Mini Project: Multithreaded Producer-Consumer Application

College of Computer Engineering and Science

American University of Sharjah

Fall 2025

Professor Shadi Banitaan

CMP310: Operating Systems

Section 2

Name	ID
Farah Tawalbeh	g00099768
Ahmad Al-Nakaleh	b00095454
Shaher Ghrewati	b00100405
Yafa Asia	g00100474

Introduction

This project implements a multithreaded Producer–Consumer system using POSIX threads, mutexes, and semaphores to ensure safe, concurrent access to a shared bounded buffer. The objective of the project is to design a reliable synchronization mechanism that allows multiple producer threads to generate items while multiple consumer threads remove and process them without race conditions or data corruption. A circular buffer structure is used to maximize memory efficiency and support continuous insertion and removal of items. Each producer creates a fixed number of items, some marked with higher urgency, and timestamps are recorded to measure per-item latency and overall system throughput. Consumers process items concurrently until they receive termination signals implemented through the poison-pill technique, ensuring a controlled and graceful shutdown of all threads. The program also records the total number of produced and consumed items, average latency, and throughput, establishing how thread interactions and buffer size impact overall system performance. All in all, the project focuses on efficient thread synchronization, effective buffer handling, safe shutdown of threads, and evaluating performance under concurrent execution.

Implementation decisions

1. Using a Circular Bounded Buffer

The shared buffer was implemented as a circular queue to allow continuous insertion and removal of items without shifting memory. By updating the in and out indices using modular arithmetic, the buffer supports fast, constant-time operations and makes efficient use of the fixed storage space:

```
shared.in = (shared.in + 1) % shared.size;
shared.out = (shared.out + 1) % shared.size;
```

This approach keeps the buffer lightweight, predictable, and well-suited for the steady flow of items between producers and consumers.

2. **Synchronization with one Mutex and two Semaphores**

To ensure safe access to shared memory and avoid race conditions, the program uses one mutex (`shared.mutex`) to protect critical sections (buffer insert/remove), one semaphore (`shared.empty`) that is initialized to the buffer size to track available slots, and one semaphore (`shared.full`) that is initialized to zero to track the number of filled slots.

In our code, we have:

Producer:

```
sem_wait(&shared.empty);
pthread_mutex_lock(&shared.mutex);
pthread_mutex_unlock(&shared.mutex);
sem_post(&shared.full);
```

Consumer:

```
sem_wait(&shared.full);
pthread_mutex_lock(&shared.mutex);
pthread_mutex_unlock(&shared.mutex);
sem_post(&shared.empty);
```

This design ensures correct blocking behavior, because the producers block when the buffer is full, whilst the consumers block when the buffer is empty. This is the classic solution to the Producer–Consumer problem and prevents busy waiting.

3. **Fixed Production Count Per Producer**

Each producer generates a fixed number of items (`ITEMS_PER_PRODUCER = 20`), and therefore, the total expected number of real items is:

```
total_items = producers × ITEMS_PER_PRODUCER
```

With a set number of items per producer, the total workload is straightforward to calculate and simplifies the transition to the termination phase.

4. **Poison-Pill Termination Strategy**

The program uses a poison-pill mechanism to cleanly terminate consumer threads. After all producers finish, the main thread inserts one poison pill (`POISON_PILL = -1`) for each consumer. Each consumer terminates as soon as they read this special item. The consumer check:

```
if (item.data == POISON_PILL) { ... break; }
```

This prevents any deadlocks, guarantees that no consumers will be waiting indefinitely, and ensures that all threads will exit. There will be no need for forced thread cancellation.

5. **Thread Safe Random Number Generation**

This design supports the bonus requirement for performance metrics. Producers use `rand_r()` instead of `rand()` to avoid conflicts over shared global state. Each producer is given its own seed (calculated using the current time and its thread ID) to ensure unique and thread-safe random number generation.

```
unsigned int seed = time(NULL) + id;
    item.data = rand_r(&seed) % 1000;
item.priority = (rand_r(&seed) % MAX_URGENT_RATIO == 0) ? 1 : 0;
```

This guarantees that each producer generates its own independent sequence of random values, prevents race conditions, and provides the variability needed for measuring latency, throughput, and item priority distribution.

