Dana Maloney

# Operating Systems Assignment 2

Due 10/28/2024
(Late submission with extension)

# Objective:

The purpose of this project is demonstrate different thread/resource management practices, specifically the

# Functions Used:

## provider:

The provider function is called once on the provider thread. This function generates random numbers to be 'bought' by the other threads. (The random numbers are seeded off of the thread's pthread id.) When adding its generated numbers to the buffer, a mutex is used to lock the buffer. Semaphores are used to indicate when the buffer is empty or populated.

## buyer:

The buyer function is called on every user thread: The function tries to lock the buffer, consume an item, and then release the buffer. It uses semaphores to check if the buffer has items in it, and also uses them to indicate when a space is empty in the buffer.

## main:

Main is used to create all of the threads. It reads a command-line integer to determine how many consumer threads are made. Alongside creating all of the consumer/producer threads, it creates an array of IDs for all of the consumers. Pointers to the IDs are passed to the respective consumer threads. The IDs are used so it can easily be identified what threads are 'buying' items from the buffer and how many consumers/buyers there are.

# Results:

## Part 1:

test_sem3 output:

```
Compiled test_sem3.c . Running...
Creating Thread: Thread A
Thread A: INSERT item to BUFFER 1
Thread A: INSERT item to BUFFER 2
Thread A: INSERT item to BUFFER 3
Thread A: INSERT item to BUFFER 4
Thread A: INSERT item to BUFFER 5
Thread A: INSERT item to BUFFER 6
Creating Thread: Thread C
Creating Thread: Thread B
Thread C: REMOVE from BUFFER: 6
Thread B: REMOVE from BUFFER: 6
Thread C: REMOVE from BUFFER: -2
Thread B: REMOVE from BUFFER: -2
Thread B: REMOVE from BUFFER: -4
Thread C: REMOVE from BUFFER: -4
Terminate => Thread A, Thread B, Thread C!!!
Final Index: -6
Output from test_sem3.c concluded.
```

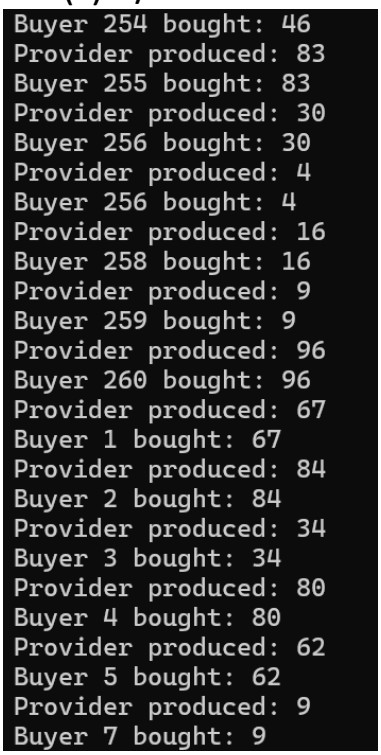
test_sem4 output:

```
Compiled test_sem4.c . Running...
Creating Thread: Thread A
Thread A: INSERT item to BUFFER 1
Thread A: INSERT item to BUFFER 2
Thread A: INSERT item to BUFFER 3
Thread A: INSERT item to BUFFER 4
Thread A: INSERT item to BUFFER 5
Thread A: INSERT item to BUFFER 6
Creating Thread: Thread B
Creating Thread: Thread C
Thread B: REMOVE item from BUFFER 6
Thread B: REMOVE item from BUFFER -1
Thread B: REMOVE item from BUFFER -2
Thread C: REMOVE item from BUFFER -3
Thread C: REMOVE item from BUFFER -4
Thread C: REMOVE item from BUFFER -5
Terminate => Thread A, Thread B, Thread C!!!
Final Index: -6
Output from test_sem4.c concluded.
```

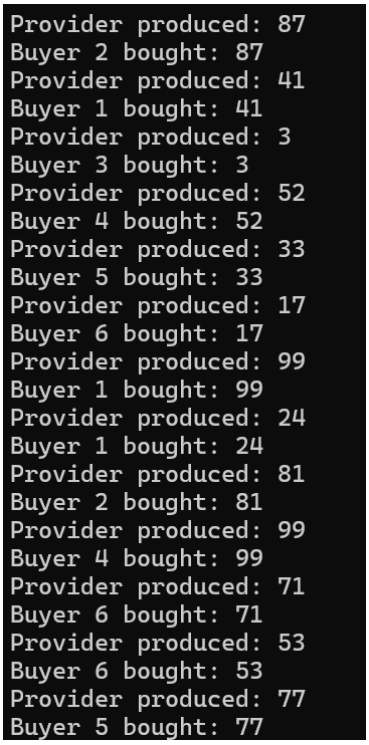
It can be observed that in test_sem3.c, Threads B and C take turns performing operations. However in test_sem4.c, threads B and C perform all of their respective operations at once before another thread can access the resources. This is because test_sem4 uses a mutex and semaphore, while test_sem3 uses a semaphore only. The semaphore is used to ensure resources are not being used by other processes/threads in both programs, but the mutex reserves a shared resource so only one thread can

access it at a time. In sem_4.c, thread B gets access to all of the resources before thread C because of the aforementioned mutex.

