**ELEC 377- Operating Systems**

**Lab 3: Producer-Consumer Problem**

**Design Document**

**Bradley Stephen & Ben Wyand**

## Overview

This program solves the producer-consumer problem using pthreads, mutexes, and condition variables to coordinate multiple producer and consumer threads. The program has a shared buffer where producer threads store data items, and consumer threads take those items out. This setup helps us understand thread synchronization, avoiding race conditions, and managing shared resources.

## How the Program Works

The program has producer and consumer functions that run as separate threads. These threads are managed using mutexes and condition variables to make sure everything stays in sync. Producers read data from input files and put it into a shared buffer, while consumers take data from the buffer and write it to output files.

- Main Function: The main() function sets up global variables, creates producer and consumer threads, and waits for all threads to finish. It takes command line arguments for the test number, the number of producers, and the number of consumers. Based on these inputs, it starts the corresponding threads, with each thread responsible for specific files.
- Producer Function (producer()): Each producer thread reads data from an input file line by line and tries to add it to the shared buffer.
    - Buffer Management: Before adding data, the producer checks if the buffer is full. If it is full, the producer waits (pthread_cond_wait(&full, &mutex)) until a consumer makes space by taking an item out.
    - After adding data, the producer signals that the buffer is no longer empty (pthread_cond_signal(&empty)). When a producer finishes reading all its data, it decreases the numProdRunning count and broadcasts to let consumers know if it's the last producer.

- Consumer Function (consumer()): Each consumer thread takes data from the buffer and writes it to an output file.
    - Buffer Management: The consumer checks if the buffer is empty and waits if needed using pthread_cond_wait(&empty, &mutex). When data is available, the consumer takes an item, writes it to the output file, and signals that space in the buffer is now available (pthread_cond_signal(&full)).
    - The consumer stops when there are no more producers running and the buffer is empty, making sure all produced items are processed before ending.
- Simulating Interrupts: The simulate_interrupt() function uses sched_yield() with a 33% chance to simulate interrupts. This adds randomness and helps test thread safety by forcing context switches between threads.

## Special Features

- Pthread Synchronization Primitives: We use pthread mutexes and condition variables to make sure only one thread accesses the shared buffer at a time, which helps prevent race conditions.
- Condition Variables (pthread_cond_t): These let producers and consumers signal each other when the buffer is full or empty, making resource sharing more efficient.
- Thread Safety: The program uses pthread_mutex_lock() and pthread_mutex_unlock() to manage access to shared variables, keeping operations thread-safe.

## Buffer Implementation

The buffer is an array with a fixed size (#define numSlots 3). The variables head, tail, and numElements keep track of the buffer's state:

- head: Points to the next spot where a producer can add data.
- tail: Points to the next spot where a consumer can take data.

- numElements: Keeps track of how many elements are currently in the buffer, ensuring we don’t overfill or under-utilize it.