children terminated, exit.
}
```

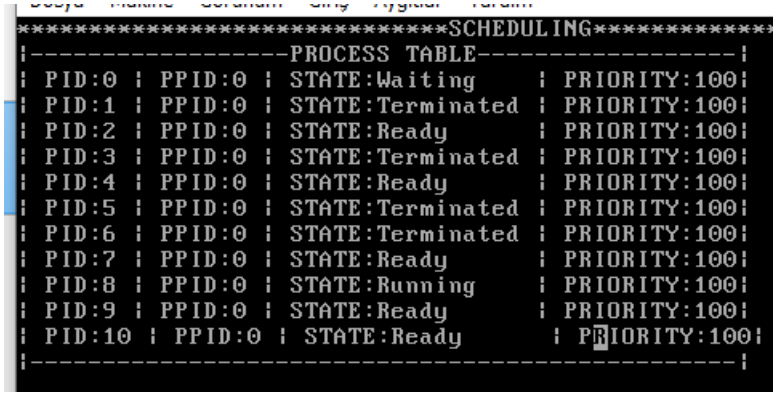
One of the programs is selected randomly and it is loaded into memory ten times. And init waits for all of them with syswaitpid function.

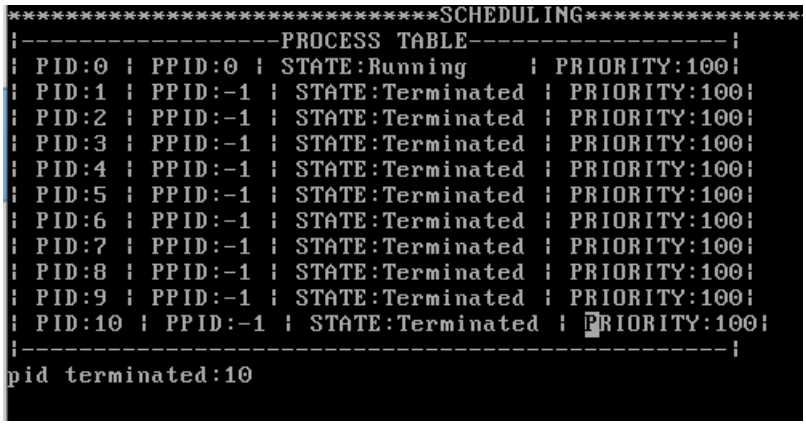
RESULTS

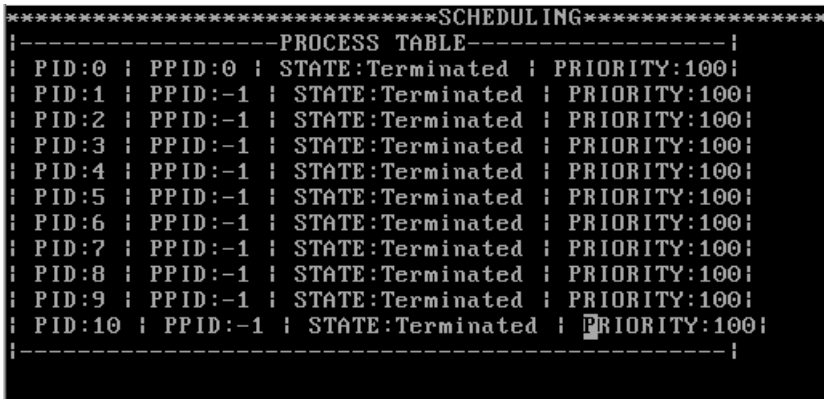
```
*******************************SCHEDULING*************
¦------------------PROCESS TABLE-------------------¦
¦ PID:0 ¦ PPID:0 ¦ STATE:Waiting    ¦ PRIORITY:100¦
¦ PID:1 ¦ PPID:0 ¦ STATE:Running    ¦ PRIORITY:100¦
¦ PID:2 ¦ PPID:0 ¦ STATE:Ready      ¦ PRIORITY:100¦
¦ PID:3 ¦ PPID:0 ¦ STATE:Ready      ¦ PRIORITY:100¦
¦ PID:4 ¦ PPID:0 ¦ STATE:Ready      ¦ PRIORITY:100¦
¦ PID:5 ¦ PPID:0 ¦ STATE:Ready      ¦ PRIORITY:100¦
¦ PID:6 ¦ PPID:0 ¦ STATE:Ready      ¦ PRIORITY:100¦
¦ PID:7 ¦ PPID:0 ¦ STATE:Ready      ¦ PRIORITY:100¦
¦ PID:8 ¦ PPID:0 ¦ STATE:Ready      ¦ PRIORITY:100¦
¦ PID:9 ¦ PPID:0 ¦ STATE:Ready      ¦ PRIORITY:100¦
¦ PID:10 ¦ PPID:0 ¦ STATE:Ready      ¦ PRIORITY:100¦
¦-----------------------------------------------¦
sysfork finished1
```


**********SCHEDULING**********
----------PROCESS TABLE----------
PID:0 | PPID:0 | STATE:Waiting | PRIORITY:100
PID:1 | PPID:0 | STATE:Ready | PRIORITY:100
PID:2 | PPID:0 | STATE:Running | PRIORITY:100
PID:3 | PPID:0 | STATE:Ready | PRIORITY:100
PID:4 | PPID:0 | STATE:Ready | PRIORITY:100
PID:5 | PPID:0 | STATE:Ready | PRIORITY:100
PID:6 | PPID:0 | STATE:Ready | PRIORITY:100
PID:7 | PPID:0 | STATE:Ready | PRIORITY:100
PID:8 | PPID:0 | STATE:Ready | PRIORITY:100
PID:9 | PPID:0 | STATE:Ready | PRIORITY:100
PID:10 | PPID:0 | STATE:Ready | PRIORITY:100
sysfork finished1


**********SCHEDULING**********
----------PROCESS TABLE----------
PID:0 | PPID:0 | STATE:Waiting | PRIORITY:100
PID:1 | PPID:0 | STATE:Ready | PRIORITY:100
PID:2 | PPID:0 | STATE:Ready | PRIORITY:100
PID:3 | PPID:0 | STATE:Terminated | PRIORITY:100
PID:4 | PPID:0 | STATE:Ready | PRIORITY:100
PID:5 | PPID:0 | STATE:Ready | PRIORITY:100
PID:6 | PPID:0 | STATE:Ready | PRIORITY:100
PID:7 | PPID:0 | STATE:Running | PRIORITY:100
PID:8 | PPID:0 | STATE:Ready | PRIORITY:100
PID:9 | PPID:0 | STATE:Ready | PRIORITY:100
PID:10 | PPID:0 | STATE:Ready | PRIORITY:100
sysfork finished1


**********SCHEDULING**********
----------PROCESS TABLE----------
PID:0 | PPID:0 | STATE:Waiting | PRIORITY:100
