OSP ASSIGNMENT ONE
SEMESTER TWO 2021

TESSA PODBURY
S3775931
https://github.com/tepo-design/s3775931_OS1

PROBLEM A – PRODUCERS AND CONSUMERS PROBLEM

The first problem implemented in my program is the Producers and Consumers Problem. I was overall very happy with this implementation. There were no memory leaks when running on valgrind and I believe that I have created a clear and concise solution.
I wouldn't refer to this as an issue, but perhaps a limitation depending on the context of what an end user would require as a producer and/or consumer.
In my implementation, it runs on a last in, first out system. There was no specification saying that this implementation was incorrect or not valid. However, in a real world setting it could be that further parameters would have to be passed through to the problem to ensure that a consumer is consuming the correct item for the task at hand and vice versa for the producer.

Variables

The variables required for this problem are as follows:

PRODUCER_AMOUNT: this variable refers to the number of producers required.
CONSUMER_AMOUNT: this variable refers to the number of consumers required.

pthread_mutex_t mutex: the variable named mutex is required to lock and unlock the thread process so only one thread has access to the bucket at any one time.

pthread_cond_t spotsFree: this is a conditional that is used in the program for the producer thread to wait until there are spots free in the bucket before allowing the thread to continue producing.

pthread_cont_t foodAvailable: this is a conditional that is used in the program for the consumer thread to wait until there is food available in the bucket before allowing the thread to consume.

int bucket[]: this is the bucket which holds the numbers produced
int count: this is a variable that keeps track of how many array slots of the bucket are full

int maximumIndex: the maximum index holds the number of the highest possible index of a given bucket.
int minimumIndex: the minimum index in this problem is always going to be 0 (an empty bucket)

bool runningA: this is our boolean value that allows the program to keep running. If runningA is true, the producer and consumer threads will continue indefinitely. If runningA is changed to false, the threads will exit their while loop.

Algorithm

The program starts in it's main method probA(). This is called from simulation.cpp.
At the beginning of the program, a PRODUCER_AMOUNT of producer threads are created and a CONSUMER_AMOUNT of consumers are created. The program starts with the boolean value of runningA set to true. This signifies that the threads should run within their methods.
Within the producer function the following occurs:

- We use the mutex to lock access to shared resources. This process ensures that we don't have race conditions of multiple threads trying to read and write from these shared processes.
- If the number of spots filled in the bucket is equal to the maximum number of spots in the bucket, the producer thread waits until is signalled that there are spots that have become free. By wrapping this in a while loop we ensure that no deadlocks occur as the thread will only wait if there are no spots in the bucket left and it will immediately continue once there are.
- A random number is generated and added to the first available spot in the bucket.
- We increase the count variable to show that a spot has been filled in the bucket.
- The producer thread signals that there is food available to be consumed.
- We unlock the mutex to allow other threads to access shared resources.

Within the consumer function the following occurs:

- As above, we use the mutex to lock access.
- If the bucket is empty, the consumer thread waits until it is signalled that there is food available to consumer. As above, the while loop ensures no deadlocks.
- The consumer thread eats the last number to be put in the bucket.
- The consumer thread signals that there are spots free in the bucket and therefore the count variable is decreased.
- We unlock the mutex.

This program will run for ten seconds (or however long is specified). Once ten seconds has passed, the runningA variable will be changed to false to signal the end of the program and threads will gracefully exit.

Real World Scenario One

The first real world scenario is a bank account. It is imperative to the functionality of a bank account that resources are accessed in a way that protects the integrity of the contents of the bank account. For example, if someone is withdrawing money and making a deposit within a similar timeframe, there must be a process where either the withdrawing or depositing locks the other process out to ensure that the bank account balance is accurate. If we tried to withdraw and deposit at the same time and both processes were allowed to occur at once, we would see inaccuracies in the total value of the account.

Real World Scenario Two

The second real world scenario is a small business owner creating tshirts. He has an online store with one tshirt listed. However, there are 5 customers trying to buy it at the same time. The problem arises because the business owner is not producing tshirts fast enough and we hit a

dead lock because there is not enough stock and neither the business owner or customers know what is happening. To fix this, we need to lock the stock numbers of the tshirt as soon as a customer puts it into his cart. The business owner and the other customers cannot access the stock levels or update the stock levels until the first consumer either buys the tshirt or releases it. This ensures security in stock levels for both the business owner and the customer. There are no race conditions or deadlocks leading to inaccurate stock numbers and confusion.

