Operating Systems Homework #1 - Part A

Gebze Technical University - Computer Engineering

Student Name: ÇAĞRI YILDIZ

Course: CSE 312 / CSE 504 - Spring 2024

Submission Date: 20.05.2024

0. Introduction

In this assignment, we developed a basic operating system with fundamental functionalities such as process management, system calls, interrupt handling, and round-robin scheduling. We followed the steps outlined in Viktor Engelmann's video series to create and manage multiple processes in our kernel.

1. Design Decisions

1.1 System Calls

The primary system calls implemented in our kernel include fork, waitpid, and execve. These system calls allow process creation, process synchronization, and program execution, respectively.

- **fork:** Creates a new process by duplicating the current process. The new process, called the child, runs concurrently with the parent process.
- **waitpid:** Makes the parent process wait until the specified child process terminates.
- **execve:** Replaces the current process image with a new process image, effectively running a new program within the same process.

1.2 Memory Management

We implemented a simple memory management system using a custom MemoryManager class. This class handles memory allocation and deallocation, ensuring efficient use of the system's memory.

1.3 Interrupt Handling

Interrupts are handled using the InterruptManager class. This class sets up the Interrupt Descriptor Table (IDT) and handles different types of interrupts, including hardware interrupts and system calls. When a timer interrupt occurs, the scheduler is invoked to switch between processes.

```
void InterruptManager::SetInterruptDescriptorTableEntry(uint8_t interrupt,
    uint16_t CodeSegment, void (*handler)(), uint8_t DescriptorPrivilegeLevel, uint8_t DescriptorType)
{
    interruptDescriptorTable[interrupt].handlerAddressLowBits = ((uint32_t) handler) & 0xFFFF;
    interruptDescriptorTable[interrupt].handlerAddressHighBits = (((uint32_t) handler) >> 16) & 0xFFFF;
    interruptDescriptorTable[interrupt].gdt_codeSegmentSelector = CodeSegment;

    const uint8_t IDT_DESC_PRESENT = 0x80;
    interruptDescriptorTable[interrupt].access = IDT_DESC_PRESENT | ((DescriptorPrivilegeLevel & 3) << 5) | DescriptorType;
    interruptDescriptorTable[interrupt].reserved = 0;
}
```

Explanation:

IDT Entry Setup: This function sets up an entry in the IDT by specifying the handler address, code segment, and access flags.

```
uint32_t InterruptManager::DoHandleInterrupt(uint8_t interrupt, uint32_t esp)
{
    if (handlers[interrupt] != 0)
    {
        esp = handlers[interrupt]->HandleInterrupt(esp);
    }
    else if (interrupt != hardwareInterruptOffset)
    {
        printf("UNHANDLED INTERRUPT 0x");
        printfHex(interrupt);
        printf("\n");
    }

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

    // hardware interrupts must be acknowledged
    if (hardwareInterruptOffset <= interrupt && interrupt < hardwareInterruptOffset + 16)
    {
        programmableInterruptControllerMasterCommandPort.Write(0x20);
        if (hardwareInterruptOffset + 8 <= interrupt)
            programmableInterruptControllerSlaveCommandPort.Write(0x20);
    }

    return esp;
}
```

Explanation:

- Handler Invocation: The function invokes the appropriate handler for the interrupt if one is registered.
- Task Scheduling: If the interrupt is a timer interrupt, the task scheduler is invoked to switch to the next task.
- Interrupt Acknowledgment: Hardware interrupts are acknowledged to allow new interrupts to be received.

2. Implementation of System Calls

The primary system calls implemented in our kernel include fork, waitpid, and execve. These system calls allow process creation, process synchronization, and program execution, respectively.