PID:1 | PPID:0 | STATE:Terminated | PRIORITY:100
PID:2 | PPID:0 | STATE:Ready | PRIORITY:100
PID:3 | PPID:0 | STATE:Terminated | PRIORITY:100
PID:4 | PPID:0 | STATE:Ready | PRIORITY:100
PID:5 | PPID:0 | STATE:Terminated | PRIORITY:100
PID:6 | PPID:0 | STATE:Terminated | PRIORITY:100
PID:7 | PPID:0 | STATE:Ready | PRIORITY:100
PID:8 | PPID:0 | STATE:Running | PRIORITY:100
PID:9 | PPID:0 | STATE:Ready | PRIORITY:100
PID:10 | PPID:0 | STATE:Ready | PRIORITY:100


**********SCHEDULING**********
----------PROCESS TABLE----------
PID:0 | PPID:0 | STATE:Running | PRIORITY:100
PID:1 | PPID:-1 | STATE:Terminated | PRIORITY:100
PID:2 | PPID:-1 | STATE:Terminated | PRIORITY:100
PID:3 | PPID:-1 | STATE:Terminated | PRIORITY:100
PID:4 | PPID:-1 | STATE:Terminated | PRIORITY:100
PID:5 | PPID:-1 | STATE:Terminated | PRIORITY:100
PID:6 | PPID:-1 | STATE:Terminated | PRIORITY:100
PID:7 | PPID:-1 | STATE:Terminated | PRIORITY:100
PID:8 | PPID:-1 | STATE:Terminated | PRIORITY:100
PID:9 | PPID:-1 | STATE:Terminated | PRIORITY:100
PID:10 | PPID:-1 | STATE:Terminated | PRIORITY:100
pid terminated:10



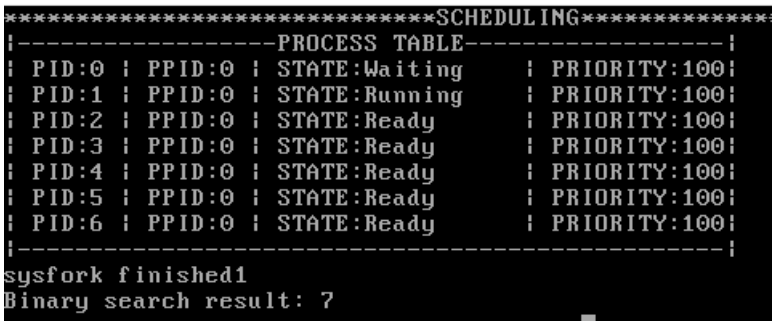
PART B_2 Second Strategy

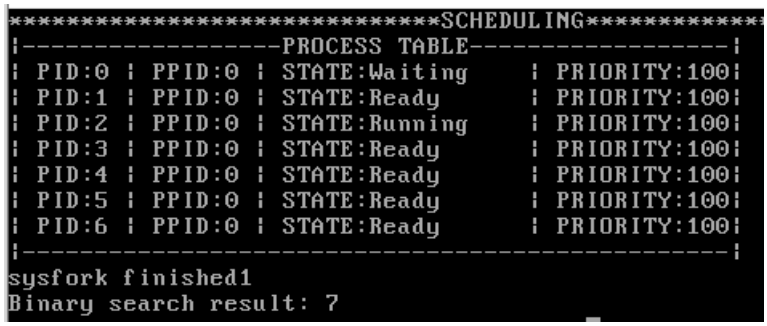
For this part the way processes are loaded is different. Round robin scheduling is used so that no difference from the previous parts except the init function.

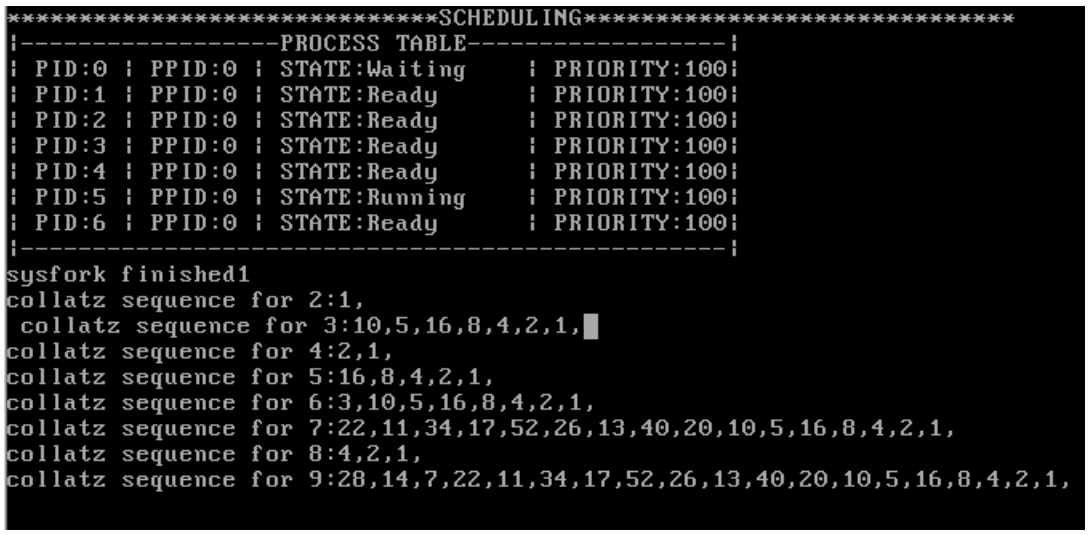
```
void init()
{
    printf("init created\n");
    uint32_t pidArray[6];
    void (*programs[])() = {collatz,longRunningProgram,binarySearch,linearSearch};
    int rand1 = rand(0, 4);
    int rand2 = rand(0, 4);
    //  make sure different programs will be selected
    while (rand2 == rand1)
    {
        rand2 = rand(0, 4);
    }
    void (*selectedProgram1)() = programs[rand1];
    void (*selectedProgram2)() = programs[rand2];
    uint32_t pid;
    for (int i = 0; i < 3; i++)
    {
        pid = sysfork();
        if (pid == 0)
        {
            sysexecve(selectedProgram1);
        }
        else
        {
            pidArray[i] = pid;
        }
    }
    for (int i = 0; i < 3; i++)
    {
        pid = sysfork();
        if (pid == 0)
        {
            sysexecve(selectedProgram2);
        }
        else
        {
            pidArray[i+3] = pid;
        }
    }

