1 Introduction

This report provides an in-depth discussion of the system call mechanism as implemented in the Simple Operating System assignment. It explains the design and implementation details, answers key questions from the assignment, and describes the inter-module interactions related to system calls. The system call interface is the primary mechanism for communication between user applications and the OS kernel.

2 Overview of System Calls in the Simple OS

In the Simple OS, system calls are not invoked directly by application code. Instead, thin wrapper functions in libraries (e.g., `libstd`) copy arguments into designated registers before transferring control to the kernel. The kernel then dispatches the appropriate system call handler by using a system call table where each unique syscall number is mapped to its corresponding function.

- **Wrapper Functions:** These functions move small integers directly into registers or pass pointers for more complex data.
- **System Call Table:** The kernel uses a table that maps each system call number (e.g., 440 for a custom system call) to the corresponding handler function.

Details regarding the system call mechanism and the role of wrapper functions are explained in Section 2.3 of the assignment document.

3 Passing Complex Arguments via System Calls

3.1 Challenges of Limited Registers

A key challenge in implementing system calls is the limited number of registers available for passing arguments. For complex data structures, simply using registers is insufficient.

3.2 The Pointer Mechanism

To overcome this limitation, the system call mechanism uses an indirect method:

1. The user-level wrapper function stores complex data in a memory block.
2. It then passes a pointer (address) to this memory block via a register.
3. In the kernel, the pointer is dereferenced to access the complete argument.

This design minimizes the need for multiple registers and leverages the operating system's memory management capabilities. Additionally, the kernel validates the pointer before accessing the memory to prevent security violations.

Question: What is the mechanism to pass a complex argument to a system call using the limited registers ?

Answer: When dealing with complex arguments that can't fit into the limited number of registers, the common approach is to pass a pointer to the data, this can be achieved by storing the complex data in user-space memory and passing a pointer to this data to the kernel. While passing pointers is effective, it has limitations, such as potential performance overhead for large copies and security risks if not handled properly.

4 Handling Long-Running System Calls

4.1 Potential Issues

A system call that takes too long to execute may cause several problems:

- **CPU Monopolization:** Extended execution in kernel mode can block other processes.
- **Reduced System Responsiveness:** Other system calls and interrupt handling may be delayed.
- **Deadlocks and Resource Starvation:** Non-preemptive long-running calls can lead to resource contention.

4.2 Mitigation Strategies

To mitigate these issues, operating systems typically employ:

- **Preemption:** Implementing time-slicing or watchdog timers to interrupt long-running calls.
- **Asynchronous Processing:** Offloading heavy operations to asynchronous routines.
- **Timeout Mechanisms:** Aborting a system call if it exceeds a predefined time limit.

Question: What happens if the syscall job implementation takes too long execution time ?
Answer: If the system call takes too long to execute, it can block other processes and hold system resources, potentially causing resource starvation or deadlocks. Proper preemption or asynchronous handling must be implemented to avoid these issues.

5 Inter-Module Interactions in System Call Processing

5.1 Process Control and Scheduling Queue Management

- **Process Control Blocks (PCBs):** Each process is represented by a PCB that stores its identifier, registers, and pointers to memory areas. For example, system calls like `killall` use the PCB information to terminate processes whose names match a given pattern.
- **Scheduling Queues:** When a system call such as `killall` is executed, the kernel searches through the scheduling queues to find matching processes and removes their PCBs from the queues.

5.2 Memory Management Interaction

- **Memory Mapping (`memmap`):** System calls for memory mapping manage the virtual memory space of a process, handling tasks like address translation and page swapping.
- **Resource Coordination:** The system call interface works closely with the memory management module to ensure that operations such as allocation (`ALLOC` and `FREE`) and paging are executed without disrupting process isolation or scheduling.

The coordination among these modules ensures that system calls like `killall` correctly terminate processes and that memory operations via `memmap` do not interfere with process scheduling or memory isolation.

6 Conclusion

The system call mechanism in the Simple Operating System provides a robust interface between user applications and kernel services. By using wrapper functions and pointer-based argument passing, the OS overcomes the register limitation for passing complex arguments. However, if a system call takes too long, it can hinder overall system performance, potentially causing delays or deadlocks. Proper mitigation strategies such as preemption, asynchronous processing, and timeout mechanisms are essential to maintain system responsiveness.

Understanding the inter-module interactions between process control, scheduling, and memory management is critical for building a robust and reliable operating system. This report has addressed the key questions:

1. Complex arguments are passed indirectly via pointers.
2. Long-running system calls can monopolize CPU resources, leading to potential resource starvation or deadlocks.