

GEBZE TECHNICAL UNIVERSITY
COMPUTER ENGINEERING FACULTY



CSE 312 OPERATING SYSTEMS
PROJECT REPORT

ENES AYSU
1901042671

Program Description:

Development of a Custom Operating System with POSIX System Call Implementation and Multi-Programming Capabilities

Objective:

The objective of this project is to develop a custom operating system (OS) that can efficiently handle multi-programming by implementing POSIX system calls, managing process scheduling, and handling interrupts. The OS will demonstrate its capabilities through the execution of multiple programs, including the calculation of Collatz sequences and a long-running computation task.

1- POSIX System Calls Implementation:

fork(): Create new processes.

waitpid(): Wait for process termination.

execve(): Execute a program. (not implemented)

Implement any additional necessary POSIX calls.

2- Process Creation:

All processes must be created using the fork() system call.

3- Loading Multiple Programs:

The kernel must be capable of loading and executing multiple programs into memory.

4- Multi-Programming Management:

Develop a Process Table to maintain information about the processes in memory.

The Process Table should include details such as process ID, state, program counter, register values, and memory pointers.

5- Interrupt Handling:

Integrate interrupt handling mechanisms into the kernel to manage context switches and respond to system events.

Implement Round Robin scheduling for process management, triggered by timer interrupts.

6- Lifecycle Management:

Each program will be loaded three times and executed in an infinite loop until all processes terminate.

7- Program Testing:

Collatz Sequence Program

Long Running Program

Implementation Overview

POSIX System Calls:

- Implement the required POSIX system calls in the kernel to manage process creation, execution, and synchronization.
- Ensure robust handling of system call parameters and return values.

Process Table:

- Design and implement a Process Table structure that includes fields for:
- Process ID (PID)
- Process state (running, waiting, terminated)
- Program counter (PC)
- Register values
- Memory pointers
- Scheduling information
- Update the Process Table during process creation, execution, and termination.

Interrupt Handling and Scheduling:

- Integrate a timer interrupt to trigger context switches.
- Implement Round Robin scheduling to ensure fair CPU time distribution among processes.
- Print the process table information during each context switch for debugging and monitoring.

Program Execution:

- Load and execute the Collatz sequence program and the long-running program multiple times.
- Ensure correct input handling and output display for the Collatz sequence program.
- Demonstrate the OS's ability to handle computationally intensive tasks with the long-running program.

Testing and Validation:

- Validate the correct implementation of POSIX system calls through unit tests.
- Test the multi-programming capabilities by running the Collatz sequence and long-running programs concurrently.
- Ensure proper context switching and process management through continuous monitoring and debugging output.

Implementation Details

```
common::uint32_t Task::getId() {  
    return pid;  
}
```

This initializes the static member `pidCounter` which keeps track of the task IDs.

```
TaskManager::TaskManager()  
{  
    numTasks = 0;  
    currentTask = -1;  
}
```

Initializes the `TaskManager` by setting the number of tasks to zero and the current task index to -1 (indicating no current task).

```

common::uint32_t TaskManager::ForkTask(CPUState *cpustate)
{
    if(numTasks >= 256) {
        return -1;
    }

    tasks[numTasks].taskState = 2;
    tasks[numTasks].pPid = tasks[currentTask].pid;
    tasks[numTasks].pid = ++Task::pIdCounter;
    tasks[currentTask].cPid = tasks[numTasks].pid;

    // copying current stack to new stack
    for (int i = 0; i < sizeof(tasks[currentTask].stack); i++)
    {
        tasks[numTasks].stack[i] = tasks[currentTask].stack[i];
    }

    // finding current task's length
    common::uint32_t currentTaskOffset = (((common::uint32_t)cpustate) -
    ((common::uint32_t)tasks[currentTask].stack));
    // make two stacks pointer's location same
    tasks[numTasks].cpustate = (CPUState*)(tasks[numTasks].stack + currentTaskOffset);

    // child returns 0
    tasks[numTasks].cpustate -> eax = 0;

    // new task created
    numTasks++;

    return 0;
}

```

This function creates a new task by forking the current task. It copies the current task's stack to the new task's stack, sets the new task's state to ready, and updates the PID values. It returns 0 for the child task.

```
common::uint32_t TaskManager::GetPID() {  
    return tasks[currentTask].pid;  
}
```

