CS 3502: Operating Systems

Assignment 3: Bounded Buffer Problem

Shared Memory & Semaphore Synchronization

Submission: Via D2L and Git repository

1 Overview

Building on your IPC work from Assignment 2, you'll now implement the classic **bounded buffer problem** using shared memory and semaphores. This assignment teaches critical synchronization concepts used in operating systems, databases, and distributed systems.

The Challenge: Multiple producer processes generate data while multiple consumer processes read it. You must coordinate access to a shared circular buffer without data corruption or deadlock.

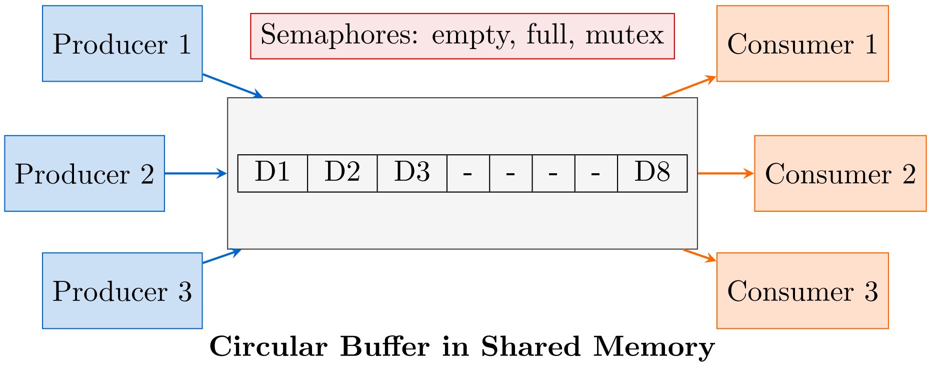

Figure 1: The Bounded Buffer Architecture

2 Learning Objectives

- Implement shared memory segments using System V IPC
- Apply semaphore synchronization to prevent race conditions
- Design and debug concurrent programs with multiple processes
- Understand the producer-consumer synchronization pattern

3 Technical Specifications

3.1 System Architecture

You will implement a producer-consumer system with the following components:

Component	Description
`buffer.h`	Shared data structures and constants
`producer.c`	Producer process implementation
`consumer.c`	Consumer process implementation

4 Requirements

4.1 Core Implementation

You will implement THREE files that work together:

4.1.1 buffer.h - Shared Definitions

Define the shared buffer structure and constants:

```c
#define BUFFER_SIZE 10
#define SHM_KEY 0x1234

typedef struct {
    int value;            // Data value
    int producer_id;      // Which producer created this
} item_t;

typedef struct {
    item_t buffer[BUFFER_SIZE];
    int head;             // Next write position
    int tail;             // Next read position
    int count;            // Current items in buffer
} shared_buffer_t;
```

4.1.2 producer.c - Producer Process

Implement a producer that:

- Takes command line args: `./producer <id> <num_items>`
- Attaches to shared memory using `shmget()` and `shmat()`
- Opens named semaphores for synchronization
- Produces `num_items` with unique sequential values
- Uses proper semaphore protocol (wait for empty, lock mutex, produce, unlock, signal full)
- Prints each production: `"Producer 1: Produced value 42"`

4.1.3 consumer.c - Consumer Process

Implement a consumer that:

- Takes command line args: `./consumer <id> <num_items>`
- Attaches to the same shared memory segment
- Opens the same named semaphores
- Consumes exactly `num_items` (or until timeout)
- Uses proper semaphore protocol (wait for full, lock mutex, consume, unlock, signal empty)
- Prints each consumption: `"Consumer 1: Consumed value 42 from Producer 2"`

4.2 Synchronization Requirements

Your solution MUST use three named semaphores:

- `/sem_mutex` - Binary semaphore for mutual exclusion (init: 1)
- `/sem_empty` - Counting semaphore for empty slots (init: BUFFER_SIZE)
- `/sem_full` - Counting semaphore for full slots (init: 0)

4.3 Testing Requirements

Your code must correctly handle these scenarios:

1. Single producer, single consumer
2. Multiple producers (3+), single consumer
3. Single producer, multiple consumers (2+)
4. Multiple producers and consumers running simultaneously
5. No data loss or corruption in any scenario