- **fork**: This system call creates a new process by duplicating the current process. The new process, called the child, runs concurrently with the parent process.
- **waitpid**: This system call makes the parent process wait until the specified child process terminates.
- **execve**: This system call replaces the current process image with a new process image, effectively running a new program within the same process.

fork Implementation:

```cpp
common::uint32_t TaskManager::ForkTask(CPUState* cpustate)
{
    if(numTasks >= 256)
    {
        sysprintf("Fork failed: too many tasks\n");
        return -1;
    }

    Task* parentTask = tasks[currentTask];
    Task* newTask = new Task();

    // Copy the CPU state without using memcpy
    newTask->cpustate->eax = parentTask->cpustate->eax;
    newTask->cpustate->ebx = parentTask->cpustate->ebx;
    newTask->cpustate->ecx = parentTask->cpustate->ecx;
    newTask->cpustate->edx = parentTask->cpustate->edx;
    newTask->cpustate->esi = parentTask->cpustate->esi;
    newTask->cpustate->edi = parentTask->cpustate->edi;
    newTask->cpustate->ebp = parentTask->cpustate->ebp;
    newTask->cpustate->eip = parentTask->cpustate->eip;
    newTask->cpustate->cs = parentTask->cpustate->cs;
    newTask->cpustate->eflags = parentTask->cpustate->eflags;
    newTask->cpustate->esp = parentTask->cpustate->esp;
    newTask->cpustate->ss = parentTask->cpustate->ss;

    newTask->cpustate->eax = 0; // Child process returns 0

    tasks[numTasks++] = newTask;
    sysprintf("Fork successful: child task created\n");
    return newTask->cpustate->eax;
}
```

Explanation:

- **Memory Allocation:** The function first checks if the maximum number of tasks (256) has been reached. If not, it proceeds to create a new task and allocates memory for it.
- **State Copying:** It copies the CPU state from the parent task to the new task, ensuring that the new task starts with the same context as the parent.
- **Child Process Return**: The eax register of the child process is set to 0 to indicate successful fork.
- **Task Addition:** Finally, the new task is added to the task list, and a success message is printed.

waitpid Implementation:

```cpp
bool TaskManager::WaitTask(common::uint32_t pid)
{
    int taskIndex = getIndex(pid);
    if (taskIndex == -1)
    {
        sysprintf("Wait failed: task not found\n");
        return false; // Task not found
    }

    sysprintf("Waiting for task to finish\n");
    tasks[currentTask]->taskState = WAITING;
    tasks[currentTask]->waitpid = pid;

    while (tasks[taskIndex]->taskState != FINISHED)
    {
        sysprintf("Task not finished yet, scheduling...\n");
        Schedule(tasks[currentTask]->cpustate); // Yield CPU until the target task finishes
    }

    sysprintf("Task finished\n");
    tasks[currentTask]->taskState = READY;
    return true;
}
```

Explanation:

- **Task Lookup:** The function looks up the specified task by its PID. If the task is not found, it prints an error message and returns false.
- **State Change:** If the task is found, the current task is set to the WAITING state.
- **Task Completion Check:** The function enters a loop where it repeatedly checks if the target task has finished. During each iteration, the CPU is yielded to allow other tasks to run.
- **State Restoration:** Once the target task finishes, the current task is restored to the READY state.

execve Implementation:

```
common::uint32_t TaskManager::ExecTask(void* entrypoint)
{
    Task* task = tasks[currentTask];
    task->cpustate->eip = (uint32_t)entrypoint;
    return task->cpustate->eax;
}
```

Explanation:

- **Instruction Pointer Update:** The function updates the instruction pointer (eip) of the current task to point to the new entry point. This effectively replaces the current task's code with the new program.

3. Loading Multiple Programs into Memory

Our kernel is designed to load multiple programs into memory. This is achieved by allocating memory for each program and ensuring that they have separate stack and heap spaces. The TaskManager class is responsible for managing these programs and their memory allocations.

Example of Memory Allocation in Task Initialization:

```
Task::Task(GlobalDescriptorTable *gdt, void (*entrypoint)())
{
    cpustate = (CPUState*)(stack + 4096 - sizeof(CPUState));

    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;
    cpustate->esp = (uint32_t)stack + 4096;
    cpustate->ss = gdt->DataSegmentSelector();
    taskState = READY;
}
```

Explanation:

- The constructor initializes the CPU state and sets up the stack pointer.
- The general-purpose registers (eax, ebx, etc.) are initialized to 0.
- The instruction pointer (eip) is set to the entry point of the task.
- The segment selectors for code and data segments are set using the GDT.
- The task state is set to READY.