    // Wait for all children
    for (int i = 0; i < 6; i++)
    {
        syswaitpid(pidArray[i]);
        printf("pid terminated:");
        printInteger((int)pidArray[i]);
        printf("\n");
    }
    sysexit(); // after all children terminated, exit.
```

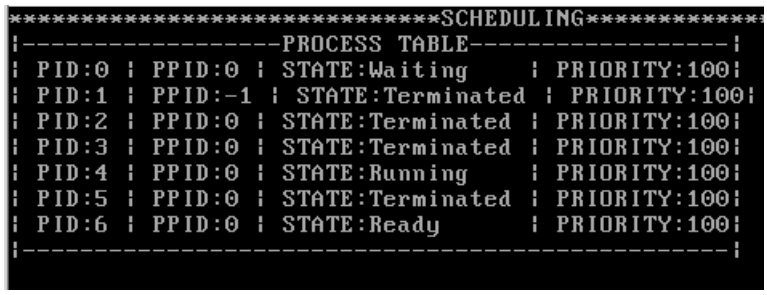
In this part two different programs are chosen by init randomly, and both of them loaded three times.

RESULTS


**********SCHEDULING**********
----------PROCESS TABLE----------
PID:0 | PPID:0 | STATE:Waiting | PRIORITY:100
PID:1 | PPID:0 | STATE:Running | PRIORITY:100
PID:2 | PPID:0 | STATE:Ready | PRIORITY:100
PID:3 | PPID:0 | STATE:Ready | PRIORITY:100
PID:4 | PPID:0 | STATE:Ready | PRIORITY:100
PID:5 | PPID:0 | STATE:Ready | PRIORITY:100
PID:6 | PPID:0 | STATE:Ready | PRIORITY:100
sysfork finished1
Binary search result: 7


**********SCHEDULING**********
----------PROCESS TABLE----------
PID:0 | PPID:0 | STATE:Waiting | PRIORITY:100
PID:1 | PPID:0 | STATE:Ready | PRIORITY:100
PID:2 | PPID:0 | STATE:Running | PRIORITY:100
PID:3 | PPID:0 | STATE:Ready | PRIORITY:100
PID:4 | PPID:0 | STATE:Ready | PRIORITY:100
PID:5 | PPID:0 | STATE:Ready | PRIORITY:100
PID:6 | PPID:0 | STATE:Ready | PRIORITY:100
sysfork finished1
Binary search result: 7


**********SCHEDULING**********
----------PROCESS TABLE----------
PID:0 | PPID:0 | STATE:Waiting | PRIORITY:100
PID:1 | PPID:0 | STATE:Ready | PRIORITY:100
PID:2 | PPID:0 | STATE:Ready | PRIORITY:100
PID:3 | PPID:0 | STATE:Ready | PRIORITY:100
PID:4 | PPID:0 | STATE:Ready | PRIORITY:100
PID:5 | PPID:0 | STATE:Running | PRIORITY:100
PID:6 | PPID:0 | STATE:Ready | PRIORITY:100
sysfork finished1
collatz sequence for 2:1,
collatz sequence for 3:10,5,16,8,4,2,1,
collatz sequence for 4:2,1,
collatz sequence for 5:16,8,4,2,1,
collatz sequence for 6:3,10,5,16,8,4,2,1,
collatz sequence for 7:22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1,
collatz sequence for 8:4,2,1,
collatz sequence for 9:28,14,7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1,


**********SCHEDULING**********
----------PROCESS TABLE----------
PID:0 | PPID:0 | STATE:Waiting | PRIORITY:100
PID:1 | PPID:-1 | STATE:Terminated | PRIORITY:100
PID:2 | PPID:0 | STATE:Terminated | PRIORITY:100
PID:3 | PPID:0 | STATE:Terminated | PRIORITY:100
PID:4 | PPID:0 | STATE:Running | PRIORITY:100
PID:5 | PPID:0 | STATE:Terminated | PRIORITY:100
PID:6 | PPID:0 | STATE:Ready | PRIORITY:100