5 Implementation Guide

5.1 Step 1: Shared Memory Setup

```c
// Create or attach to shared memory
int shm_id = shmget(SHM_KEY, sizeof(shared_buffer_t),
                    IPC_CREAT | 0666);
if (shm_id < 0) {
    perror("shmget-failed");
    exit(1);
}

shared_buffer_t* buffer = (shared_buffer_t*)shmat(shm_id, NULL, 0);
if (buffer == (void*)-1) {
    perror("shmat-failed");
    exit(1);
}
```

5.2 Step 2: Semaphore Setup

```c
// Create or open named semaphores
sem_t* mutex = sem_open("/sem_mutex", O_CREAT, 0644, 1);
sem_t* empty = sem_open("/sem_empty", O_CREAT, 0644, BUFFER_SIZE);
sem_t* full = sem_open("/sem_full", O_CREAT, 0644, 0);

if (mutex == SEM_FAILED || empty == SEM_FAILED || full == SEM_FAILED) {
    perror("sem_open-failed");
    exit(1);
}
```

5.3 Step 3: Producer Logic

```c
for (int i = 0; i < num_items; i++) {
    item_t item;
    item.value = producer_id * 1000 + i;  // Unique value
    item.producer_id = producer_id;

    sem_wait(empty);  // Wait for empty slot
    sem_wait(mutex);  // Enter critical section

    // Add to buffer
    buffer->buffer[buffer->head] = item;
    buffer->head = (buffer->head + 1) % BUFFER_SIZE;
    buffer->count++;

    printf("Producer %d: Produced value %d\n",
           producer_id, item.value);

    sem_post(mutex);  // Exit critical section
    sem_post(full);   // Signal item available

    usleep(rand() % 100000);  // Simulate work
}
```

5.4 Step 4: Consumer Logic

```c
for (int i = 0; i < num_items; i++) {
    sem_wait(full);   // Wait for item
    sem_wait(mutex);  // Enter critical section

    // Remove from buffer
    item_t item = buffer->buffer[buffer->tail];
    buffer->tail = (buffer->tail + 1) % BUFFER_SIZE;
    buffer->count--;

    printf("Consumer %d: Consumed value %d from Producer %d\n",
           consumer_id, item.value, item.producer_id);

    sem_post(mutex);  // Exit critical section
    sem_post(empty);  // Signal slot available

    usleep(rand() % 100000);  // Simulate work
}
```

6 Testing Your Implementation

6.1 Basic Test Commands

```
# Compile
gcc -o producer producer.c -pthread -lrt
gcc -o consumer consumer.c -pthread -lrt

# Test 1: Basic functionality
./producer 1 5 &
./consumer 1 5 &
wait

# Test 2: Multiple producers
./producer 1 10 &
./producer 2 10 &
./producer 3 10 &
./consumer 1 30 &
wait

# Test 3: Multiple consumers
./producer 1 20 &
./consumer 1 10 &
./consumer 2 10 &
wait

# Cleanup (if needed)
ipcrm -M 0x1234
rm /dev/shm/sem_*
```

6.2 Expected Output Example

```
Producer 1: Produced value 1000
Producer 2: Produced value 2000
Consumer 1: Consumed value 1000 from Producer 1
Producer 1: Produced value 1001
Consumer 1: Consumed value 2000 from Producer 2
Consumer 2: Consumed value 1001 from Producer 1
...
```

7 Common Issues and Debugging

Problem	**Solution**
Deadlock	Check semaphore order: always empty → mutex → produce → mutex → full
Segmentation fault	Verify shared memory attachment and pointer validity
Wrong values consumed	Ensure head/tail updates are inside critical section
Semaphore errors	Check permissions, use `ipcs -s` to view semaphores
Buffer corruption	All buffer access must be protected by mutex

8 Submission Requirements

1. Submit THREE files to D2L: `buffer.h`, `producer.c`, `consumer.c`
2. Include your Git repository URL in the D2L submission comments
3. Ensure your code compiles with: `gcc -Wall -pthread -lrt`
4. Include your name and section in header comments of each file

9 Academic Integrity

This is an individual assignment. You may discuss concepts with classmates, but all code must be your own. Use of AI code generation tools is prohibited. However, debugging code with AI is only allowed when you cite any references when used.