📋 Homework-7: Multi-Threaded Number Processing Pipeline using Synchronization Primitives

⚠️ **Important:** This homework focuses on implementing the Round-Robin (RR) CPU scheduling algorithm. Students will work with process scheduling concepts including arrival times, burst times, time quantum, waiting times, and turnaround times.

💡 All the necessary concepts required to complete this homework are covered in the **CPU Scheduling lecture materials and textbook chapter on scheduling algorithms.**

You must use these strict gcc compilation flags while building your application:
-std=c17
-D_POSIX_C_SOURCE=200809L
-Wall
-Wextra
-Werror
-g
-O0

⚠️ **There is only one submission - RESUBMISSION IS NOT ALLOWED. You must test your implementation with the provided testing framework on the Ocelot server before the final submission.**

🔧 Grader Contact Information

For any questions or concerns related to assignment grading, please reach out to the TA listed below.

Communication Guidelines:

- Ensure that all communication is polite and respectful
- **You must contact the Graders first for any grading issues**
- If the matter remains unresolved, feel free to reach out to the instructor

Name: Instructor – bbhatkal@fiu.edu (⚠️ via the Canvas inbox only)
Section: COP 4610-U01 (84957)

Name: Angie Martinez – amart1070@fiu.edu (⚠️ via the Canvas inbox only)
Section: COP 4610-U02 (85031)

Name: Srushti Visweswaraiah – svisw003@fiu.edu(⚠️ via the Canvas inbox only)

Sections: COP 4610-UHA (85236)

☑ Section 1: Homework Overview

To implement a multi-threaded number processing pipeline that coordinates multiple threads using mutexes and semaphores to safely share data across bounded buffers, ensuring correct ordering and synchronization at every stage.

This assignment involves implementing a **pipeline architecture** with three stages:

1. **Generator threads** (producers) that read from a shared input array and insert numbers into a bounded buffer (**Buffer1**).
2. **Processor threads** (intermediate producers/consumers) that remove numbers from **Buffer1**, square them, and place the results into another bounded buffer (**Buffer2**).
3. A **Writer thread** (final consumer) that removes squared results from **Buffer2** and prints them in the correct input order using a holding buffer and sequence logic.

Each stage is fully parallelized and synchronized using **POSIX semaphores and mutexes**. Correct use of synchronization primitives is essential to avoid data races, deadlocks, and incorrect outputs.

Your implementation will be tested for:

- Correct functional output
- Synchronization correctness (no race conditions or deadlocks)
- Proper thread partitioning and data flow
- Ordered final output (based on input index)

☑ Section 2: Learning Objectives

By completing this assignment, students will:

- Understand and implement the **Producer-Consumer pattern** using **bounded buffers**
- Learn how to create and manage **multiple POSIX threads** for concurrent tasks
- Apply **mutexes** to enforce mutual exclusion during shared buffer access
- Use **counting semaphores** to track buffer state (empty/full slots)
- Coordinate thread communication using synchronization primitives
- Enforce **ordered output** across asynchronous threads
- Detect and avoid common pitfalls such as:
    - Race conditions
    - Deadlocks (especially when mutexes precede semaphores in incorrect order)
    - Buffer overflows or underflows
- Design thread-safe and modular C programs using pthreads and shared data structures

☑ Section 3: Problem Description

This assignment requires you to implement a multi-threaded number processing pipeline using synchronization primitives. You will simulate a pipeline with three fixed stages: **Generators** → **Processors** → **Writer**, with data passed through two bounded buffers.

The main idea is to implement correct inter-thread coordination using **mutexes** and **semaphores** to ensure:

- Proper buffer access (no overflows or underflows)
- Mutual exclusion in critical sections
- Ordering of output based on input position (sequence ID)

Conceptual Workflow

- **Generators (3 threads)**:
  - Divide input numbers among themselves (each thread gets ~N/3 numbers)
  - Each thread inserts `NumberData` structs (number + ID) into BUFFER1
- **Processors (3 threads)**:
  - Consume from BUFFER1
  - Compute `number * number` (square)
  - Insert result as `NumberData` (squared value + same ID) into BUFFER2
  - When finished, signal completion using `sem_post(&processor_done)`
- **Writer (1 thread)**:
  - Collect results from BUFFER2
  - **Buffer out-of-order entries internally**
  - Print results to `stdout` **in input ID order**
  - Waits for all 3 processor threads to finish (via 3 `sem_wait(&processor_done)`)