```
*****************************SCHEDULING*************
¦-----------------PROCESS TABLE-----------------¦
¦ PID:0 ¦ PPID:0 ¦ STATE:Running    ¦ PRIORITY:100¦
¦ PID:1 ¦ PPID:-1 ¦ STATE:Terminated ¦ PRIORITY:100¦
¦ PID:2 ¦ PPID:-1 ¦ STATE:Terminated ¦ PRIORITY:100¦
¦ PID:3 ¦ PPID:-1 ¦ STATE:Terminated ¦ PRIORITY:100¦
¦ PID:4 ¦ PPID:-1 ¦ STATE:Terminated ¦ PRIORITY:100¦
¦ PID:5 ¦ PPID:-1 ¦ STATE:Terminated ¦ PRIORITY:100¦
¦ PID:6 ¦ PPID:-1 ¦ STATE:Terminated ¦ PRIORITY:100¦
¦-----------------------------------------------¦
pid terminated:6
```

```
*****************************SCHEDULING*************
¦-----------------PROCESS TABLE-----------------¦
¦ PID:0 ¦ PPID:0 ¦ STATE:Terminated ¦ PRIORITY:100¦
¦ PID:1 ¦ PPID:-1 ¦ STATE:Terminated ¦ PRIORITY:100¦
¦ PID:2 ¦ PPID:-1 ¦ STATE:Terminated ¦ PRIORITY:100¦
¦ PID:3 ¦ PPID:-1 ¦ STATE:Terminated ¦ PRIORITY:100¦
¦ PID:4 ¦ PPID:-1 ¦ STATE:Terminated ¦ PRIORITY:100¦
¦ PID:5 ¦ PPID:-1 ¦ STATE:Terminated ¦ PRIORITY:100¦
¦ PID:6 ¦ PPID:-1 ¦ STATE:Terminated ¦ PRIORITY:100¦
¦-----------------------------------------------¦
```

PARTB_3 Third Strategy

This is different from the previous strategies because now, priorities should be handled. Also, ready queue needed. For this reason i created a readyqueue class which simply handles enqueue and dequeue operations.

