

CSE312 HW1 REPORT

COŞKUN HASAN ŞALTU

1801042631

1) Problem Solution Approach

The purpose of this project is to write an operating system that supports multiprocessing. For this, a ready-made operating system whose source code belongs to Viktor Engelmann is used. Additions have been made to support multiprocessing and support some POSIX system calls. The operating system is coded using C++ and of course assembly. The system contains interrupt algorithms, scheduling algorithm so, it supports multiprocessing. There are 2 class structure used for multiprocessing. They are Process and ProcessTable.

Process

The Process class represents a single process in an operating system. It has private data members including a stack array of size 4096 bytes, a pointer to CPUState, an integer PID, an integer PPID (parent process ID), and an integer state. The class also has a static data member nextpid which is used to assign the next available PID to a new process. Constructor takes a GlobalDescriptorTable pointer and an entry point function pointer as arguments and creates a new process with a new stack and CPU state.

```
class Process
{
    friend class ProcessTable;

private:
    common::uint8_t stack[4096]; // 4 KiB
    CPUState *cpustate;
    int pid;
    int ppid;
    int state;
    static int nextpid;

public:
    Process(GlobalDescriptorTable *gdt, void entrypoint());
    Process(GlobalDescriptorTable *gdt, void entrypoint(), int ppid);
    Process(CPUState *cpustate, int ppid);
    ~Process();
    int GetPID();
    int GetPPID();
    int GetState();
    void SetState(int state);
    void SetPPID(int ppid);
    CPUState *GetCPUState();
    void SetCPUState(CPUState *cpustate);
};
```

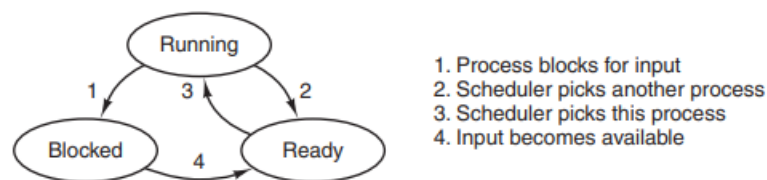
Process Class

The Process class also has several member functions:

- GetPID: Returns the PID of the process.
- GetPPID: Returns the parent process ID of the process.
- GetState: Returns the state of the process.
- SetState: Sets the state of the process to a given value.
- SetPPID: Sets the parent process ID of the process to a given value.
- GetCPUState: Returns a pointer to the CPU state of the process.
- SetCPUState: Sets the CPU state of the process to a given value.

The GlobalDescriptorTable (GDT) is a data structure that stores the global segment descriptors for the CPU in an operating system. The GDT is used to set up memory segments for various purposes such as code, data, and stack.

The entrypoint function in the given code is a function pointer that represents the starting point of the process code. When a process is created, the entrypoint function is passed to the constructor of the Process class, and it is used to set up the CPU state of the process. The CPU state includes the instruction pointer (IP) and the stack pointer (SP), which are set to the entry point and the stack of the process respectively. When the process is scheduled to run by the operating system, the CPU starts executing instructions from the entry point of the process.



Process States

The process state should be READY, RUNNING and BLOCKED. Any process's state turns READY when process is created. State turns RUNNING when the process is added to process table and scheduling algorithm. State turns BLOCKED when the process is terminated.

```

Process processA(&gdt, BinarySearch);
Process processB(&gdt, LinearSearch);
Process processC(&gdt, Collatz);

```

Examples Create Process

To implement the process model, the operating system maintains a table (an array of structures), called the process table, with one entry per process. (Some authors call these entries process control blocks.) This entry contains important information about the process' state, including its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from running to ready or blocked state so that it can be restarted later as if it had never been stopped.

