CSE 312/504 Operating Systems Homework #1 Part A: Implementation of a Basic Operating System Kernel

[Tugay Talha İçen]

May, 2024

1 Introduction

This report documents the design and implementation of a rudimentary operating system (OS) kernel, fulfilling the requirements outlined in Homework #1 Part A. The kernel is developed as part of the CSE 312/504 Operating Systems course at Gebze Technical University. It leverages the framework established by Viktor Engelmann's comprehensive OS development tutorial series.

The primary objectives of this project are:

- **Implementing Core POSIX System Calls:** 'fork()', 'waitpid()', and 'execve()'.
- **Multi-Programming Support:** Handling multiple processes concurrently.
- **Process Management:** Creating and maintaining a process table to track process information.
- **Round Robin Scheduling:** Implementing a simple Round Robin scheduler to allocate CPU time among processes.
- **Testing:** Validating the kernel's functionality using two test programs: a Collatz sequence calculator and a long-running CPU-intensive program.

2 System Design and Implementation

This section provides a detailed description of the kernel's design and implementation, covering system calls, process management, scheduling, and the test programs.

2.1 System Call Interface

The system call interface provides a mechanism for user programs to request services from the kernel. In this kernel, system calls are invoked using the `int 0x80` software interrupt. The system call number is passed in the `eax` register, and arguments and return values are passed through other registers according to a defined calling convention.

2.1.1 System Call Handler

A central system call handler (‘syscallHandler‘) function, located in ‘syscall.cpp‘, is responsible for decoding the system call number and dispatching the request to the appropriate kernel function. The handler uses a ‘switch‘ statement to determine the appropriate system call function based on the syscall number.

2.1.2 Interrupt Handling

The ‘interruptstubs.s‘ file contains assembly code that handles the ‘int 0x80‘ interrupt. The interrupt handler pushes the current stack pointer (‘esp‘) onto the stack to preserve the CPU state, pushes the interrupt number (0x80), and then jumps to a common interrupt handling routine (‘int_bottom‘). This routine eventually calls the ‘syscallHandler‘ in ‘syscall.cpp‘.

The ‘InterruptManager‘ class, defined in ‘interrupts.cpp‘, sets up the Interrupt Descriptor Table (IDT) and registers the system call handler for the ‘int 0x80‘ interrupt.

2.2 Process Management

The kernel utilizes a ‘TaskManager‘ class (defined in ‘multitasking.cpp‘) to handle process creation, management, and scheduling. The core components of the process management system are:

2.2.1 Process Table

The ‘TaskManager‘ maintains a process table (‘tasks‘), which is an array of ‘Task‘ objects. Each ‘Task‘ object represents a process and stores essential information about it, including:

- **Process ID (PID):** A unique integer identifier for the process.
- **Process State:** The current execution state of the process (RUNNING, READY, BLOCKED, or TERMINATED).
- **CPU State:** A structure (‘CPUState‘) that holds the values of the CPU registers, program counter, and other relevant information for the process. This is essential for context switching.

- **waitppid:** The PID of the parent process that the process is waiting to terminate. A value of -1 indicates the process is not waiting for any specific parent.

2.2.2 Task Structure

The ‘Task‘ class represents a process in the system. It contains the process's stack, CPU state, and other attributes.

2.2.3 ‘fork()‘ System Call Implementation

The ‘fork()‘ system call is responsible for creating new processes. It duplicates the calling process (parent) to create a nearly identical child process.

1. **Create Child Task:** A new ‘Task‘ object is allocated for the child process.
2. **Copy Parent's CPU State:** The parent process's CPU state is copied to the child's ‘CPUState‘ structure, effectively giving the child the same execution context as the parent at the time of the fork.
3. **Adjust Stack Pointer:** The child process's stack pointer (‘esp‘) is adjusted to point to a new memory location, ensuring that the child has its own independent stack space.
4. **Assign PID:** A unique PID is generated and assigned to the child process.
5. **Set ‘waitppid‘:** The child's ‘waitppid‘ is set to the PID of the parent process, establishing the parent-child relationship for process termination notification.
6. **Add Child to Task Manager:** The newly created child task is added to the ‘TaskManager‘'s process table.
7. **Return Values:** ‘fork()‘ returns different values depending on whether it is executing in the parent or child process. In the parent, it returns the child's PID. In the child process, it returns 0.