Returns the child PID of the current task.

```
int TaskManager::getIndex(common::uint32_t pid)  
{  
    int index = -1;  
    for (int i = 0; i < numTasks; i++)  
    {  
        if(tasks[i].pid == pid)  
        {  
            index = i;  
            break;  
        }  
    }  
    return index;  
}
```

Finds and returns the index of a task given its PID. Returns -1 if the task is not found.

```
bool TaskManager::ExitTask() {  
    tasks[currentTask].taskState = 0;  
    return true;  
}
```

Marks the current task as finished.

```

bool TaskManager::WaitTask(common::uint32_t pid) {
    tasks[currentTask].taskState = 1;
    tasks[currentTask].waitPid = pid;
    return true;
}

```

Sets the current task to a waiting state and records the PID it is waiting for.

```

bool TaskManager::AddTask(Task* task) {
    if(numTasks >= 256) {
        return false;
    }

    tasks[numTasks].taskState = 2;
    tasks[numTasks].pid = ++Task::pIdCounter;
    tasks[numTasks].cpustate = (CPUState*)(tasks[numTasks].stack + 4096 -
sizeof(CPUState));

    tasks[numTasks].cpustate -> eax = task->cpustate->eax;
    tasks[numTasks].cpustate -> ebx = task->cpustate->ebx;
    tasks[numTasks].cpustate -> ecx = task->cpustate->ecx;
    tasks[numTasks].cpustate -> edx = task->cpustate->edx;

    tasks[numTasks].cpustate -> esi = task->cpustate->esi;
    tasks[numTasks].cpustate -> edi = task->cpustate->edi;
    tasks[numTasks].cpustate -> ebp = task->cpustate->ebp;

    tasks[numTasks].cpustate -> eip = task->cpustate->eip;
    tasks[numTasks].cpustate -> cs = task->cpustate->cs;
    tasks[numTasks].cpustate -> eflags = task->cpustate->eflags;
    tasks[numTasks].cpustate -> esp = task->cpustate->esp;
    //tasks[numTasks].cpustate -> ss = task->cpustate->ss;

    numTasks++;
}

```



```

    return true;
}

```

Adds a new task to the TaskManager, copying the CPU state from the provided task.

```

void TaskManager::taskTable(){
    printf("\n-----\n");
    printf("PID  PPID STATE\n");
    for (int i = 0; i < numTasks; i++)
    {
        printfHex(tasks[i].pid);
        printf(" ");
        printfHex(tasks[i].pPid);
        printf(" ");
        if(tasks[i].taskState== 2){
            if(i==currentTask)
                printf("RUNNING");
            else
                printf("READY");
        }else if(tasks[i].taskState==1){
            printf("WAITING");
        }else if(tasks[i].taskState==0){
            printf("FINISHED");
        }
        printf("\n");
    }
    printf("-----\n");
    for (int i = 0; i < 10000000; i++){
        printf("");
    }
}

```

Prints the task table showing each task's PID, parent PID, and state.

```

CPUState* TaskManager::Schedule(CPUState* cpustate)
{
    if (cpustate->eax == 6) {
        WaitTask(cpustate->ebx);
    }
    if(numTasks <= 0) {
        return cpustate;
    }
    if(currentTask >= 0) {
        tasks[currentTask].cpustate = cpustate;
    }

    int findTask = (currentTask + 1) % numTasks;
    while(tasks[findTask].taskState != 2) {
        if(tasks[findTask].taskState == 1) {
            int waitTaskIndex = 0;
            waitTaskIndex = getIndex(tasks[findTask].waitPid);
            if(waitTaskIndex > -1 && tasks[waitTaskIndex].taskState != 1) {
                if(tasks[waitTaskIndex].taskState == 0) {
                    printf("found new task\n");
                    tasks[findTask].waitPid = 0;
                    tasks[findTask].taskState = 2;
                    continue;
                } else if (tasks[waitTaskIndex].taskState == 2) {
                    printf("still searching\n");
                    findTask = waitTaskIndex;
                    continue;
                }
            }
        }
        findTask = (findTask + 1) % numTasks;
    }
    currentTask = findTask;
    return tasks[currentTask].cpustate;
}

```

This function is the core of the task scheduler. It determines which task to run next based on their states:

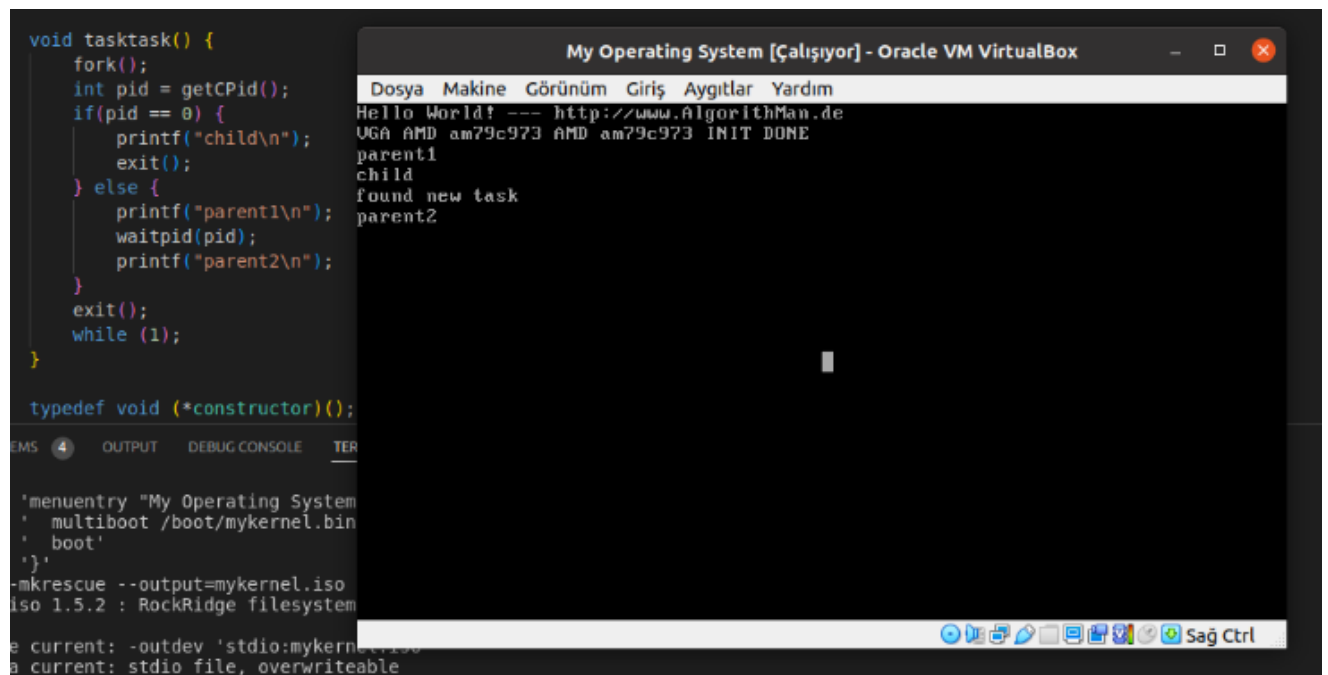
- It skips finished and waiting tasks.
- If a task is waiting for another task to finish, it checks the state of that task.
- It updates the currentTask and returns the CPUState of the next ready task.

Tests

For Compilation

\$ make

Basic Demonstration of Running Fork Function



```
void tasktask() {
    fork();
    int pid = getCPid();
    if(pid == 0) {
        printf("child\n");
        exit();
    } else {
        printf("parent1\n");
        waitpid(pid);
        printf("parent2\n");
    }
    exit();
    while (1);
}

typedef void (*constructor)();

'menuentry "My Operating System
'  multiboot /boot/mykernel.bin
'  boot'
'}
```