```
class ReadyQueue
{
public:
    void printQueue();
    void enqueue(myos::Process *process);
    myos::Process* dequeue();
    void insert(myos::Process *process,int pos);
    ReadyQueue();

    myos::Process *arr[30];
    static int size ;
private:
```

This is the structure of ReadyQueue.

```
void myos::ReadyQueue::enqueue(myos::Process *process)
{

    int index = 0;
    for (int i = 0; i < size; i++)
    {
        if (process->GetPriority() < arr[i]->GetPriority())
        {
            index = i + 1;
        }
    }
    size++;
    insert(process, index);
}
myos::Process *myos::ReadyQueue::dequeue()
{
    size--;
    return arr[size];
}
```

As you can see, enqueue function enqueues according to priority. And dequeue function removes the last element. So the process with the highes priority (lowes priority number) will be placed at the end, when we dequeue we will get the highest priority process.

Also there is another class which is BlockedList

```
class BlockedList
{
public:
    Process *elementAt(int index);
    Process *removeAt(int index);
    void printList();
    void add(myos::Process *process);
    BlockedList();

    myos::Process *arr[30];
    static int size;

private:
};
```

I created this to store waiting processes.

Both readyqueue and blockedlist are stored in processManager.

Let’s look at init process

```
void init()
{
    printf("init created\n");
    uint32_t pidArray[4];
    void (*programs[])() = {collatz,longRunningProgram,binarySearch,linearSearch};
    uint32_t pid;
    pid = sysfork();
    if (pid == 0)
    {
        sysexecve(collatz);
    }
    pidArray[0] = pid;

    sysblock();

    pid = sysfork();
    if (pid == 0)
    {
        sysexecve(longRunningProgram);
    }
    pidArray[1] = pid;
    pid = sysfork();
    if (pid == 0)
    {
        sysexecve(binarySearch);
    }
    pidArray[2] = pid;
    pid = sysfork();
    if (pid == 0)
    {
        sysexecve(linearSearch);
    }
    pidArray[3] = pid;

    // Wait for all children
    for (int i = 0; i < 4; i++)
    {
        syswaitpid(pidArray[i]);
        printf("pid terminated:");
        printInteger((int)pidArray[i]);
        printf("\n");
    }

    sysexit(); // after all children terminated, exit.
}
```

It loads collatz into process table, and then it blocks itself. It will be unblocked when 3 timer interrupt occurs(in pdf 5 is said but collatz sometimes runs so fast and terminates before 5th interrupt) . After that it loads the other processes and starts waiting for them.

```
case 99:
    runnningProcess->SetState(Process::Blocked);
    // ProcessManager::blockedList.add(runnningProcess);
    esp = (uint32_t)interruptManager->processManager->Schedule(cpu);

    break;
```

Sysblock Works like this. It changes the state of process and schedules to put another process in cpu.

```
if (interrupt == hardwareInterruptOffset)
{

    InterruptManager::timerInterruptCount++;
    CanRun::canRun = false;
    if (InterruptManager::timerInterruptCount == 3)
    {
        Process *init = ProcessManager::GetProcessByPid(0);
        init->SetState(Process::Ready);
        ProcessManager::readyQueue.enqueue(init);
    }
    esp = (uint32_t)processManager->Schedule((CPUState *)esp);
}
```

This runs at every timer interrupt, so when three timer interrupt happens, the init process is added to ready queue.

This is Schedule function it is different from the previous ones because the scheduling is no more round robin. Actually lt acts like round robin if all the processes have the same priority. I thought it would be better design this way. If higher priority process is available, it will run, but if there are more than one process sharing the highest priority, then cpu will Schedule between them in a round robin way.

```
CPUState *ProcessManager::Schedule(CPUState *cpustate)
{

    clearScreen();
    printf("*****************************SCHEDULING*****************************\n");
    UpdateWaitingProcesses();

    if (readyQueue.size <= 0)
    {
        printf("|------------------PROCESS TABLE------------------|\n");
        for (int i = 0; i < numProcess; i++)
        {
            processTable[i].PrintProcess();
        }
        printf("|-------------------------------------------------|\n");

        return cpustate;
    }

    if (runningProcess == nullptr)
    {
        printf("running process null\n");
        runningProcess = readyQueue.dequeue();
        runningProcess->SetState(Process::Running);
        return runningProcess->cpustate;
    }

    runningProcess->cpustate = cpustate;
    if (runningProcess->state == Process::Running)
    {
        runningProcess->SetState(Process::Ready);
        readyQueue.enqueue(runningProcess);
        runningProcess = nullptr;
    }

    runningProcess = readyQueue.dequeue();
    runningProcess->SetState(Process::Running);
    printf("|------------------PROCESS TABLE------------------|\n");
    for (int i = 0; i < numProcess; i++)
    {
        processTable[i].PrintProcess();
    }
    printf("|-------------------------------------------------|\n");
    readyQueue.printQueue();
    return runningProcess->cpustate;
}
```

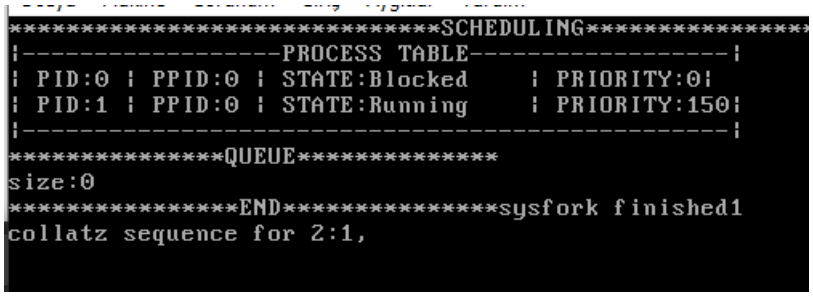
UpdateWaitingProcesses function make sure no process left waiting.