Process Table

The ProcessTable class represents a table of processes in an operating system. The class has private data members including an array of process pointers, an integer to store the number of processes, an integer to store the index of the current process, and a boolean flag to indicate whether scheduling is allowed or not.

```

class ProcessTable
{
private:
    Process *processes[256];
    int numProcesses;
    int currentProcess;
    bool schedulingFlag;

public:
    ProcessTable();
    ~ProcessTable();
    bool AddProcess(Process *process);
    Process *GetProcess(int pid);
    bool UpdateProcessState(int pid, int state);
    CPUState *Schedule(CPUState *cpustate);
    Process *GetCurrentProcess();
    int GetCurrentProcessID();
    void StopScheduling();
    void StartScheduling();
    int GetNumProcesses();
};

```

Process Table Class

The ProcessTable class has several member functions:

- **AddProcess:** Adds a process to the process array.
- **GetProcess:** Returns a process with a given PID.
- **UpdateProcessState:** Updates the state of a process with a given PID.
- **Schedule:** Selects the next process to run based on the scheduling algorithm implemented in the operating system and returns the CPUState pointer of the next process to run.
- **GetCurrentProcess:** Returns a pointer to the current process.
- **GetCurrentProcessID:** Returns the PID of the current process.
- **StopScheduling:** Disables the scheduling flag, indicating that the scheduler should not allow processes to run.
- **StartScheduling:** Enables the scheduling flag, indicating that the scheduler should allow processes to run.
- **GetNumProcesss:** Returns the number of processes in the process array.

Scheduling

In round-robin scheduling, each process is given a fixed time slice to execute on the CPU. Once the time slice expires, the scheduler selects the next process in a circular manner, regardless of the process priority. The Schedule function in the ProcessTable class implements the round-robin scheduling algorithm by selecting the next running process in a circular fashion and returning its CPU state pointer, which is used to switch context to that process and allow it to run on the CPU for its time slice.

```

CPUState *ProcessTable::Schedule(CPUState *cpustate)
{
    if (!schedulingFlag)
        return cpustate;
    if (numProcesses <= 0)
        return cpustate;

    if (currentProcess >= 0)
    {
        processes[currentProcess]->cpustate = cpustate;
    }
    if (++currentProcess >= numProcesses)
    {
        currentProcess %= numProcesses;
    }
    while (processes[currentProcess]->GetState() != PROCESS_STATE_RUNNING)
    {
        if (++currentProcess >= numProcesses)
        {
            currentProcess %= numProcesses;
        }
    }
    return processes[currentProcess]->cpustate;
}

```

Scheduling Algorithm

System Calls

WaitPid

The waitpid system call takes a process ID as an argument and checks if the corresponding process is in a blocked state. If the process is blocked, indicating that it is currently waiting for an event, waitpid returns 0. Otherwise, it returns -1 to indicate that the process is not blocked. The syswaitpid function is a wrapper around the waitpid system call that invokes the system call using an interrupt instruction (int \$0x80) and returns the value of the eax register after the interrupt has been handled, which contains the return value of waitpid.

```

int waitpid(int pid)
{
    if (processTable.GetProcess(pid)->GetState() == PROCESS_STATE_BLOCKED)
    {
        return 0;
    }
    else
    {
        return -1;
    }
}

int syswaitpid(int pid)
{
    asm("int $0x80"
        : "a"(7), "b"(pid));
    int eax = 0;
    asm("mov %%eax, %0"
        : "=r"(eax));
    return eax;
}

```

Waitpid System Call

Fork

The fork system call creates a new process by duplicating the calling process, called the parent process, resulting in the creation of a new child process. The child process is an exact copy of the parent process, except for its unique process ID and the fact that it has a different memory space. The fork system call takes no arguments and returns the process ID of the child process to the parent process and 0 to the child process. The new process is placed at the end of the process table and is eligible for scheduling by the operating system. The parent process and the child process execute independently of each other, each with their own CPU state and program counter.

```
int fork(CPUState *cpu, int index)
{
    processTable.StopScheduling();
    int pid = processTable.GetCurrentProcess()->GetPID();
    processArray[index]->SetCPUState(cpu);
    processArray[index]->SetPPID(pid);
    processTable.AddProcess(processArray[index]);
    processTable.StartScheduling();
    return processArray[index]->GetPID();
}

int sysfork()
{
    asm("int $0x80"
        :
        : "a"(2), "b"(index));

    // read eax register and return the value
    int eax = 0;
    asm("mov %%eax, %0"
        : "=r"(eax));
    return eax;
}
```