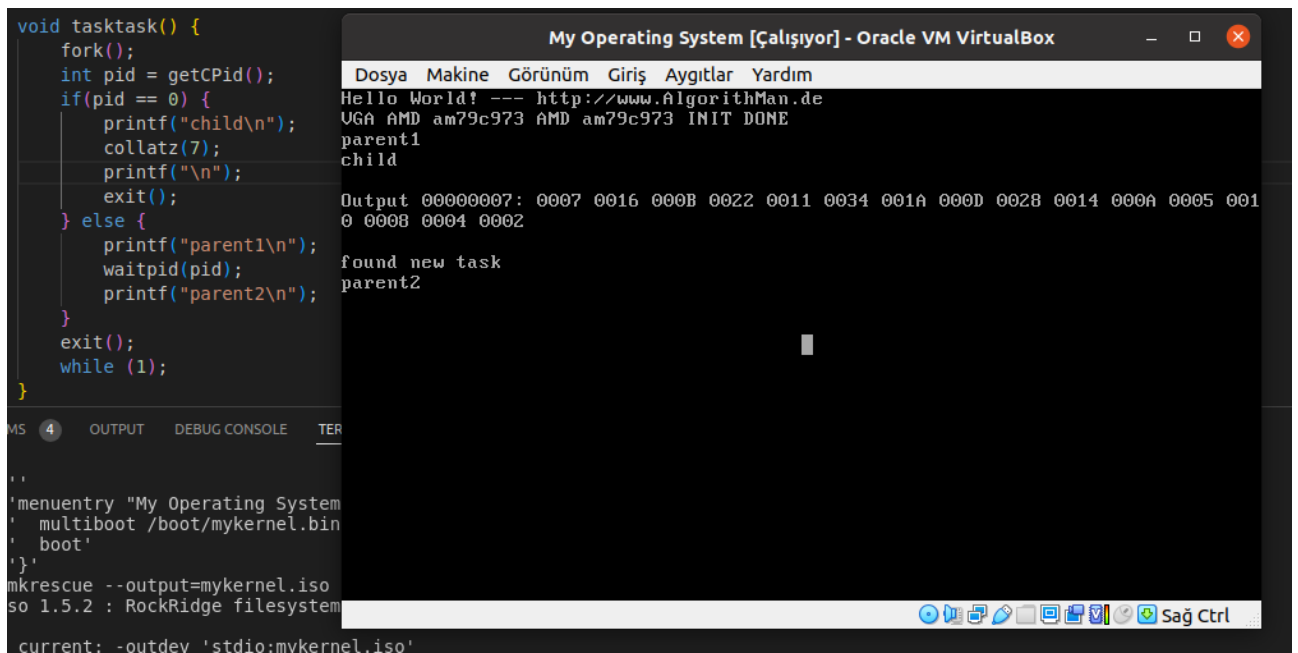
```
My Operating System [Çalışıyor] - Oracle VM VirtualBox
Dosya Makine Görünüm Giriş Aygıtlar Yardım
Hello World! --- http://www.AlgorithmMan.de
UGA AMD am79c973 AMD am79c973 INIT DONE
parent1
child
found new task
parent2
```

```
EMS 4 OUTPUT DEBUG CONSOLE TER
'menuentry "My Operating System
'  multiboot /boot/mykernel.bin
'  boot'
'}
```

```
-mkrescue --output=mykernel.iso
iso 1.5.2 : RockRidge filesystem

e current: -outdev 'stdio:mykernel.iso
a current: stdio file, overwriteable
```