System Architecture

```
┌──────────────────────────────────────────────────────┐
│  GENERATOR THREADS (3 threads - FIXED)               │
│  Role: Producers for BUFFER1                         │
│  Task: Generate numbers 1, 2, 3, ..., N              │
│        Each generator gets N/3 numbers to generate   │
└──────────────────────────────────────────────────────┘
                            ↓
┌──────────────────────────────────────────────────────┐
│  BUFFER1 (Bounded Buffer)                            │
│  Capacity: 10 slots                                  │
│  Each slot: NumberData struct (number + id)          │
│  Synchronization:                                    │
│    - sem_t empty1   // Counting semaphore (init: 10) │
│    - sem_t full1    // Counting semaphore (init: 0)  │
│    - pthread_mutex_t mutex1 // Binary semaphore      │
└──────────────────────────────────────────────────────┘
```

```
                                ↓
┌──────────────────────────────────────────────────────────────┐
│  PROCESSOR THREADS (3 threads - FIXED)                       │
│  Role: Consumers of BUFFER1, Producers for BUFFER2           │
│  Task:                                                       │
│    1. Remove number from BUFFER1 (consumer)                  │
│    2. Apply operation: result = number * number (square)     │
│    3. Add result to BUFFER2 (producer)                       │
└──────────────────────────────────────────────────────────────┘
                                ↓
┌──────────────────────────────────────────────────────────────┐
│  BUFFER2 (Bounded Buffer)                                    │
│  Capacity: 10 slots                                          │
│  Each slot: NumberData struct (result + id)                  │
│  Synchronization:                                            │
│    - sem_t empty2   // Counting semaphore (init: 10)         │
│    - pthread_mutex_t mutex2 // Binary semaphore              │
└──────────────────────────────────────────────────────────────┘
                                ↓
┌──────────────────────────────────────────────────────────────┐
│  WRITER THREAD (1 thread - FIXED)                            │
│  Role: Consumer for BUFFER2                                  │
│  Task: Remove results from BUFFER2 → write to output file    │
│        Must write results in CORRECT ORDER (by id)           │
└──────────────────────────────────────────────────────────────┘
                                ↓
                    Output File: results.txt
┌──────────────────────────────────────────────────────────────┐
│  COMPLETION SIGNALING                                        │
│  Semaphore: processor_done (init: 0)                         │
│  - Each processor: sem_post(&processor_done) when done       │
│  - Writer: sem_wait(&processor_done) 3 times                 │
└──────────────────────────────────────────────────────────────┘
```

⚠️ ⚠️ CRITICAL RULES:

1. **ALWAYS** `sem_wait` BEFORE `mutex_lock` (prevents deadlock)
2. **ALWAYS** `mutex_unlock` BEFORE `sem_post`
3. **ALWAYS** use circular indexing: `(index + 1) % BUFFER_SIZE`

Example Execution

Input:

```
./pipeline 10 output.txt
# Process numbers 1 through 10
```

Internal Flow:

```
Generator 0: generates 1, 2, 3, 4    → BUFFER1
Generator 1: generates 5, 6, 7       → BUFFER1
Generator 2: generates 8, 9, 10      → BUFFER1

Processor 0: gets 1 → squares → 1    → BUFFER2
Processor 1: gets 5 → squares → 25   → BUFFER2
Processor 2: gets 3 → squares → 9    → BUFFER2
Processor 0: gets 2 → squares → 4    → BUFFER2
... (order varies due to parallelism)

Writer: writes in ORDER by id:
  1 → 1
  2 → 4
  3 → 9
  4 → 16
  5 → 25
  ...
  10 → 100
```

Output File (results.txt):

```
Input: 1, Result: 1
Input: 2, Result: 4
Input: 3, Result: 9
Input: 4, Result: 16
Input: 5, Result: 25
Input: 6, Result: 36
Input: 7, Result: 49
Input: 8, Result: 64
Input: 9, Result: 81
Input: 10, Result: 100
```

☑ Section 4: Provided Files Framework

📖 **Refer to these files** for understanding the exact requirements.

⚠️ **Please do not make any modifications to the framework files provided.** You are required to implement **only** the file `submission_HW7.c`. Any changes to the framework files may result in errors during evaluation or loss of credit.

