Report

Author: Roberto Cervantes
Username: cs017

Abstract

This report discusses the implementation of semaphores to synchronize producer-consumer operations using a shared memory circular buffer. The producer thread reads characters from a file and places them into the buffer, while the consumer retrieves and prints the data. Semaphores were used to ensure proper synchronization, prevent race conditions, and maintain data integrity. The implementation demonstrates effective thread communication and synchronization through semaphores and critical section protection.

1. Introduction

Managing concurrent access to shared resources is a critical aspect of multi-threaded programming. This project implements a producer-consumer model using a circular buffer of size 15, shared between two threads.

The **producer** reads characters from a file (`mytest.dat`) and writes them to the buffer, while the **consumer** retrieves the characters and prints them. Synchronization was achieved using three semaphores:

- **sem_items:** Tracks items in the buffer, ensuring the consumer waits if the buffer is empty.
- **sem_space:** Tracks available space, ensuring the producer waits if the buffer is full.
- **sem_mutex:** Protects the critical section by allowing only one thread to access the buffer at a time.

This implementation ensures proper synchronization between threads, eliminates race conditions, and handles data transfer reliably until the end-of-file (EOF) marker (`*`) is reached.

2. System Setup

- **Programming Language:** C
- **Compiler:** GCC
- **Operating System:** Linux
- **Development Environment:** Linux virtual machine
- **Development Tools:** Visual Studio Code

3. Implementation Details

3.1 Circular Buffer

A circular buffer of size 15 is used to exchange data between the producer and consumer. The buffer supports continuous data flow, with the producer adding characters to the buffer and the consumer retrieving them.

3.2 Semaphore Implementation

- **sem_items:** Incremented by the producer when data is added, decremented by the consumer when data is retrieved.
- **sem_space:** Decremented by the producer when adding data, incremented by the consumer when removing data.
- **sem_mutex:** Ensures that the producer and consumer do not access the buffer simultaneously, preventing race conditions.

3.3 Producer Thread

The producer reads characters from `mytest.dat` and writes them to the buffer. Upon completion, the producer writes an EOF marker (`*`) to signal the consumer that no more data is available.

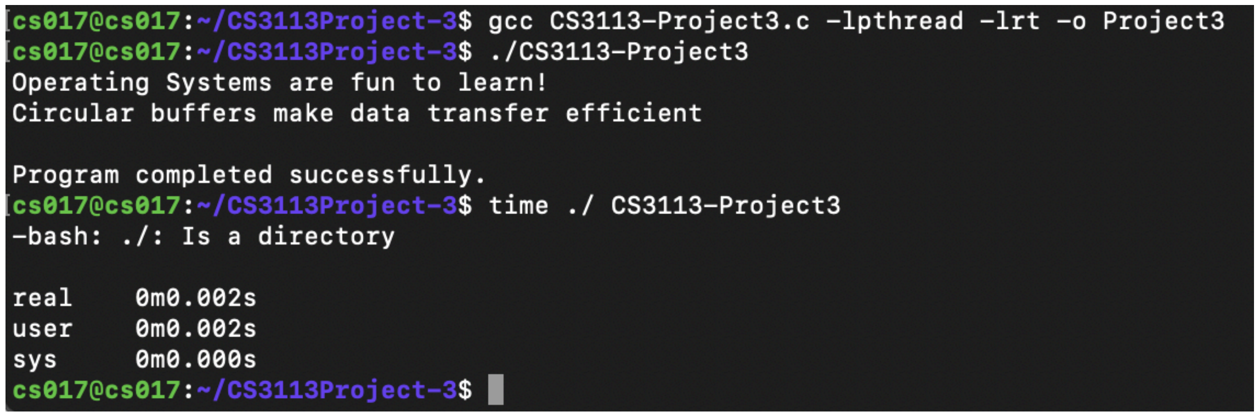
3.4 Consumer Thread

The consumer retrieves data from the buffer and prints it to the screen with a delay of one second between reads. The consumer stops processing when it encounters the EOF marker.