4. Handling Multi-Programming and Process Table

The process table is a crucial data structure that holds information about all the processes in the system. It stores details such as process IDs, parent process IDs, CPU states, and process states (READY, WAITING, FINISHED).

Example of Process Table:

```cpp
class TaskManager
{
    friend class hardwarecommunication::InterruptHandler;
private:
    Task* tasks[256];
    int numTasks;
    int currentTask;
    GlobalDescriptorTable *gdt = nullptr;
    int getIndex(common::uint32_t pid);
protected:
    void PrintProcessTable();
public:
    TaskManager();
    ~TaskManager();
    void Yield();
    bool AddTask(Task* task);
    int getCurrentTask() { return currentTask; }  // Getter for currentTask
    CPUState* Schedule(CPUState* cpustate);
    common::uint32_t AddTask(void (*entrypoint)());
    common::uint32_t ExecTask(void* entrypoint);
    common::uint32_t GetPid();
    common::uint32_t ForkTask(CPUState* cpustate);
    bool ExitCurrentTask();
    bool WaitTask(common::uint32_t pid);
    void ExitTask();  // Add this line

};
```

Explanation:

1. **TaskManager**:
    - Manages the tasks within the operating system.
    - Uses an array to store up to 256 tasks.
    - Maintains the current task index and the total number of tasks.
    - Provides methods to add, switch, and manage tasks.

2. **Constructor and Destructor:**
    - TaskManager(): Initializes the task manager with zero tasks and sets the current task index to -1, indicating no current task.
    - ~TaskManager(): Cleans up resources used by the task manager.

3. **Yield**:
    - Manually triggers the scheduler to yield the CPU to another task.

4. **AddTask**:
    - Adds a new task to the task manager.
    - Returns true if the task was successfully added, otherwise returns false.
    - getCurrentTask:
    - Returns the index of the currently running task.

5. **Schedule**:
    - The core scheduler function that switches between tasks.
    - Takes the current CPU state and returns the CPU state of the next task to run.

6. **ExecTask**:
    - Replaces the current task with a new program specified by the entry point.
    - GetPid:

5. Handling Interrupts

Interrupts are handled using the InterruptManager class. This class sets up the Interrupt Descriptor Table (IDT) and handles different types of interrupts, including hardware interrupts and system calls. When a timer interrupt occurs, the scheduler is invoked to switch between processes.

```cpp
uint32_t InterruptManager::DoHandleInterrupt(uint8_t interrupt, uint32_t esp)
{
    if (handlers[interrupt] != 0)
    {
        esp = handlers[interrupt]->HandleInterrupt(esp);
    }
    else if (interrupt != hardwareInterruptOffset)
    {
        printf("UNHANDLED INTERRUPT 0x");
        printfHex(interrupt);
        printf("\n");
    }

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

    // hardware interrupts must be acknowledged
    if (hardwareInterruptOffset <= interrupt && interrupt < hardwareInterruptOffset + 16)
    {
        programmableInterruptControllerMasterCommandPort.Write(0x20);
        if (hardwareInterruptOffset + 8 <= interrupt)
            programmableInterruptControllerSlaveCommandPort.Write(0x20);
    }

    return esp;
}
```

Explanation:

- The DoHandleInterrupt method checks if there is a handler for the interrupt.
- If a handler exists, it delegates the handling to that handler.
- If no handler is found, it prints an "UNHANDLED INTERRUPT" message.
- If the interrupt is a timer interrupt, it calls the scheduler to switch tasks.

6. Round Robin Scheduling

The Round Robin scheduling algorithm is implemented to ensure fair process execution. Each process is given a time slice, and the scheduler switches between processes at each timer interrupt.