Folder Structure

```
PROVIDED_FILES/
├── skeleton_submission_HW7.c   # Implementation template file (rename to submission_HW7.c)
├── synchronization.h           # Header file (Do not modify!)
├── driver.c                    # Main driver program (Do not modify!)
├── Makefile                    # Build system (Do not modify!)
├── autograder_HW7.sh           # Testing script (Do not modify!)
├── batchgrader_HW7.sh          # Batch testing script (Do not modify!)
```

```
├── Sample_Executable/
│   └── synch                    # Sample executable for testing
└── Testing/                     # Test cases (Do not modify!)
    ├── Testcases/
    │   ├── input1.txt           # Test case 1
    │   ├── input2.txt           # Test case 2
    │   └── input3.txt           # Test case 3
    └── Expected_Output/
        ├── output1.txt          # Expected output for test 1
        ├── output2.txt          # Expected output for test 2
        └── output3.txt          # Expected output for test 3
```

☑ Section 5: Submission Requirements

File Requirements (⚠ exact names required)

- **submission_HW7.c**: Your complete Round-Robin implementation
- **README.txt or README.pdf or README.md**: Student information and documentation
    - Your README ⚠ **MUST include**:

```
# Student Information
- Full Name: [Your Full Name]
- PID: [Your FIU Panther ID]
- Section: [Your Course Section]
- FIU Email: [Your @fiu.edu email]

# Homework: Number Processing Pipeline
[Brief description of your implementation approach]
[Describe how you implemented the synchronization for buffer1 and buffer2]
[Explain your use of semaphores (empty/full) and mutex locks]
[Discuss how you prevented race conditions and deadlocks]
[Mention any challenges faced and how you solved them, For example: deadlock issues, race
conditions, incorrect output ordering, etc.]
```

Deliverable Folder (`ZIP file`) Structure - ⚠ exact structure required - no additional files/subfolders

> ⚠ **DO NOT submit any other files other than these required files. The batch autograder may treat any additional items in the ZIP file as invalid, which will result in a grade of zero: submission_HW7.c, README.txt (or README.pdf or README.md)**
>
> 📁 All other required framework files will be supplied by the instructor to the autograder during final grading.

- ⚠ You are required to submit your work as **a single ZIP archive file**.
- ⚠ **Filename Format**: your `Firstname_Lastname.zip` (**exact format required**).
- **You will be held responsible for receiving a ZERO grade if the submission guidelines are not followed**.

```
Harry_Potter.zip
├── submission_HW7.c  # Your Round-Robin scheduler implementation
└── README.txt        # Your details and implementation approach
```

☑️ Section 6: Developing and Testing Your Implementation

> ⚠️ **DO NOT modify any following provided framework files:**
> autograder_HW7.sh, batchgrader_HW7.sh, synchronization.h, driver.c, Makefile, and Testing/ folder.

You must implement this function in `submission_HW7.c`.

- 💡 **Refer to the detailed TODO comments in the skeleton code for specific implementation tasks**.
- 💡 The provided `driver.c` file handles all **file processing tasks**, including reading and writing.

1. **void init_buffer(BoundedBuffer *buf, int capacity)**;
2. **void buffer_add(BoundedBuffer *buf, NumberData data);**
3. **NumberData buffer_remove(BoundedBuffer *buf);**
4. **void destroy_buffer(BoundedBuffer *buf)**;

STEP 1 - Sample Executable Testing:

💡 To better understand the Round-Robin implementation requirements, you are **highly encouraged** to test the instructor-provided executable:

```
# Navigate to Sample_Executable folder
cd Sample_Executable

# Provide execute permissions
chmod 777 synch

# Run with test cases (redirect input from test files)
# Repeat the process for all 3 test cases by changing the input file name accordingly.
./synch < ../Testing/Testcases/input1.txt

# Compare output with expected results
./synch < ../Testing/Testcases/input1.txt > my_output1.txt
diff my_output1.txt ../Testing/Expected_Output/output1.txt
```

STEP 2 - Manual testing:

Ensure the following files are located in the **same directory**:

- **Your implementation**: `submission_HW7.c`
- **All the provided framework files**
- **The provided testcase folder:** `Testing/`

```
# Explore the Makefile first to understand build targets and compilation settings
cat Makefile

# Build the executable
make clean
make

# Test with all eight test cases
./synch < ../Testing/Testcases/input1.txt

# Save your output for comparison
# Repeat the process for all 3 test cases by changing the input file name
# and student_output file name accordingly.
./synch < Testing/Testcases/input1.txt > student_output1.txt

# Compare your output with expected output
# Repeat the process for all 3 test cases by changing the student_output file name
# and  expected output file name accordingly.
diff student_output1.txt Testing/Expected_Output/output1.txt

# If diff shows no output, your implementation is correct!
```