```
for (int k = 0; k < blockedList.size; k++)
{
    if (blockedList.elementAt(k)->state != Process::Waiting)
    {
        continue;
    }
    uint32_t waitpid = blockedList.elementAt(k)->waitpid;
    // if waitpid parameter is 0, then check for any terminated child
    if (waitpid <= 0)
    {
        // skip init, it cannot be child
        for (int i = 1; i < numProcess; i++)
        {
            if (processTable[i].ppid == blockedList.elementAt(k)->pid && processTable[i].state == Process::Terminated)
            {
                processTable[i].ppid = -1; // update ppid to not wait it again
                blockedList.elementAt(k)->state = Process::Ready;
                readyQueue.enqueue(blockedList.removeAt(k));
            }
        }
    }
    else if (waitpid > 0)
    {
        if (processTable[waitpid].state == Process::Terminated)
        {
            processTable[waitpid].ppid = -1;
            blockedList.elementAt(k)->state = Process::Ready;
            readyQueue.enqueue(blockedList.removeAt(k));
        }
    }
}
```

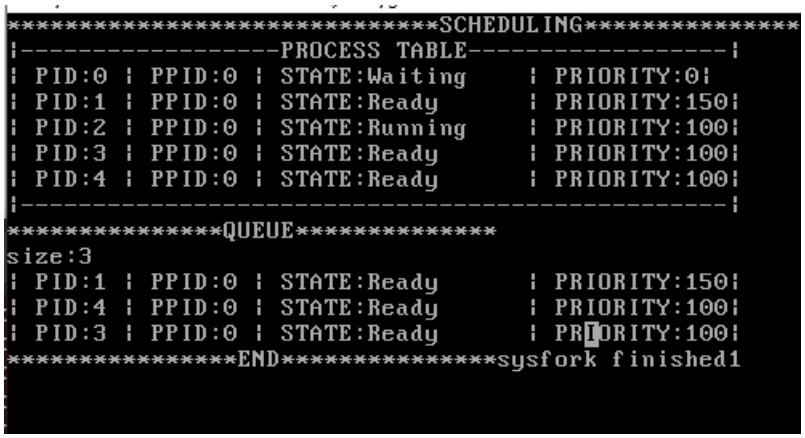
The same logic as partA but this time the processes in blockedList should be checked.

RESULTS:

Collatz is loaded and the init is blocked so that other processes cannot be loaded

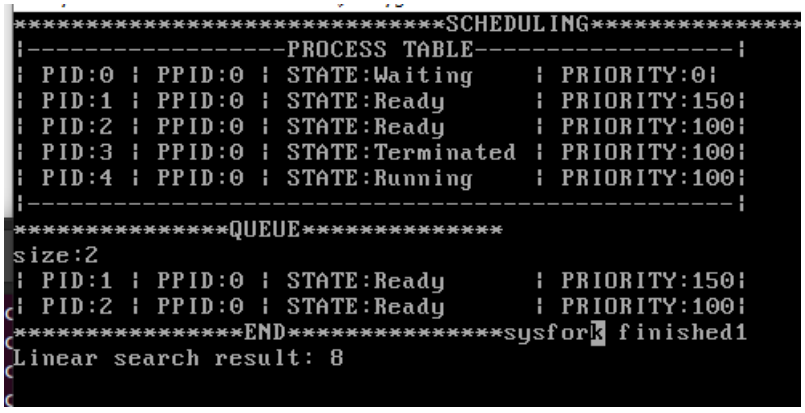


After third interrupt, other processes are loaded and as you can see collatz (pid is 1) is at the end of the queue

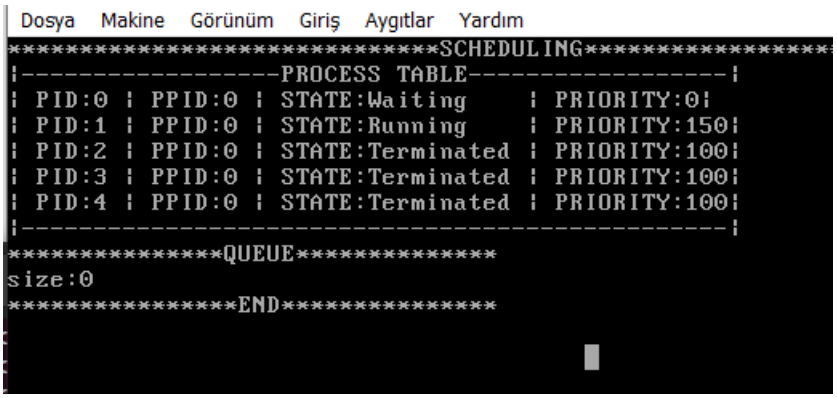


Scheduling occurs but collatz is still at the end of the queue because it has lower priority

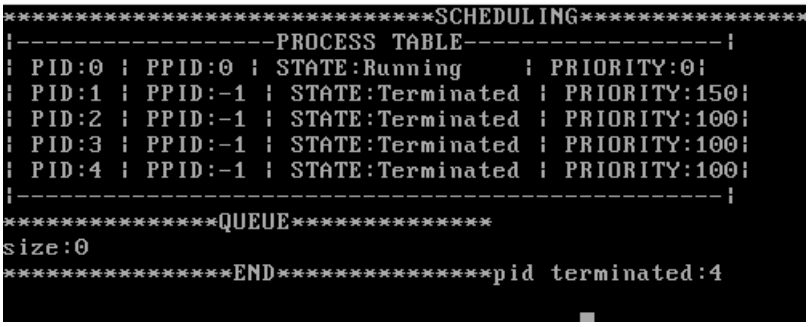
Processes keep executing and scheduling between them but collatz cannot execute



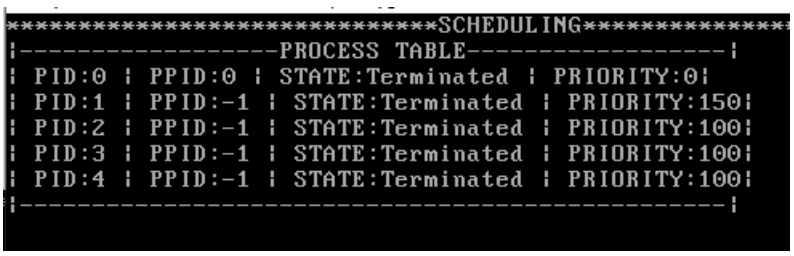
All processes terminated, collatz can execute



All processes terminated, init is able to run



İnit is terminated as well



PART B_4 Dynamic Priority Strategy:

For this part I am a little confused about what to do because there are a few ambiguous things in the pdf so firstly i will explain what i understand from the pdf. The Word priority is used as low and high but it is said that “We must consider lowest number as highest priority” so when i say high priority i mean the priority not the priority number which should be low in this case.

Collatz program should be in the ready queue alone at the start with high priority (low priority number). Because we want to run it for some time. And after some time, other processes are loaded all having the same priority number. If collatz has been running for some time for example 5 consecutive time interrupts, then its priority decreased (priority number increased) in this way other processes get to run for some time. Because they all have the same priority, they are scheduled in round robin way. After some time (for example 5th interrupt) collatz’s priority increased (priority number decreased) and then collatz gets to run. And this continues till all the programs executed. So, other processes’ priority stay same throughout the execution as it is requested in the pdf and the priority of collatz is dynamically adjusted. I used same ready queue as in third strategy.