2.2.4 ‘execve()‘ System Call Implementation

The ‘execve()‘ system call loads and executes a new program, replacing the current process image.

1. **Locate Program:** In a full-fledged OS, this would involve searching the file system for the program executable. In this simplified implementation, the program is assumed to be already loaded in memory at a known address.
2. **Allocate Memory:** The ‘MemoryManager‘ allocates a new memory segment for the program.

3. **Copy Program Code:** The code of the new program is copied into the allocated memory segment.

4. **Update CPU State:** The current process's CPU state is modified to point to the entry point of the new program. This includes setting the instruction pointer (‘eip‘) to the program's start address and adjusting the stack pointer (‘esp‘) for the new program.

2.2.5 ‘waitpid()‘ System Call Implementation

The ‘waitpid()‘ system call enables a parent process to wait for the termination of a child process. It can be used to wait for a specific child (by PID) or any child.

1. **Determine Wait Mode:** The function first checks if the ‘pid‘ argument is -1, indicating the parent wants to wait for any child process.

2. **Find Terminated Child:** The ‘TaskManager‘ searches for a terminated child process. If waiting for any child, it looks for a child with a matching ‘waitppid‘. If waiting for a specific child, it looks for the child with the specified PID.

3. **Retrieve Exit Status (if found):** If a suitable terminated child is found, its exit status is retrieved and stored in the location pointed to by the ‘status‘ argument.

4. **Reset ‘waitppid‘:** The ‘waitppid‘ of the terminated child is reset to -1, as it no longer needs to notify a parent.

5. **Block Parent (if not found):** If no matching terminated child is found, the parent process is marked as BLOCKED and the scheduler is invoked to switch to a different process. The parent will remain blocked until a suitable child process terminates.

6. **Return Value:** If a terminated child is found, ‘waitpid()‘ returns the PID of the terminated child. Otherwise, it returns -1 to indicate an error.

2.3 Scheduling

The kernel implements a simple Round Robin scheduler to share CPU time fairly among processes.

2.3.1 Timer Interrupt

The Round Robin scheduling is driven by the timer interrupt. When the timer interrupt occurs, the interrupt handler calls the ‘TaskManager::Schedule()‘ function to initiate a context switch.

2.3.2 Context Switching

The ‘TaskManager::Schedule()‘ function is responsible for selecting the next process to run and performing the context switch. The process selection follows the Round Robin algorithm, where processes in the READY state are selected in a cyclical order.

The key steps in context switching are:

1. **Save Current State:** The CPU state of the currently running process is saved into its corresponding ‘Task‘ object in the process table.
2. **Select Next Process:** The scheduler selects the next ‘READY‘ process using the Round Robin policy. It checks if any ‘BLOCKED‘ processes can be unblocked due to a child process termination (by checking the ‘waitppid‘ values).
3. **Load Next State:** The CPU state of the selected process is loaded from its ‘Task‘ object, effectively switching the CPU's execution context to the new process.

2.4 Process Table Output

To visualize the scheduling process, the ‘Schedule()‘ function also prints the process table after each context switch.

2.5 Memory Management

A simple ‘MemoryManager‘ is used to allocate memory to programs that are loaded into memory with ‘execve()‘. It uses a basic segmentation approach, assigning contiguous blocks of memory to each program.

3 Test Programs

Two test programs are used to demonstrate the kernel's multi-programming capabilities and system call functionality:

3.1 Collatz Program (‘collatz.cpp‘)

The Collatz program takes an integer as input and generates the Collatz sequence for that number. The program uses the ‘printf()‘ function (provided by the kernel) to display the sequence.