4. Quantitative Data

- **Buffer Size:** 15
- **File Input Size:** 150 characters (maximum)
- **Special Marker:** `*` (indicating EOF)
- **Producer Delay:** None
- **Consumer Delay:** 1 second between reads

- Execution Time (for final run):
- Real Time: 0m0.002s
- User Time: 0m0.002s
- System Time: 0m0.00s



```
cs017@cs017:~/CS3113Project-3$ gcc CS3113-Project3.c -lpthread -lrt -o Project3
cs017@cs017:~/CS3113Project-3$ ./CS3113-Project3
Operating Systems are fun to learn!
Circular buffers make data transfer efficient

Program completed successfully.
cs017@cs017:~/CS3113Project-3$ time ./ CS3113-Project3
-bash: ./: Is a directory

real    0m0.002s
user    0m0.002s
sys     0m0.000s
cs017@cs017:~/CS3113Project-3$
```

5. Code Overview

Input File

Filename: `mytest.dat`
Content:

```
Operating Systems are fun to learn!
Circular buffers make data transfer efficient
```

This file contains a string of fewer than 150 characters, as required. The producer reads this file character by character.

Program Output

Expected Output:

```
Operating Systems are fun to learn! Circular buffers make data
transfer efficient
Program completed successfully.
```

Each character is printed one at a time with a delay between consecutive characters due to the simulated slower consumer.

Key Observations

Thread Synchronization

- The semaphores ensure smooth and conflict-free data exchange between the producer and consumer.
- The circular buffer effectively manages the limited resource, allowing continuous data flow even with differing production and consumption speeds.

Concurrency

- The program demonstrates effective use of threads to perform producer and consumer tasks concurrently.

Scalability

- The modular design of the buffer and semaphore-based synchronization makes the program extensible for larger buffers or more complex producer-consumer models.

Challenges and Solutions

EOF Handling

- Challenge: Signaling the consumer when the producer finishes reading the file.
- Solution: Use a special EOF marker (`*`) placed in the buffer by the producer.

Synchronization Issues

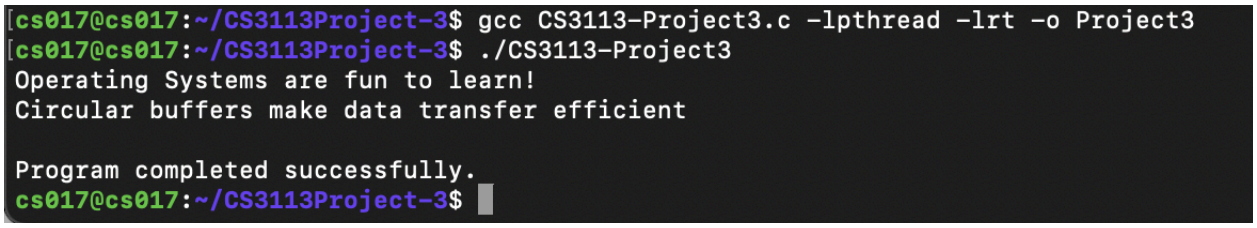
- Challenge: Preventing data race conditions and ensuring proper buffer access.
- Solution: Mutex semaphore (`sem_mutex`) ensures mutual exclusion during buffer access.

Buffer Size Constraints

- Challenge: Efficiently managing the limited buffer size.
- Solution: Circular indexing with modulo arithmetic ensures continuous buffer reuse.

Final Attempt (CORRECT OUTPUT)

- **Corrections:**
  - Implemented `sem_mutex` to safeguard the critical section.
  - Ensured reliable signaling of EOF using the `*` marker.
- **Outcome:**
  - Smooth operation with accurate data transfer.
  - All characters, including the EOF marker, were processed successfully without race conditions.



```
[cs017@cs017:~/CS3113Project-3$ gcc CS3113-Project3.c -lpthread -lrt -o Project3
[cs017@cs017:~/CS3113Project-3$ ./CS3113-Project3
Operating Systems are fun to learn!
Circular buffers make data transfer efficient

Program completed successfully.
cs017@cs017:~/CS3113Project-3$
```

6. Conclusion

This project successfully demonstrates the use of semaphores to synchronize threads and protect shared resources. By enforcing mutual exclusion and proper signaling, semaphores ensured orderly execution and data integrity.

Key Outcomes:

- The **producer** efficiently added data to the buffer, signaling when done.
- The **consumer** accurately retrieved and processed data until EOF.
- **Synchronization** prevented race conditions and ensured proper execution order.

Future work could explore alternative synchronization mechanisms, such as condition variables, to improve performance in high-throughput scenarios. Additionally, integrating robust error handling for semaphore operations would enhance reliability further.