Fork System Call

2) Test Cases

The project has been tested with 3 different program: Binary Search, Linear Search and Collatz:

```
void BinarySearch()
{
    int arr[10] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170};
    int key = 110, start = 0, end = 9, mid;
    int pid = processTable.GetCurrentProcess()->GetPID();
    bool found = false;
    while (1)
    {
        while (start <= end)
        {
            mid = (start + end) / 2;
            if (arr[mid] == key)
            {
                // Element found
                found = true;
            }
            else if (arr[mid] < key)
            {
                // Look in the right half
                start = mid + 1;
            }
            else
            {
                // Look in the left half
                end = mid - 1;
            }
        }
        if (!found)
        {
            printf("Element found at index: ");
            printInt(mid);
            printf("\n");
        }
        else
        {
            printf("Element found in the array \n");
        }
        for (int i = 0; i < 1000000000; i++)
        {
            ;
        }
        processTable.UpdateProcessState(pid, PROCESS_STATE_BLOCKED);
    }
}
```

Binary Search Program

```
void LinearSearch()
{
    int pid = processTable.GetCurrentProcess()->GetPID();
    int arr[10] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170};
    int key = 110;
    bool found = false;
    int i;
    while (1)
    {
        for (i = 0; i < 10; i++)
        {
            if (arr[i] == key)
            {
                found = true;
                break;
            }
        }
        if (!found)
        {
            printf("Element not found in the array \n");
        }
        else
        {
            printf("Element found at index: ");
            printInt(i);
            printf("\n");
        }
        for (int i = 0; i < 1000000000; i++)
        {
            ;
        }
        processTable.UpdateProcessState(pid, PROCESS_STATE_BLOCKED);
    }
}
```

Linear Search Program


```

void Collatz()
{
    int pid = processTable.GetCurrentProcess()->GetPID();
    int num = 10;

    while (1)
    {
        while (num != 1)
        {
            if (num % 2 == 0)
            {
                num /= 2;
            }
            else
            {
                num = num * 3 + 1;
            }
            printInt(num);
            printf(" ");
        }
        printf("\n");
        for (int i = 0; i < 100000000; i++)
        {
            ;
        }
        processTable.UpdateProcessState(pid, PROCESS_STATE_BLOCKED);
    }
}

```

Collatz Program

The project has been tested with 3 different kernel methods:

In the first strategy init process will initialize Process Table, load 3 different programs (listed below) to the memory start them and will enter an infinite loop until all the processes terminate.

Second strategy is randomly choosing one of the programs and loads it into memory 10 times (Same ll program 10 different processes), start them and will enter an infinite loop until all the processes terminate.

Final Strategy is choosing 2 out 3 programs randomly and loading each program 3 times start them and will enter an infinite loop until all the processes terminate.

```

Process processA(&gdt, BinarySearch);
Process processB(&gdt, LinearSearch);
Process processC(&gdt, Collatz);
void firstStrategy()
{
    int num = 0;
    processTable.AddProcess(&processA);
    processTable.AddProcess(&processB);
    processTable.AddProcess(&processC);
    int pidA = processA.GetPID();
    int pidB = processB.GetPID();
    int pidC = processC.GetPID();
    bool terminated = false;
    num = processTable.GetNumProcesses();
    while (1)
    {
        if (terminated)
        {
            sysprintf("Process Number: ");
            printInt(num);
            sysprintf("      All processes terminated\n");
            break;
        }
        else
        {
            sysprintf("All processes running\n");
            if (syswaitpid(pidA) == 0)
            {
                sysprintf("Binary Search terminated\n");
                terminated = true;
            }
            else
            {
                sysprintf("Binary Search running\n");
                terminated = false;
            }
            if (syswaitpid(pidB) == 0)
            {
                sysprintf("Linear Search terminated\n");
                terminated = true;
            }
            else
            {
                sysprintf("Linear Search running\n");
                terminated = false;
            }
            if (syswaitpid(pidC) == 0)
            {
                sysprintf("Collatz terminated\n");
                terminated = true;
            }
            else
            {
                sysprintf("Collatz running\n");
                terminated = false;
            }
        }
    }
}