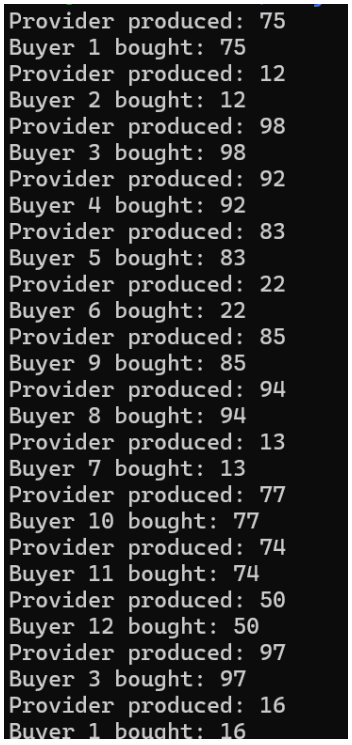
## Part(s) 2/3:



Left: Part 2, 260 consumers.



Center: Part 3, 6 consumers.



Right: Part 3, 12 consumers.

One thing that could be done to make the code faster is to create multiple buffers and thread pools with subsets of consumer threads. Another thing that could be done would be to expand the buffer, so more threads can consume data at once (assuming the producer can produce fast enough to meet demand).

# Observations/Discussions:

One thing that I noticed is that the consumer threads consistently seem to ‘buy’ things in order from thread 0 to n, for ‘n’ consumer threads. However, as all of the threads complete their first cycle, this pattern tends to break down as the consumers all sleep for unique amounts of time.

Using a combination of semaphores and mutexes help prevent errors from accessing uninitialized spots in the buffer as well as ensuring there aren’t any race conditions where multiple consumers ‘buy’ the same value.

# Conclusions:

Overall, this lab was successful in teaching about how to manage resources shared between threads for simple applications like a buffer. The content of this lab could be applied in real life for something such as a system taking in data from multiple sensors, which can split the processing of this data up between multiple threads (potentially running on multiple cores with proper utilization).

# Source Code:

GitHub Repo Link: https://github.com/danamon2002/OS_Lab2.git

```
/* Author: Dana Maloney
   Operating Systems FA24 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <pthread.h>
#include <signal.h>

#define BUFFER_SIZE 10
int buffer[BUFFER_SIZE];
int buffer_items = 0;  // # items in buffer
//int next_item = 0;  // next item for producer.

//semaphores for buffer status:
sem_t empty;  // track empty slots
sem_t full;   // track full slots
pthread_mutex_t buffer_mutex;  // Mutex to protect buffer access

void *provider(void *arg) {
    /*  Provider Function:
        Come up with integers to get bought. */

    //seed random num generator
    srand(pthread_self());

    while (1) {
        //int item = next_item++;  // Produce an item (just a sequential integer here)
        int item = (rand() % 100) + 1; // come up with integer 1 to 100.
        sem_wait(&empty);  // Wait if no empty slot in buffer
        pthread_mutex_lock(&buffer_mutex);  // Lock buffer access

        // Add item to the buffer
        buffer[buffer_items++] = item;
        printf("Provider produced %d.\n", item);

        pthread_mutex_unlock(&buffer_mutex);  // Unlock buffer access
        sem_post(&full);  // Signal that a new item is available in buffer

        sleep(1);  // Simulate delay in production
    }
    return NULL;
}

// Consumer function for each buyer thread
void *buyer(void *arg) {
    int buyer_id = *(int *)arg; // ID number that's human readable.

    srand(pthread_self()); // seed random number.

    while (1) {
        sem_wait(&full);  // Wait if no items in buffer.
        pthread_mutex_lock(&buffer_mutex);  // Access buffer.

        // Consume item from the buffer
        int item = buffer[--buffer_items];
        printf("Buyer %d bought %d.\n", buyer_id, item);

        pthread_mutex_unlock(&buffer_mutex);  // Release buffer.
        sem_post(&empty); // indicate open spot in buffer.

        sleep(rand() % 5 + 1);  // 1 - 5 Second delay.
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Please enter one integer argument for the number of \'buyer\' threads.");
        return 1;
    }

    int num_buyers = atoi(argv[1]);
    if (num_buyers < 1) {
        fprintf(stderr, "Please give a number >= 1 for the number of buyers.\n");
        return 1;
```

```
    }

    pthread_t provider_thread; // single provider thread
    pthread_t buyers[num_buyers]; // array of buyer threads
    int buyer_id_num[num_buyers];

    // Initialize semaphores and mutex
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&buffer_mutex, NULL);

    // Create threads.
    pthread_create(&provider_thread, NULL, provider, NULL); // Provider thread

    //create consumers with unique IDs. (buyer_id_num is an array so ints can be passed thru as pointers)
    for (int i = 0; i < num_buyers; i++) {
        buyer_id_num[i] = i + 1;
        pthread_create(&buyers[i], NULL, buyer, &buyer_id_num[i]);
    }

    // If all threads complete, join them. (Shouldn't be reached because of infinite loop "^-^ )
    pthread_join(provider_thread, NULL);
    for (int i = 0; i < num_buyers; i++) {
        pthread_join(buyers[i], NULL);
    }

    // Cleanup (Also shouldn't be reached.)
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&buffer_mutex);

    return 0;
}
```