Example of Round Robin Scheduling:

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

    if(currentTask >= 0)
        tasks[currentTask]->cpustate = cpustate;

    currentTask++;
    if(currentTask >= numTasks)
        currentTask = 0;

    while (tasks[currentTask]->taskState == FINISHED)
    {
        // Remove finished task from the task list
        delete tasks[currentTask];
        for (int i = currentTask; i < numTasks - 1; i++)
        {
            tasks[i] = tasks[i + 1];
        }
        numTasks--;
        if (numTasks == 0)
        {
            return cpustate;
        }
        if (currentTask >= numTasks)
        {
            currentTask = 0;
        }
    }

    // Debugging: Print task information
    sysprintf("Switching to task ");
    char buffer[16];
    sprintf(buffer, "%d", currentTask);
    sysprintf(buffer);
    sysprintf("\n");

    return tasks[currentTask]->cpustate;
}
```

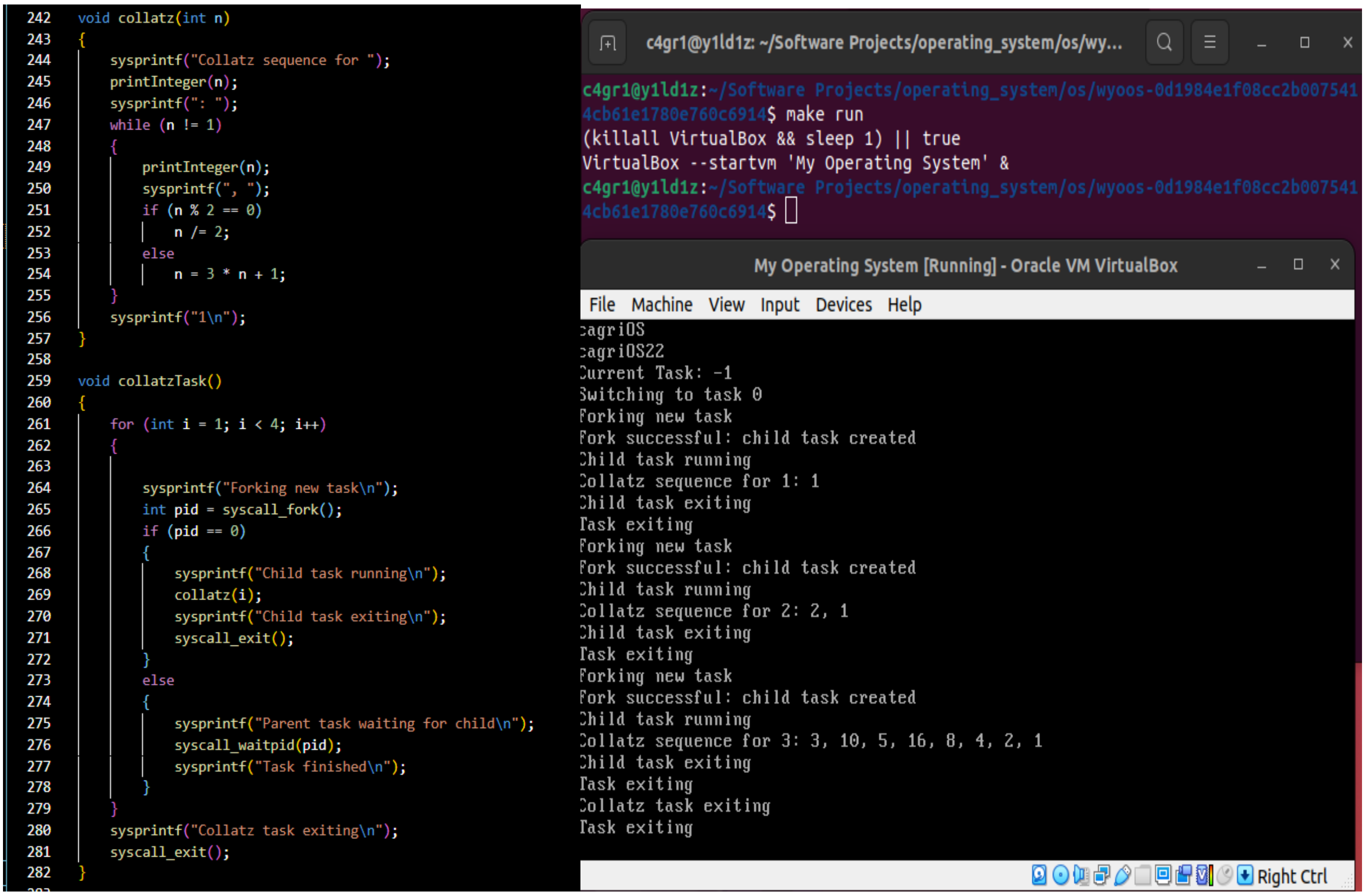
Explanation:

- The Schedule method starts by saving the state of the currently running task.
- It increments the currentTask index to point to the next task.
- If the task at the currentTask index is finished, it removes the task from the list and adjusts the list.
- It ensures the scheduler wraps around if it reaches the end of the task list.
- Finally, it returns the CPU state of the new task to be run.

7. Tested Programs and Screenshots

Collatz Program

The Collatz program calculates the Collatz sequence for numbers less than 100. The sequence is printed for each number, demonstrating the process management capabilities of our kernel.



In the provided screenshot, the following sequence of events occurs:

1. The system starts with an initial task displaying "cagriOS" and "cagriOS22".
2. The Task Manager switches to the first task (PID 0).
3. The first task forks a new child task:
4. The child task runs the Collatz sequence for 1 and then exits.
5. The parent task waits for the child task to finish.