```

First Strategy

```

void secondStrategy()
{
    int num = 0;
    bool terminated = false;
    int index = 0;
    for (int i = 0; i < 10; i++)
    {
        *processArray[i] = Process(&gdt, LinearSearch);
    }
    for (int i = 0; i < 10; i++)
    {
        index = i;
        sysfork();
    }
    num = processTable.GetNumProcesses();
    while (1)
    {
        if (terminated)
        {
            sysprintf("Process Number: ");
            printInt(num);
            sysprintf("      All processes terminated\n");
            break;
        }
        else
        {
            sysprintf("All processes running\n");
            for (int i = 0; i < 10; i++)
            {
                if (syswaitpid(processArray[i]->GetPID()) == 0)
                {
                    sysprintf("Linear Search terminated\n");
                    terminated = true;
                }
                else
                {
                    sysprintf("Linear Search running\n");
                    terminated = false;
                }
            }
        }
    }
}

```

Second Strategy

```

void finalStrategy()
{
    int num = 0;
    bool terminated = false;
    int index = 0;
    for (int i = 0; i < 6; i++)
    {
        if (i < 3)
            *processArray[i] = Process(&gdt, LinearSearch);
        else
            *processArray[i] = Process(&gdt, BinarySearch);
    }
    for (int i = 0; i < 6; i++)
    {
        index = i;
        sysfork();
    }
    num = processTable.GetNumProcesss();
    while (1)
    {
        if (terminated)
        {
            sysprintf("Process Number: ");
            printInt(num);
            sysprintf("      All processes terminated\n");
            break;
        }
        else
        {
            sysprintf("All processes running\n");
            for (int i = 0; i < 6; i++)
            {
                if (syswaitpid(processArray[i]->GetPID()) == 0)
                {
                    terminated = true;
                }
                else
                {
                    terminated = false;
                }
            }
        }
    }
}

```

Final Strategy

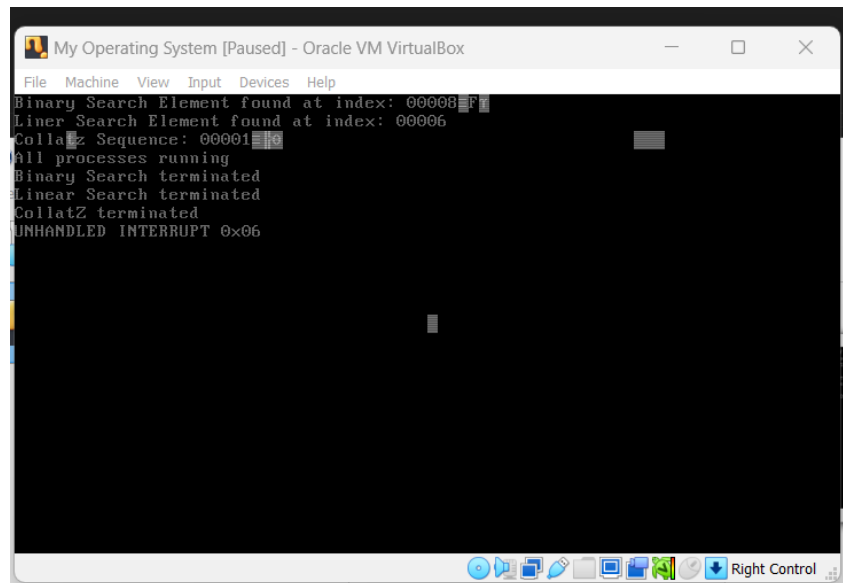
The first strategy is a program that creates three processes (processA, processB, and processC) and waits for each process to terminate before printing a message indicating whether it has terminated or is still running. The program uses a loop that checks whether each process has terminated and sets a flag to true if it has. If all processes have terminated, the loop ends and the program prints a message indicating that all processes have terminated.