```
    Process *collatz;
    collatz = processManager->GetProcessByPid(1);
    // collatz is terminated no need to adjust its priority
    if (collatz->GetState() != Process::Terminated)
    {
        // Adjust dynamic scheduling
        if (collatz->GetState() == Process::Running)
        {
            processManager->collatzRunCount++;
        }
        else if (collatz->GetState() == Process::Ready)
        {
            processManager->collatzRunCount--;
        }
        // if collatz has been running for some time then increase its priority number (decrease its priority)
        if (processManager->collatzRunCount >= 3)
        {
            collatz->SetPriority(110);
            // reset count
            processManager->collatzRunCount = 0;
        }
        // if collatz has been running for some time then decrease its priority number (increase its priority)
        else if (processManager->collatzRunCount <= -3)
        {
            collatz->SetPriority(90);
            // remove it from queue
            processManager->readyQueue.removePid(collatz->GetPid());
            // and add to queue again in this way it will be the first in the queue because of its priority
            processManager->readyQueue.enqueue(collatz);
            // reset count
            processManager->collatzRunCount = 0;
        }
    }

    esp = (uint32_t)processManager->Schedule((CPUState *)esp);
}
```

This is the part where dynamic scheduling occurs. Pid of collatz is 1 because it is loaded firstly. At each timer interrupt this code runs and adjusts the priority. If collatz has been running for long time, then its priority is decreased and collatzRunCount is reset. If collatz has been ready for long time, then it's priority is increased and it is removed from ready queue and enqueued again the reason i do it this way is it will be find its place automatically when enqueued according to its priority number.

The other parts are the same as the third strategy.

RESULTS

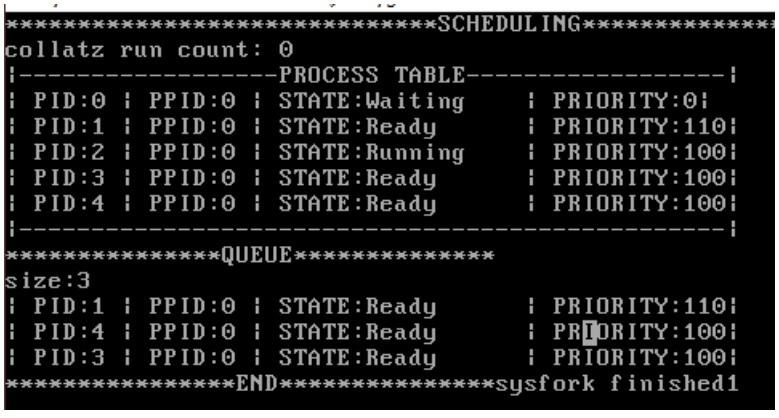
```
**********************************SCHEDULING***************
collatz run count: 0
¦------------------PROCESS TABLE------------------¦
¦ PID:0 ¦ PPID:0 ¦ STATE:Blocked    ¦ PRIORITY:0¦
¦ PID:1 ¦ PPID:0 ¦ STATE:Running    ¦ PRIORITY:90¦
¦-------------------------------------------------¦
***************QUEUE***************
size:0
*****************END****************sysfork finished1
collatz sequence for 2:1,
```


**********SCHEDULING**********
collatz run count: 1
----------PROCESS TABLE----------
| PID:0 | PPID:0 | STATE:Blocked | PRIORITY:0|
| PID:1 | PPID:0 | STATE:Running | PRIORITY:90|

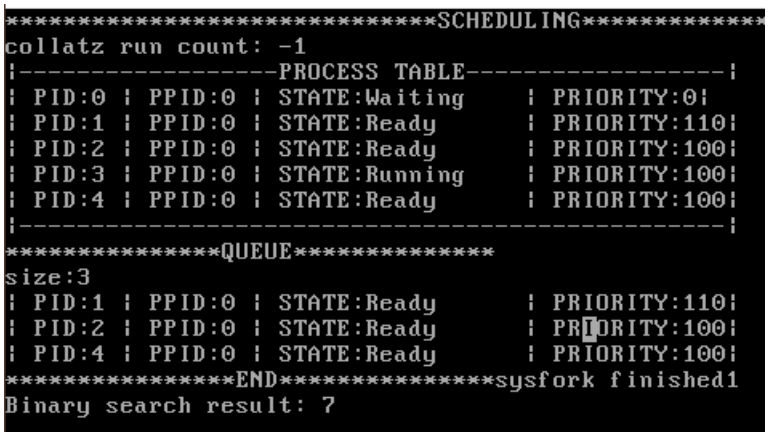

**********SCHEDULING**********
collatz run count: 2
----------PROCESS TABLE----------
| PID:0 | PPID:0 | STATE:Blocked | PRIORITY:0|
| PID:1 | PPID:0 | STATE:Running | PRIORITY:90|


**********SCHEDULING**********
collatz run count: 3
----------PROCESS TABLE----------
| PID:0 | PPID:0 | STATE:Blocked | PRIORITY:0|
| PID:1 | PPID:0 | STATE:Running | PRIORITY:90|

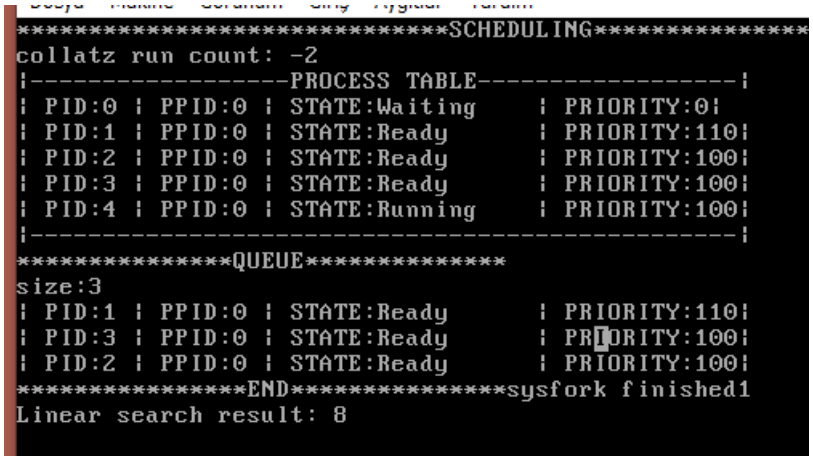

**********SCHEDULING**********
collatz run count: 4
----------PROCESS TABLE----------
| PID:0 | PPID:0 | STATE:Blocked | PRIORITY:0|
| PID:1 | PPID:0 | STATE:Running | PRIORITY:90|

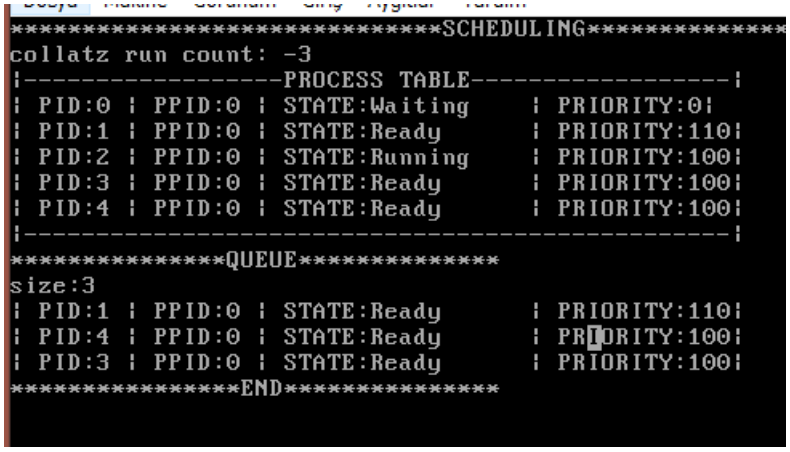

**********SCHEDULING**********
collatz run count: 0
----------PROCESS TABLE----------
PID:0	PPID:0	STATE:Waiting	PRIORITY:0
PID:1	PPID:0	STATE:Ready	PRIORITY:110
PID:2	PPID:0	STATE:Running	PRIORITY:100
PID:3	PPID:0	STATE:Ready	PRIORITY:100
PID:4	PPID:0	STATE:Ready	PRIORITY:100

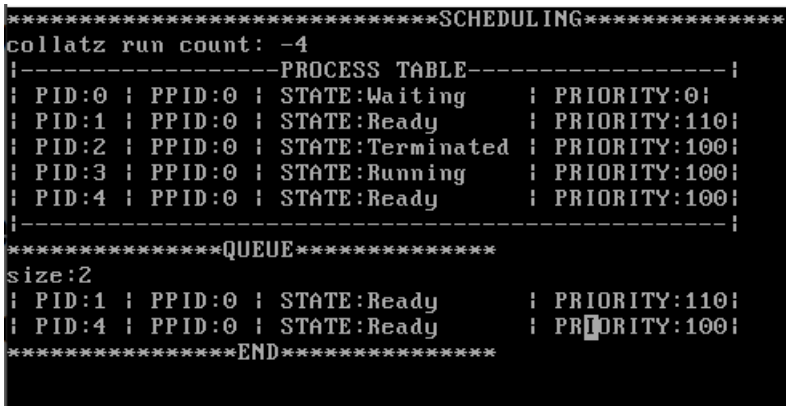
**********QUEUE**********
size:3
PID:1	PPID:0	STATE:Ready	PRIORITY:110
PID:4	PPID:0	STATE:Ready	PRIORITY:100