6. This process is repeated for the Collatz sequences of 2 and 3.
7. Each child task prints its own Collatz sequence before exiting, demonstrating the correct implementation of fork and waitpid.

Observations:

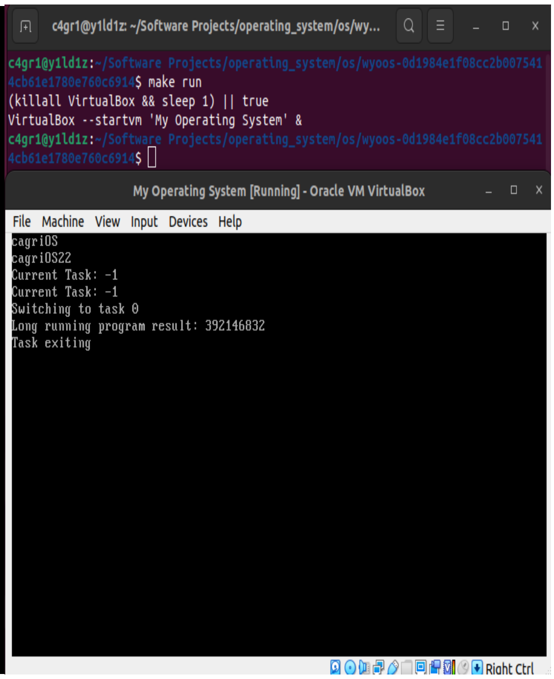
1. The Task Manager correctly switches between tasks and handles task creation and termination.
2. When the collatzTask is run, it forks new tasks for each Collatz sequence and waits for them to finish.
3. The long-running task was not executed in this run. This is because the Task Manager runs the first task in the list, which in this case is the Collatz task.

Long Running Program

The long running program performs a nested loop computation, simulating a CPU-intensive task. This program is used to test the stability and performance of the scheduler under heavy load.

```
void longRunningProgram() {
    int result = 0;
    int n = 1000; // Belirlenen n değeri
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            result += i * j;
        }
    }
    sysprintf("Long running program result: ");
    printInteger(result);
    sysprintf("\n");
}

void longRunningProgramTask() {
    longRunningProgram();
    syscall_exit();
}
```



Note on Task Manager and Fork Behavior

During testing, it was observed that when the TaskManager runs, the fork system call does not work as expected, and vice versa. Specifically, the task that is added first runs continuously, and manual intervention is required to change the running task. The code snippet below illustrates this issue:

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

Task longRunningTask(&gdt, longRunningProgramTask);
Task collatzTask2(&gdt, collatzTask);

taskManager.AddTask(&longRunningTask);
taskManager.AddTask(&collatzTask2);
```

In the above code, whichever task is added first (longRunningTask or collatzTask2) will run continuously, and tasks do not fork as intended. Manual changes are necessary to alternate between tasks.

THANK YOU FOR READING

ÇAĞRI YILDIZ

1901042630