Sistemas Operativos **Ano lectivo 2025/2026**

Trabalho Prático 2

IPC with Semaphores Project[1]

3-Week Work Plan

Course Information

Target Audience: Undergraduate students in Operating Systems course

Project Duration: 3 weeks (approximately 21 days)

Language: C programming language

Prerequisites: Basic knowledge of C programming, processes, and threads

Project Overview

This project focuses on implementing Inter-Process Communication (IPC) mechanisms using semaphores in C. Students will design and implement a multi-process application that demonstrates synchronization, mutual exclusion, and coordination between processes using POSIX semaphores. The project emphasizes practical understanding of concurrent programming challenges including race conditions, deadlocks, and resource sharing.

Project Objectives

1. Implement IPC using POSIX semaphores (named and unnamed)
2. Solve classic synchronization problems (Producer-Consumer, Reader-Writer, or Dining Philosophers)
3. Demonstrate proper resource management and prevent race conditions
4. Implement error handling and debugging mechanisms
5. Document design decisions and provide comprehensive testing

Learning Outcomes

Upon completion of this project, students will be able to:

1. Understand and implement semaphore-based synchronization mechanisms
2. Identify and prevent common concurrency issues (race conditions, deadlocks)
3. Design multi-process applications with proper synchronization
4. Debug concurrent programs using appropriate tools and techniques

[1] The text of this project proposal had AI contributions to its completion.

5. Analyze and evaluate the performance of concurrent systems

Technical Requirements

1. **Programming Language:** C (C99 standard or later)
2. **Semaphore Library:** POSIX semaphores (semaphore.h)
3. **Operating System:** Linux/Unix-based system
4. **Compiler:** GCC with pthread library (-lpthread flag)
5. **Minimum Processes:** 3-5 concurrent processes
6. **Code Style:** Follow consistent coding standards with proper comments

3-Week Schedule Breakdown

Week 1: Design and Foundation (Days 1-7)

Goals

- Understand semaphore concepts and POSIX API
- Select a synchronization problem to implement
- Design system architecture and data structures

Day	Tasks
Day 1-2	**Research & Problem Selection** • Review semaphore theory (wait/signal operations) • Study POSIX semaphore API (sem_open, sem_wait, sem_post, sem_close) • Select synchronization problem (Producer-Consumer recommended for beginners) • Review example code and tutorials
Day 3-4	**System Design** • Design process architecture (number of processes, roles) • Define shared resources (buffers, counters) • Identify required semaphores (mutex, full, empty, etc.) • Create flowcharts/diagrams for process interactions
Day 5-7	**Basic Implementation** • Set up development environment and project structure • Implement basic process creation (fork/exec) • Create semaphore initialization and cleanup functions • Implement simple test case with 2 processes

Week 1 Deliverable: Design document (2-3 pages) with diagrams and basic skeleton code

Week 2: Core Implementation (Days 8-14)

Goals

- Implement complete synchronization logic
- Handle edge cases and error conditions
- Begin testing and debugging

Day	Tasks
Day 8-10	**Synchronization Logic** • Implement critical sections with proper semaphore operations • Add producer logic (data generation and buffer insertion) • Add consumer logic (data retrieval and processing) • Implement shared memory or message queue for data exchange
Day 11-12	**Error Handling & Robustness** • Add error checking for all system calls • Implement graceful shutdown and resource cleanup • Handle signal interrupts (SIGINT, SIGTERM) • Add logging for debugging purposes
Day 13-14	**Testing & Debugging** • Test with different numbers of processes • Verify no race conditions occur (use stress testing) • Check for deadlock scenarios • Use debugging tools (gdb, valgrind)

Week 2 Deliverable: Working prototype with core functionality and test results

Week 3: Refinement and Documentation (Days 15-21)

Goals

- Optimize performance and fix remaining bugs
- Complete comprehensive documentation
- Prepare final presentation/demo

Day	Tasks
Day 15-16	**Code Refinement** • Code review and refactoring • Optimize semaphore usage and reduce overhead • Add comments and improve code readability • Ensure compliance with coding standards
Day 17-18	**Documentation** • Write user manual with compilation and execution instructions • Create technical report explaining design and implementation • Document test cases and results • Include performance analysis and observations
Day 19-21	**Final Testing & Presentation** • Conduct comprehensive final testing • Prepare demonstration script • Create presentation slides (if required) • Package all deliverables for submission

Week 3 Deliverable: Complete project submission with all documentation

Final Deliverables

1. **Source Code:** Complete, well-commented C code with Makefile
2. **Design Document:** Architecture diagrams, flowcharts, and design rationale (2-3 pages)
3. **Technical Report:** Implementation details, challenges faced, and solutions (4-6 pages)
4. **User Manual:** Instructions for compilation, execution, and configuration
5. **Test Results:** Test cases, output logs, and performance measurements
6. **Demo Video/Presentation:** 5-10 minute demonstration (optional but recommended)

Recommended Resources

Essential Reading

- Modern Operating Systems (Tanenbaum) - Chapter 2
- Operating System Concepts (Silberschatz, Galvin, Gagne) - Chapter 6
- The Linux Programming Interface (Kerrisk) - Chapters 53-54

Online Resources

- POSIX Semaphore Manual: man 7 sem_overview
- Linux IPC Tutorial: https://man7.org/linux/man-pages/
- GeeksforGeeks: Process Synchronization tutorials

Development Tools

- **GDB:** GNU Debugger for debugging concurrent programs
- **Valgrind:** Memory leak detection and profiling
- **Helgrind:** Thread error detector (part of Valgrind)
- **strace:** System call tracer

Tips for Success

- **Start early -** Synchronization bugs can be difficult to debug
- **Test frequently -** Run your code multiple times to catch race conditions
- **Use version control -** Git helps track changes and revert mistakes
- **Read man pages -** POSIX documentation is your best friend
- **Collaborate responsibly -** Discuss concepts but write your own code
- **Document as you go -** Don't leave all documentation for the last day

Common Pitfalls to Avoid

- **Forgetting to initialize semaphores** - Always check return values
- **Not cleaning up resources** - Use sem_close() and sem_unlink()
- **Busy waiting** - Use blocking semaphore operations, not loops
- **Incorrect semaphore ordering** - Can lead to deadlock
- **Not handling signals** - Implement SIGINT handler for clean shutdown
- **Assuming atomic operations** - Always use synchronization primitives

Conclusion

This 3-week work plan provides a structured approach to mastering Inter-Process Communication using semaphores. By following this schedule, students will gain hands-on experience with one of the most fundamental concepts in operating systems - process synchronization. The project emphasizes not just coding skills, but also system design, debugging, and documentation abilities that are essential for any systems programmer.

Good luck with your implementation!

For questions or clarifications, please contact Prof. Pedro Azevedo Fernandes (paf@ua.pt) or Prof. Nuno Lau (nunolau@ua.pt).