PID:3	PPID:0	STATE:Ready	PRIORITY:100
**********END**********sysfork finished1

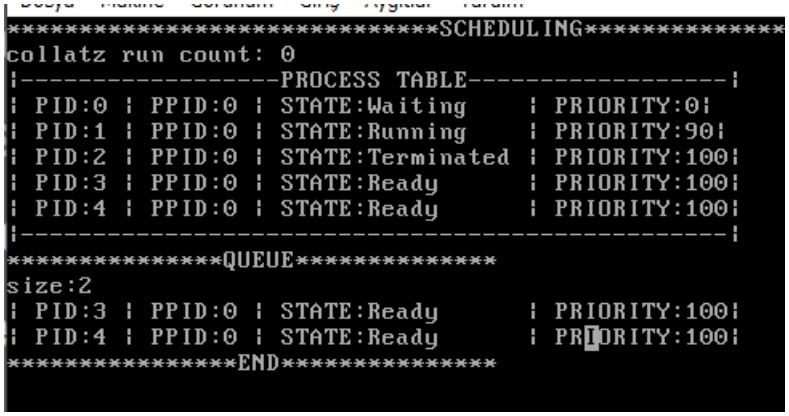

**********SCHEDULING**********
collatz run count: -1
----------PROCESS TABLE----------
PID:0	PPID:0	STATE:Waiting	PRIORITY:0
PID:1	PPID:0	STATE:Ready	PRIORITY:110
PID:2	PPID:0	STATE:Ready	PRIORITY:100
PID:3	PPID:0	STATE:Running	PRIORITY:100
PID:4	PPID:0	STATE:Ready	PRIORITY:100

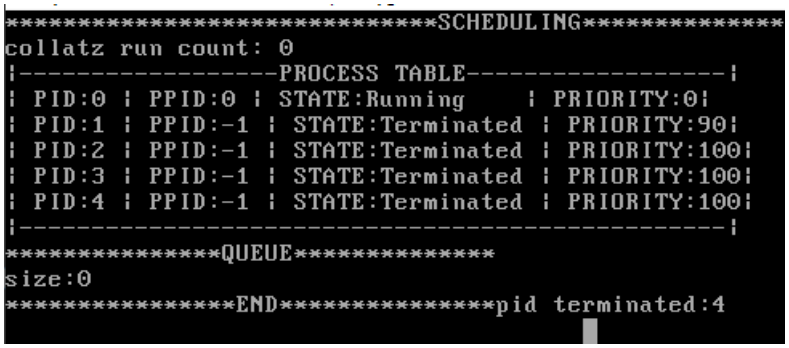
**********QUEUE**********
size:3
PID:1	PPID:0	STATE:Ready	PRIORITY:110
PID:2	PPID:0	STATE:Ready	PRIORITY:100
PID:4	PPID:0	STATE:Ready	PRIORITY:100
**********END**********sysfork finished1
Binary search result: 7

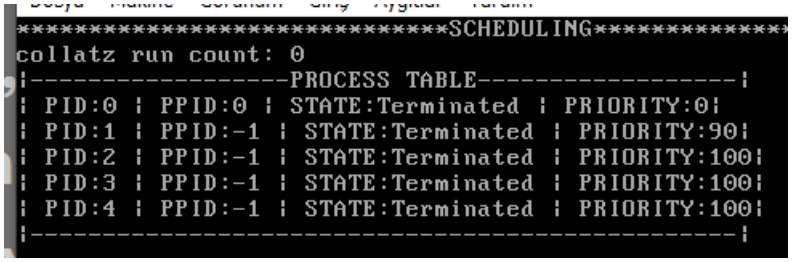

**********SCHEDULING**********
collatz run count: -2
¦----------PROCESS TABLE----------¦
¦ PID:0 ¦ PPID:0 ¦ STATE:Waiting ¦ PRIORITY:0¦
¦ PID:1 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:110¦
¦ PID:2 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:100¦
¦ PID:3 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:100¦
¦ PID:4 ¦ PPID:0 ¦ STATE:Running ¦ PRIORITY:100¦
¦----------¦
**********QUEUE**********
size:3
¦ PID:1 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:110¦
¦ PID:3 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:100¦
¦ PID:2 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:100¦
**********END**********sysfork finished1
Linear search result: 8


**********SCHEDULING**********
collatz run count: -3
¦----------PROCESS TABLE----------¦
¦ PID:0 ¦ PPID:0 ¦ STATE:Waiting ¦ PRIORITY:0¦
¦ PID:1 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:110¦
¦ PID:2 ¦ PPID:0 ¦ STATE:Running ¦ PRIORITY:100¦
¦ PID:3 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:100¦
¦ PID:4 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:100¦
¦----------¦
**********QUEUE**********
size:3
¦ PID:1 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:110¦
¦ PID:4 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:100¦
¦ PID:3 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:100¦
**********END**********


**********SCHEDULING**********
collatz run count: -4
¦----------PROCESS TABLE----------¦
¦ PID:0 ¦ PPID:0 ¦ STATE:Waiting ¦ PRIORITY:0¦
¦ PID:1 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:110¦
¦ PID:2 ¦ PPID:0 ¦ STATE:Terminated ¦ PRIORITY:100¦
¦ PID:3 ¦ PPID:0 ¦ STATE:Running ¦ PRIORITY:100¦
¦ PID:4 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:100¦
¦----------¦
**********QUEUE**********
size:2
¦ PID:1 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:110¦
¦ PID:4 ¦ PPID:0 ¦ STATE:Ready ¦ PRIORITY:100¦
**********END**********


**********SCHEDULING**********
collatz run count: 0
PROCESS TABLE
PID:0 | PPID:0 | STATE:Waiting | PRIORITY:0
PID:1 | PPID:0 | STATE:Running | PRIORITY:90
PID:2 | PPID:0 | STATE:Terminated | PRIORITY:100
PID:3 | PPID:0 | STATE:Ready | PRIORITY:100
PID:4 | PPID:0 | STATE:Ready | PRIORITY:100
**********QUEUE**********
size:2
PID:3 | PPID:0 | STATE:Ready | PRIORITY:100
PID:4 | PPID:0 | STATE:Ready | PRIORITY:100
**********END**********


**********SCHEDULING**********
collatz run count: 0
PROCESS TABLE
PID:0 | PPID:0 | STATE:Running | PRIORITY:0
PID:1 | PPID:-1 | STATE:Terminated | PRIORITY:90
PID:2 | PPID:-1 | STATE:Terminated | PRIORITY:100
PID:3 | PPID:-1 | STATE:Terminated | PRIORITY:100
PID:4 | PPID:-1 | STATE:Terminated | PRIORITY:100
**********QUEUE**********
size:0
**********END**********pid terminated:4


**********SCHEDULING**********
collatz run count: 0
PROCESS TABLE
PID:0 | PPID:0 | STATE:Terminated | PRIORITY:0
PID:1 | PPID:-1 | STATE:Terminated | PRIORITY:90
PID:2 | PPID:-1 | STATE:Terminated | PRIORITY:100
PID:3 | PPID:-1 | STATE:Terminated | PRIORITY:100
PID:4 | PPID:-1 | STATE:Terminated | PRIORITY:100