**Project 3**

**Intro to Operating Systems**

**Masiko Mamba**

Table of Contents

[Abstract 3](#_Toc183543326)

[Introduction 3](#_Toc183543327)

[Requirements 3](#_Toc183543328)

[Project goal 3](#_Toc183543329)

[Constraints 3](#_Toc183543330)

[Resources 3](#_Toc183543331)

[Environment 3](#_Toc183543332)

[Methodology 3](#_Toc183543333)

[Design 3](#_Toc183543334)

[Synchronization 3](#_Toc183543335)

[Semaphores: 3](#_Toc183543336)

[Implementation: 4](#_Toc183543337)

[Results and Discussion 4](#_Toc183543338)

[Performance 4](#_Toc183543339)

[Synchronization 4](#_Toc183543340)

[Challenges 4](#_Toc183543341)

[Sample output 4](#_Toc183543342)

[Analysis 5](#_Toc183543343)

[Conclusion 5](#_Toc183543344)

# Abstract

This project is exploring the use of producers and consumers in process synchronization using semaphores. We use a circular buffer with a size of 15 to facilitate communication between the producer and consumer threads. The producer is used to read characters from a file and inserts them into the buffer. The consumer gets and prints the characters from the buffer. The semaphores ensure mutual exclusion and synchronization, preventing data races and buffer overflow.

# Introduction

The producer and consumer is a synchronization problem used to study communication between concurrent processes or threads. In this project, a producer thread is used to read characters from a file, “mytest.dat” and writes them to a circular. The circular thread retrieves the characters and prints them. Synchronization is achieved using three semaphores: one to track empty slots, one to track filled slots, and a mutex for access to the critical section. T he project demonstrates the use of semaphores to manage shared resources efficiently and prevent race conditions.

# Requirements

## Project goal

* Implement process synchronization using semaphores for the producer and consumer
* Show communication between producer and consumer threads via a circular buffer.

## Constraints

* Buffer has a size of 15
* Producer can read up to 150 characters

## Resources

* Circular buffer
* Semaphores
* Programming language: C with POSIX threads

## Environment

* Linux system
* GCC compiler

# Methodology

## Design

* Use a shared circular buffer with a size of 15 for communication between the producer and consumer threads.
* The producer will read the characters from “mytest.dat” and insert them into a buffer
* The consumer gets the characters from the buffer and prints them.

## Synchronization

### Semaphores:

* Empty: track available slots in buffer
* Full: track filled slots in buffer
* Mutex: ensure mutual exclusion to control access

## Implementation:

* The producer and consumer are created as separate threads in the main process
* The producer writes a “**\***” to signal end of file.
* The main process waits for both threads to complete and then destroys all semaphores.

# Results and Discussion

## Performance

* The program reconstructs the content of the file successfully
* The circular buffer handles communication ensuring that the data is not lost or overwritten.

## Synchronization

* Semaphores prevent race conditions and maintain proper synchronization.

## Challenges

* Ensuring initialization and that semaphores are properly destroyed

### Sample output

The output your program produces will be a reconstruction of the original thread contained in “mytest.dat”.

A screen shot of a computer

Description automatically generated

### Analysis

* The buffer is utilized effectively and allows the program to prevent race conditions effectively
* Semaphore utilization is good, access is restricted, and the program runs as expected without race conditions.

# Conclusion

This project demonstrates the effective use of semaphores. The implemented solution ensures synchronization, mutual exclusion, and efficient communication between threads. The results validate the effectiveness of semaphores in preventing race conditions and resource conflicts. This project provides insights into process synchronization techniques and allows us to solve more complex concurrency problems.