STEP 3 - Autograder testing:

Ensure the following files are located in the **same directory**, and then run the **Autograder**:

- **Your implementation**: `submission_HW7.c`
- **All the provided framework files**
- **The provided testcase folder:** `Testing/`

```
# Make autograder executable
chmod +x autograder_HW7.sh

# Run the autograder
./autograder_HW7.sh
```

- **Check your grade** and fix any issues
- **Repeat until** you achieve the desired score

STEP 4 - Batch Autograder testing:

> ⚠️ **Important:** The batch autograder may take a bit of time. Please be patient!

The batch autograder, **used by the instructor**, processes all student submissions at once. It utilizes the provided framework files, extracts the required files from your submitted `.zip` file, and performs grading accordingly.

⚠️ **Backup Files Before Running Batch Grader - The script deletes existing files to rebuild the environment from scratch.**

Ensure the following files are located in the **same directory**, and then run the **Batch Autograder**:

- **All the provided framework files**
- **The provided testcase folder:** `Testing/`
- **Your final submission** `ZIP` **file** consisting the required files for the submission: example, `Harry_Potter.zip`

```
# Make batch autograder executable
chmod +x batchgrader_HW7.sh

# Run the batch-autograder
./batchgrader_HW7.sh
```

Final Testing requirements:

- ⚠️ **MUST test on ocelot server**: `ocelot-bbhatkal.aul.fiu.edu`
- ⚠️ **Test thoroughly before submission** - no excuses accepted
- ⚠️ **"Works on my computer" is NOT accepted** as an excuse
- ✅ **If you pass all test cases on the server, you will likely pass instructor's final grading**
- ✅ **What you see is what you get** - autograder results predict your final grade

---

☑️ Section 7: Grading Criteria

🤖 Autograder Testing

- ⚠️ Your implementation `submission_HW7.c` will be tested against the provided test cases as well as additional instructor test cases using the autograder.
- ⚠️ **Exact output matching required** - any deviation results in point deduction
- ⚠️ **Program must compile** without errors or warnings

⚖️ Test Case Distribution:

- 💉 **TEST 1: Functionality & Correctness Tests (75 points):**
  - Test Case 1.1 (input1.txt) - *25 points*
  - Test Case 1.2 (input2.txt) - *25 points*
  - Test Case 1.3 (input3.txt) - *25 points*
- 💉 **TEST 2: Synchronization Tests (15 points)**:
  - Test 2.1: Deadlock detection (10 runs of test case 1) - *8 points*
  - Test 2.2: Race condition detection (20 runs for consistency) - *7 points*
- 👩‍🏫 **Manual Grading (10 points)**

⛔ Penalties

- ⚠️ **Missing README**: **-10 points**
- ⚠️ **Missing** `submission_HW7.c` : **ZERO grade** (autograder compilation failure)
- ⚠️ **Incorrect ZIP filename**: **ZERO grade** (autograder extraction failure)
- ⚠️ **Wrong source filename**: **ZERO grade** (autograder compilation failure)
- 🔒 **Resubmission**: **NOT ALLOWED**

☑️ Section 8: Technical Specifications

Data Structures defined in the header file `synchronization.h`

NumberData Structure:

```
typedef struct {
    int number;      // The actual number (input or result)
    int id;          // Sequence number (0, 1, 2, ... N-1)
} NumberData;
```

Bounded Buffer Structure:

```
typedef struct {
    NumberData buffer[BUFFER_SIZE];  // 10 slots
    int in;                          // Producer index
    int out;                         // Consumer index
    sem_t empty;                     // Counting semaphore: empty slots
    sem_t full;                      // Counting semaphore: full slots
    pthread_mutex_t mutex;           // Mutex: buffer protection
} BoundedBuffer;
```

☑️ Section 9: Academic Integrity

- 👤 **This is an individual assignment**
- 💬 **You may discuss concepts** but not share code
- 📝 **All submitted code** must be your original work
- 📚 **You are encouraged to refer to textbook and lecture materials** but must write your own implementation
- ⚠️ **Plagiarism** is a serious offense and will result in penalties - **ZERO grade** (academic integrity violation).

🚀 **Good luck with your number generator pipeline implementation!**