Basic Demonstration of Running Collatz Function

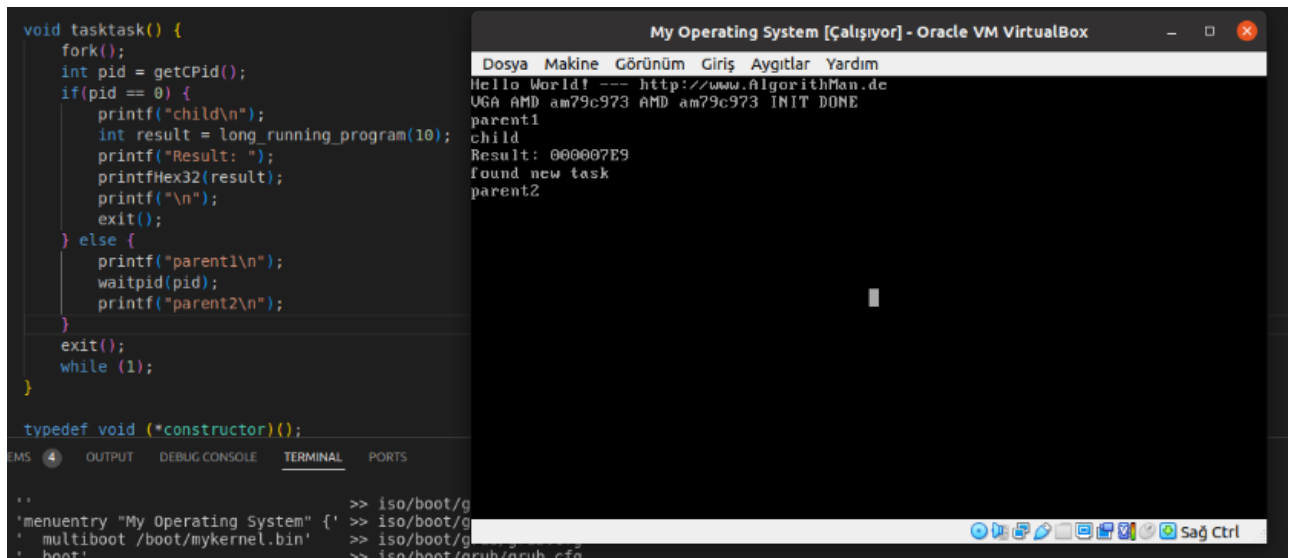


The screenshot shows a VM window titled "My Operating System [Çalışıyor] - Oracle VM VirtualBox". The code on the left defines a `tasktask()` function that forks a child process to calculate the 7th Collatz number. The terminal output on the right shows the program's execution, including the initialization of parent and child processes, the calculation of the 7th Collatz number (16), and the output of the sequence of numbers from 0 to 16.

```
void tasktask() {
    fork();
    int pid = getCPid();
    if(pid == 0) {
        printf("child\n");
        collatz(7);
        printf("\n");
        exit();
    } else {
        printf("parent1\n");
        waitpid(pid);
        printf("parent2\n");
    }
    exit();
    while (1);
}
```

```
My Operating System [Çalışıyor] - Oracle VM VirtualBox
Dosya Makine Görünüm Giriş Aygıtlar Yardım
Hello World! --- http://www.AlgorithMan.de
UGA AMD am79c973 AMD am79c973 INIT DONE
parent1
child
Output 00000007: 0007 0016 000B 0022 0011 0034 001A 000D 0028 0014 000A 0005 0010 000B 0004 0002
found new task
parent2
```

Basic Demonstration of Running Long Running Program Function



The screenshot shows a VM window titled "My Operating System [Çalışıyor] - Oracle VM VirtualBox". The code on the left defines a `tasktask()` function that forks a child process to run a long-running program. The terminal output on the right shows the program's execution, including the initialization of parent and child processes, the execution of the long-running program, and the output of the result (000007E9).

```
void tasktask() {
    fork();
    int pid = getCPid();
    if(pid == 0) {
        printf("child\n");
        int result = long_running_program(10);
        printf("Result: ");
        printfHex32(result);
        printf("\n");
        exit();
    } else {
        printf("parent1\n");
        waitpid(pid);
        printf("parent2\n");
    }
    exit();
    while (1);
}
```

```
My Operating System [Çalışıyor] - Oracle VM VirtualBox
Dosya Makine Görünüm Giriş Aygıtlar Yardım
Hello World! --- http://www.AlgorithMan.de
UGA AMD am79c973 AMD am79c973 INIT DONE
parent1
child
Result: 000007E9
found new task
parent2
```

Here is the program simulation video links:

https://drive.google.com/file/d/1SLkWrIdDY1r6WrueARw8tDww0I1z_88E/view?usp=s_haring

https://drive.google.com/file/d/1R2ks72M4E8HE3pnV2f8TnAOx8ZvWCDjo/view?usp=s_haring

Conclusion

The provided code implements a basic task management system for an operating system, showcasing how tasks are initialized, managed, and scheduled. The `Task` class is responsible for setting up the CPU state for each task, with constructors and destructors to handle task lifecycle management. The `TaskManager` class orchestrates the overall management of tasks, including adding new tasks, forking tasks, and handling state transitions between ready, running, waiting, and finished states. The `Schedule` function is pivotal, determining the next task to execute based on its state, ensuring smooth and efficient task transitions. Additionally, functions for exiting and waiting on tasks, along with methods to get task IDs and display the task table, provide comprehensive control over task management. This structured approach forms the backbone of multitasking in an operating system, allowing for dynamic and responsive task handling.