The second strategy is a program that creates ten instances of the Process class with the LinearSearch function pointer, and adds them to an array processArray. The program then enters a loop that forks a new process for each instance in processArray. This means that a total of ten processes will be created, each executing the LinearSearch function. The program then enters a loop that checks whether each process has terminated, using the syswaitpid function. If a

process has terminated, the program sets a flag terminated to true. If all processes have terminated, the loop ends and the program prints a message indicating that all processes have terminated. Otherwise, the loop continues and prints a message indicating which processes are still running. The program prints out a message for each process indicating whether it is still running or has terminated. The messages indicate which process is running or has terminated and are printed to the console using the `sysprintf` function. Overall, the second strategy creates multiple instances of the Process class, forks a new process for each instance, and manages them using a loop that waits for each process to terminate before printing out a message indicating whether the process has terminated or is still running.

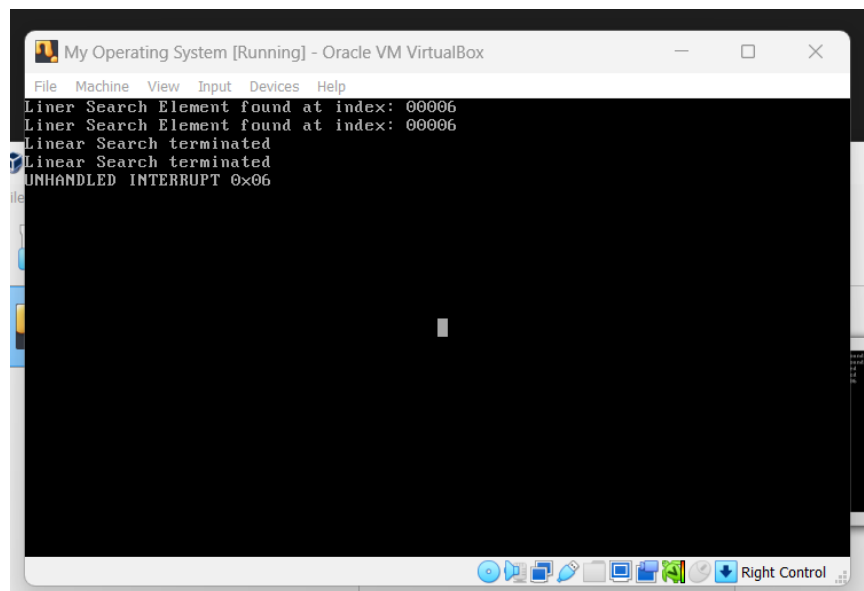
The final strategy is a program that creates six instances of the Process class, with three instances running the LinearSearch function and three instances running the BinarySearch function, and adds them to an array `processArray`. The program then enters a loop that forks a new process for each instance in `processArray`. This means that a total of six processes will be created, with three executing the LinearSearch function and three executing the BinarySearch function. The program then enters a loop that checks whether each process has terminated, using the `syswaitpid` function. If a process has terminated, the program sets a flag terminated to true. If all processes have terminated, the loop ends and the program prints a message indicating that all processes have terminated. Otherwise, the loop continues and prints a message indicating which processes are still running. The program prints out a message for each process indicating whether it is still running or has terminated. The messages indicate which process is running or has terminated and are printed to the console using the `sysprintf` function. Overall, the final strategy creates multiple instances of the Process class, forks a new process for each instance, and manages them using a loop that waits for each process to terminate before printing out a message indicating whether the process has terminated or is still running. This strategy includes a mix of processes running different functions, with three executing LinearSearch and three executing BinarySearch.