PROBLEM D – SLEEPING BARBER PROBLEM

The second problem I chose was the Sleeping Barber Problem.
My issues with this problem arose when I combined Problem A and Problem D into one program. Prior to this, when I ran Problem D on it's own program, there were no memory leaks and all ran smoothly. Unfortunately, occasionally in the dual program, I see 576 bytes of possibly lost memory. It is a positive that there is no definitely lost memory however, I was unable to locate the memory leak that sometimes occurs in the dual program.
Outside of Valgrind, the program runs perfectly however, valgrind affects the thread speed. Another limitation is that I was unable to use thread IDs to call via thread due to lack of knowledge and no content on this in the course. This didn't affect my implementation too much but with further experience it would be beneficial to use the thread ids to call the next customer to have their haircut.

Variable Explanation

pthread_mutex_t appointmentInProgress: mutex to ensure only one appointment is occurring.

pthread_mutex_t updateQueue: mutex so only one thread has access to update customer queue.

pthread_mutex_t updateSeatsAvailable: mutex so only one thread has access to update the number of seats available in the waiting room.

sem_t customer: semaphore to signal when customer is ready.

sem_t barber: semaphore to signal when barber is ready.

pthread_t customerCreator: thread to run the function arrivingCustomers for duration of program runtime.

bool runningD = true: this is our boolean value that allows the program to keep running. If runningD is true, the thread functions will run. If runningD is changed to false, the threads will exit their while loop.

int seatsAvailable : number of seats available in the waiting room.

int queue[NUMBER_SEATS]: a queue that holds the order of who is served by the barber.

int noCustomersVisited: holds the number of customers visited whether they were able to get their hair cut or not.

int maxRandomNumber: the maximum number allowed for random amount of sleep.

int sleepTimeDivisor : divisor to convert seconds to milliseconds

float postOutputSleep: how long to sleep threads for after output or action.

Algorithm

The program starts by creating one customer creator thread. This thread activates the arrivingCustomers function which creates threads continuously while running is true and sends those threads to the waiting room. This programs the idea of customers walking continuously through the door.
Following that, the barber threads are created.
The algorithm works as follows:
As arrivingCustomers creates customers they are sent to the waiting room. Once they enter the waiting room, if there are seats available, they take a seat and they are added to the back of the queue. The number of customers waiting is increased and the seats available is decreased. The waiting customer semaphore signals that there are customers ready and the customers then wait for a barber to become available. When a barber becomes available, the next customer in the queue is served, removed from the queue and the seats available increases again.
If there are no seats available, the customer is told to leave and is not served.
To protect from deadlocks, we use the two semaphores, customer, and barber. The barber thread waits for the customer semaphore to signals that there are customers in the waiting room before proceeding. In the waiting room, the customer waits for the barber to signal that they are ready to receive a customer. These semaphores avoid a deadlock by working in opposite fashion to each other in their functions and ensuring that there is never a case where there is busy waiting.
We avoid race conditions by using multiple mutexes. The updateQueue and updateSeatsAvailable ensure that only one thread at any given moment can be reading or writing data to the queue and the seats available.
We use the appointmentInProgress mutex to aim to create equal work for the barbers. As only one barber can access the appointmentInProgress, this inherently ensures that the work is spread more evenly across the barber threads.

Real World Problem One

A call centre. Picture an employee of a call centre sleeping at their desk, they are the barber. When a customer calls into the call centre they enter a portal that advises them whether there is space for their call or whether they are unable to be served right now. If there is space, the signal is sent to the call centre employee to wake up and serve the customer. There are a certain number of customers that can wait in the call centre FIFO queue. Every time the queue is empty, the employee has another nap until they are woken up again.

Real World Problem Two

A coffee shop. Picture a barista hungover after a night out. They are slumped against their coffee machine. When a customer arrives in store, they are advised of the wait time and whether there is space for them to be served. If there is the capability, they can wait in the queue (or be served straight away if the queue is empty). Once it has been established they can be served, the barista is woken up to make their coffee. The barista will sleep every time the queue is empty and wait to be woken up when the customer needs their coffee.