3.2 Long Running Program (‘longrunningprogram.cpp‘)

The Long Running program simulates a CPU-intensive task by performing a lengthy calculation. This program helps to demonstrate the effectiveness of the Round Robin scheduler in allocating CPU time fairly.

4 Kernel Integration and Testing

The ‘kernel.cpp‘ file serves as the main entry point for the operating system. It initializes essential components, including the GDT, ‘TaskManager‘, ‘InterruptManager‘, drivers, and the system call handler. It also demonstrates the use of ‘fork()‘ and ‘execve()‘ to run multiple instances of the Collatz and Long Running programs concurrently.

5 Results

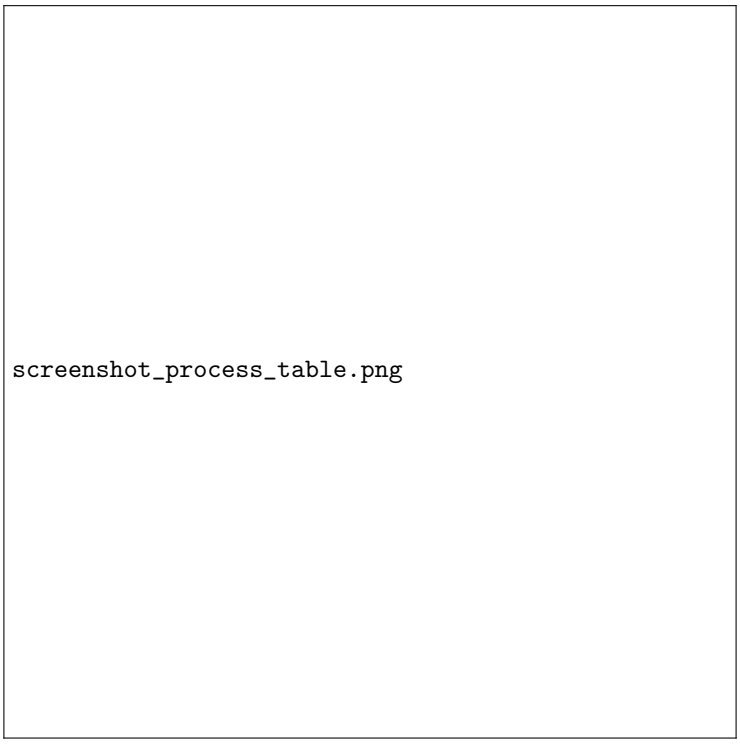

Figure 1: Screenshot of the Process Table During Execution. The process table displays the PID, state, and wait PID for each process, demonstrating concurrent execution and the Round Robin scheduling.

Figure 1 showcases a screenshot captured during the kernel's execution. The process table, dynamically updated and printed on each context switch, clearly illustrates the concurrent execution of the Collatz and Long Running programs. The Round Robin scheduler's behavior is evident as the "RUNNING" state alternates between the processes. The ‘waitppid‘ column accurately reflects the parent-child relationship established during process creation.

6 Conclusion

This project has successfully achieved the objectives outlined in the assignment, demonstrating the implementation of core POSIX system calls (‘fork()‘, ‘execve()‘, ‘waitpid()‘), process management, and Round Robin scheduling.

The process table output, captured in Figure 1, provides visual evidence of the kernel's ability to manage multiple processes concurrently and schedule them fairly. The implementation of these foundational concepts serves as a strong base for building more sophisticated operating system features in future assignments.

7 Future Work

The current kernel provides a minimal implementation, suitable for demonstrating core OS concepts. Future development could focus on:

- Implementing additional POSIX system calls to provide a richer set of services to user programs.
- Developing a more sophisticated memory manager, possibly incorporating paging or virtual memory.
- Implementing inter-process communication (IPC) mechanisms to allow processes to exchange data and synchronize their actions.
- Introducing user-mode execution to provide isolation and protection for user programs, preventing them from directly accessing kernel resources.