Outputs



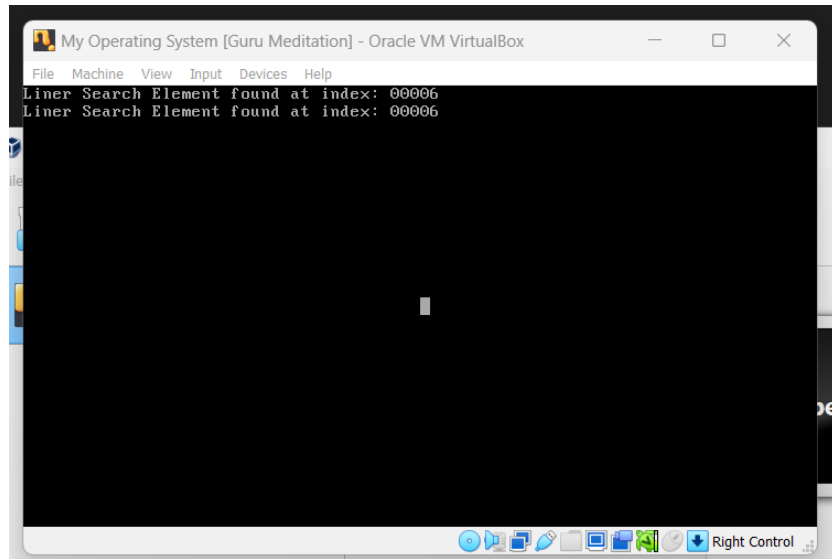
```
My Operating System [Paused] - Oracle VM VirtualBox
File Machine View Input Devices Help
Binary Search Element found at index: 00008
Linear Search Element found at index: 00006
Collatz Sequence: 00001
All processes running
Binary Search terminated
Linear Search terminated
Collatz terminated
UNHANDLED INTERRUPT 0x06
```

First Strategy Output

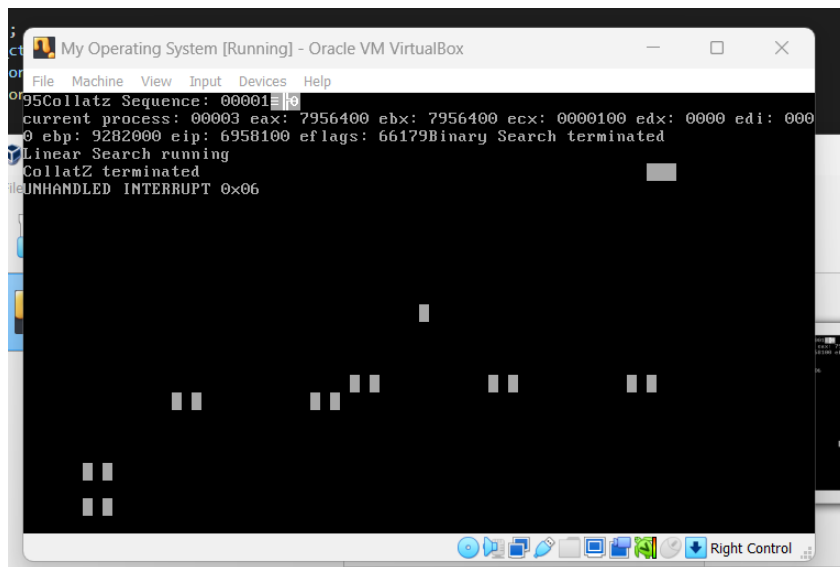


```
My Operating System [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Linear Search Element found at index: 00006
Linear Search Element found at index: 00006
Linear Search terminated
Linear Search terminated
UNHANDLED INTERRUPT 0x06
```

Second Strategy Output



Third Strategy Output



Cpu State Output (When Scheduling)