6. **Latency and Throughput Measurements**

‘clock_gettime(CLOCK_MONOTONIC, ..)’, and latency measurement is done during dequeue operations. The throughput is then calculated using the total time taken from ‘start_time’ to ‘end_time’. The code meets the bonus requirement of implementing performance metrics and enables the measurement of efficiency of the system for various buffer size and thread numbers.

```
clock_gettime(CLOCK_MONOTONIC, &item.enqueue_time);  // producer
...
clock_gettime(CLOCK_MONOTONIC, &dequeue_time);      // consumer
double latency = timespec_diff(&item.enqueue_time, &dequeue_time);
```

This ensures that each item will record its precise enqueue and dequeue times, which allows accurate calculations for the latency per item. Using CLOCK_MONOTONIC guarantees that measurements are immune to system clock changes, providing accurate time for the upcoming performance metrics.

```
printf("Average latency: %.3f seconds\n", shared.total_latency /
                    shared.items_consumed);
printf("Throughput: %.2f items/second\n", shared.items_consumed /
                        total_time);
```

This computes average latency and throughput for all the items processed, which allows us to examine the program's efficiency under different buffer sizes and thread counts.

7. Minimal Shared Global States

There is a single struct named `shared_state_t` that groups all shared variables. This struct serves to make the program more modular, debug-friendly, and manageable because all the essential shared data is contained within a single struct.

```
typedef struct {
buffer_item *buffer;
    int size;
  int in, out;
pthread_mutex_t mutex;
 sem_t empty, full;
 int items_produced;
```

```
    int items_consumed;
    double total_latency;
    struct timespec start_time, end_time;
} shared_state_t;

static shared_state_t shared;  // global access for all threads
```

This ensures that all shared variables are encapsulated into a single struct, which clarifies which resources are shared. This structure also simplifies mutex and semaphore operations, reducing the chance of race condition.

```
pthread_mutex_lock(&shared.mutex);
    shared.items_consumed++;
shared.total_latency += latency;
pthread_mutex_unlock(&shared.mutex);

    sem_wait(&shared.empty);
    sem_post(&shared.full);
```

This code is important as it does the following:

- The mutex used ensure that updates to shared statistics (items_consumed, total_latency) are atomic and race-free
- The semaphores guarantee that producers and consumers will wait for when the buffer is full or empty, which ensures proper synchronization and prevents deadlocks or busy waiting.

Challenges

During the development stage of the project, there were a few issues that came up that needed to be tackled very carefully. Firstly, the updating of shared statistics such as items_consumed and total_latency did not involve the usage of mutex locks, resulting in a race condition when accessed concurrently by multiple consumer threads. This issue got resolved by using the same mutex used for buffer access to safeguard all the updates. The second issue came up during the creation of the consumer and producer threads and the synchronization of the program. If the producer threads were created first, the program could potentially face a deadlock. This got resolved when error checks and the synchronization of the start of all the consumer and producer threads were implemented. The final issue lay within the calculation of the throughput and latency of performance measurement, which was impacted because of the inconsistency introduced because of the usage of time(NULL). The usage of CLOCK_MONOTONIC helped get the appropriate and accurate timing readings.

Results table

Buffer Size	**Number of threads (producer + consumer threads)**	**Average Latency (s)**	**Throughput (items/s)**
10 (recommended example in the instructions given)	59+ 59= 118	0.846	13.41
20	99+ 99 = 198	1.080	19.99
4	79+ 79= 158	0.325	24.9
50 (large buffer)	399+399= 798	0.559	127.32

2 (small buffer)	19+19= 38	0.474	5.76

Interpretation:

Generally, Smaller buffers result in low latencies but moderate throughput due to frequent blockings. As for medium to large buffers, latency increases as items wait longer, and throughput slightly improves with better thread utilization. Fluctuations do, however, occur due to various reasons, including the number of producers and consumers, and others. For example, Smaller buffers lead to moderate throughput due to the fact that they enable faster handoffs between producers and consumers, keeping threads continuously active, which will temporarily increase the throughput. On the other hand, when it comes to medium to large buffers, throughput in larger buffers may reduce slightly due to the immediate processing rate. Overall, buffer size leads to a latency and throughput tradeoff.

Conclusion

This project successfully implements a multithreaded Producer–Consumer model using POSIX threads, mutexes, and semaphores to ensure safe, concurrent access to a shared bounded buffer. The system meets its core goals of maintaining synchronization correctness, preventing race conditions, and ensuring efficient use of shared resources. The use of poison pills provides a controlled shutdown process for all consumer threads, reinforcing the importance of proper termination handling in concurrent applications. Additionally, including the recorded measurements of latency and throughput is beneficial as it allows for performance analysis, clearly showing how buffer size and thread interactions influence overall execution behavior. Overall, the project demonstrates a solid understanding of operating system synchronization mechanisms, thread-safe programming practices, and performance evaluation within a multithreaded environment.