SYSC 4001 – Assignment 2 Report

Shael Kotecha – 101301744
Declan Koster – 101310649

Part II – Concurrent Processes in Unix

This assignment explored the creation and management of concurrent processes in a Unix/Linux environment using C++. The objective was to develop multiple processes that communicate and synchronize via system calls such as fork(), exec(), wait(), shared memory, and semaphores. Each part built upon the previous one, reinforcing core OS concepts in process creation, communication, and synchronization.

In **Part 1**, two independent processes were created using fork(). The parent and child each maintained their own counter in infinite loops, incrementing independently. To keep the output readable, was used to introducentroduced a short delay was introduced. The processes were terminated manually using ps and kill, demonstrating basic process creation and independent execution.

Part 2 introduced the exec() call. After forking, the child replaced itself with a new executable via execl(). The parent incremented its counter while the child decremented its own, each printing when their counters were multiples of three. This showed concurrent execution of separate programs and the independence of their memory spaces.

Part 3 added synchronization using wait(). The parent created a child via exec() and then paused until the child completed its task—decrementing a counter until −500. Once the child terminated, the parent resumed and exited. This illustrated how parent processes can synchronize with children, ensuring orderly termination.

Part 4 introduced **shared memory** for inter-process communication. Using shmget() and shmat(), both processes shared two variables: multiple and counter. The child began only after the counter exceeded 100 and terminated once it surpassed 500. Shared memory was detached and removed via shmdt() and shmctl(), highlighting efficient, file-free data sharing.

Finally, **Part 5** integrated **semaphores** to protect concurrent access to shared variables. Calls to semget(), semop(), and semctl() enforced mutual exclusion, preventing race conditions and maintaining data consistency. This completed the demonstration of safe, synchronized inter-process communication.

Part III – Design and Implementation of an API Simulator

Part III focused on designing an API simulator that models the behavior of fork() and exec() system calls. The simulator emulates OS-level process management using fixed memory partitions, Process Control Blocks (PCBs), and simulated interrupt routines. Managing shared

tables, loading external programs, and logging system calls demonstrates how operating systems clone processes, load executables, and handle scheduling events.

Test Case 1 – Basic FORK and EXEC

The trace begins with a FORK, creating a child process. The kernel saves the parent context, clones the PCB, and places both processes in the ready queue. The child then executes EXEC program1, replacing its image with the new program and loading it into memory based on its size. After completing a 100 ms CPU burst, control returns to the parent, which performs EXEC program2, containing a single system call. The system status confirms two PCBs exist—the child running and the parent waiting—validating correct process creation and execution replacement.

Test Case 2 – Nested FORKs and Multiple EXECs

This test verifies nested process creation. The initial FORK creates PID 1, which performs EXEC program1. That program forks again, creating PID 2, which executes program2. After PID 2 finishes, PID 1 also executes program2. The simulator dynamically allocates memory, updates PCBs, and manages process termination. The system status reflects correct creation and cleanup of multiple PCBs, confirming accurate nested process and memory handling.

Test Case 3 – FORK, EXEC, and I/O Interrupt Handling

This test demonstrates interrupt management. After a FORK, the child performs a 10 ms CPU burst and terminates. The parent then executes EXEC program1, which runs a 50 ms CPU burst before triggering a SYSCALL. The simulator switches to kernel mode, saves the context, and locates the appropriate ISR via the interrupt vector. After handling the call, execution resumes with another 15 ms CPU burst, followed by an END_IO interrupt signaling completion. Logs confirm accurate context switching and ISR handling, modeling realistic OS interrupt behavior.

Conclusion

This assignment demonstrated essential Linux concepts in concurrent programming, including process creation, execution, synchronization, and inter-process communication. Through implementing fork(), exec(), wait(), shared memory, and semaphores, we gained practical insight into process coordination and resource sharing. The API simulator further reinforced understanding of process management and system-level operations, forming a solid foundation for future work in operating system design.