Programming Assignment 1 Report

Student ID: s3778713

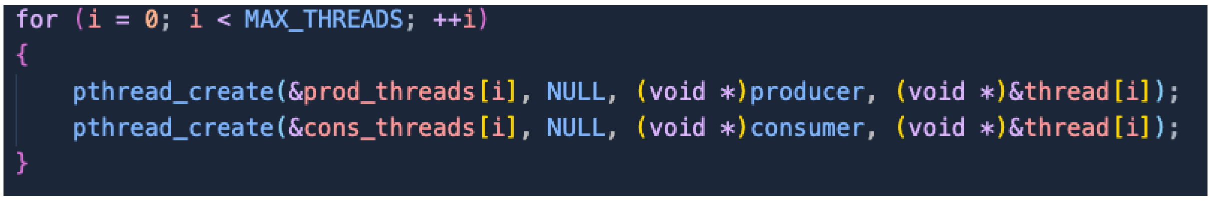
Name: Amna Alfarah Campos Alonto

Github Project URL: https://github.com/aalonto/OSP_A1

A. **Producer Consumer Problem**

In this problem, there are 5 producer threads and 5 consumer threads that are concurrently producing and consuming an item to and from the buckets. The solution I have implemented does not experience any deadlocks and have exited the program successfully.

Looking at the algorithm in-depth, the main function initialises a producer and consumer thread which have similar indices in one for loop. Although the producer is written before the consumer, this does not guarantee that it will run first. Thus, a boolean variable called *isFilled* that is initially set to false is used by both the consumer and producer functions to ensure the producer function gets to run first. This is set to true once the producer has added an item to the buckets in which the consumer is then able to run, also avoiding starvation due to alternating between producing and consuming.



```
for (i = 0; i < MAX_THREADS; ++i)
{
    pthread_create(&prod_threads[i], NULL, (void *)producer, (void *)&thread[i]);
    pthread_create(&cons_threads[i], NULL, (void *)consumer, (void *)&thread[i]);
}
```

This solution also ensures that there is always an item to consume for the consumer and a space in the buckets for the producer to keep producing an item. This is also protected by the integer *full* which stores the amount of filled buckets. Hence, the algorithm avoids busy waiting and a deadlock consequently.

The concurrency of threads did not lead to deadlocks as a mutex and a condition is used to lock critical sections of the methods. This ensures that the buckets are not modified by other threads as only one thread can do that at a time.

This scenario can be seen in production pipelines wherein the production is based on how high or low the supply is. If there is a surplus of goods in a given product, then the production can be postponed for a short time until the quota of sales is reached. This is important especially in the food and beverage industry where expiration dates of products can affect their longevity.

In this scenario, the producer threads are the factories and the consumer threads are the product sales. The mutex locks can be the person in charge to monitor the sales of the product and signals the factory whether to stop or resume the production.

Another real-life situation would be two cars going opposite directions sharing a one-lane street. They are not able to pass as this street can only accommodate one passing car at a time. One car has to wait for the other car to pass for it to continue its driving. In this situation, the two cars are the producer and consumer threads who are waiting for the other process to end and the street would be the single mutex lock they share. The mutex is unlocked once the other car has driven past the one-lane street.

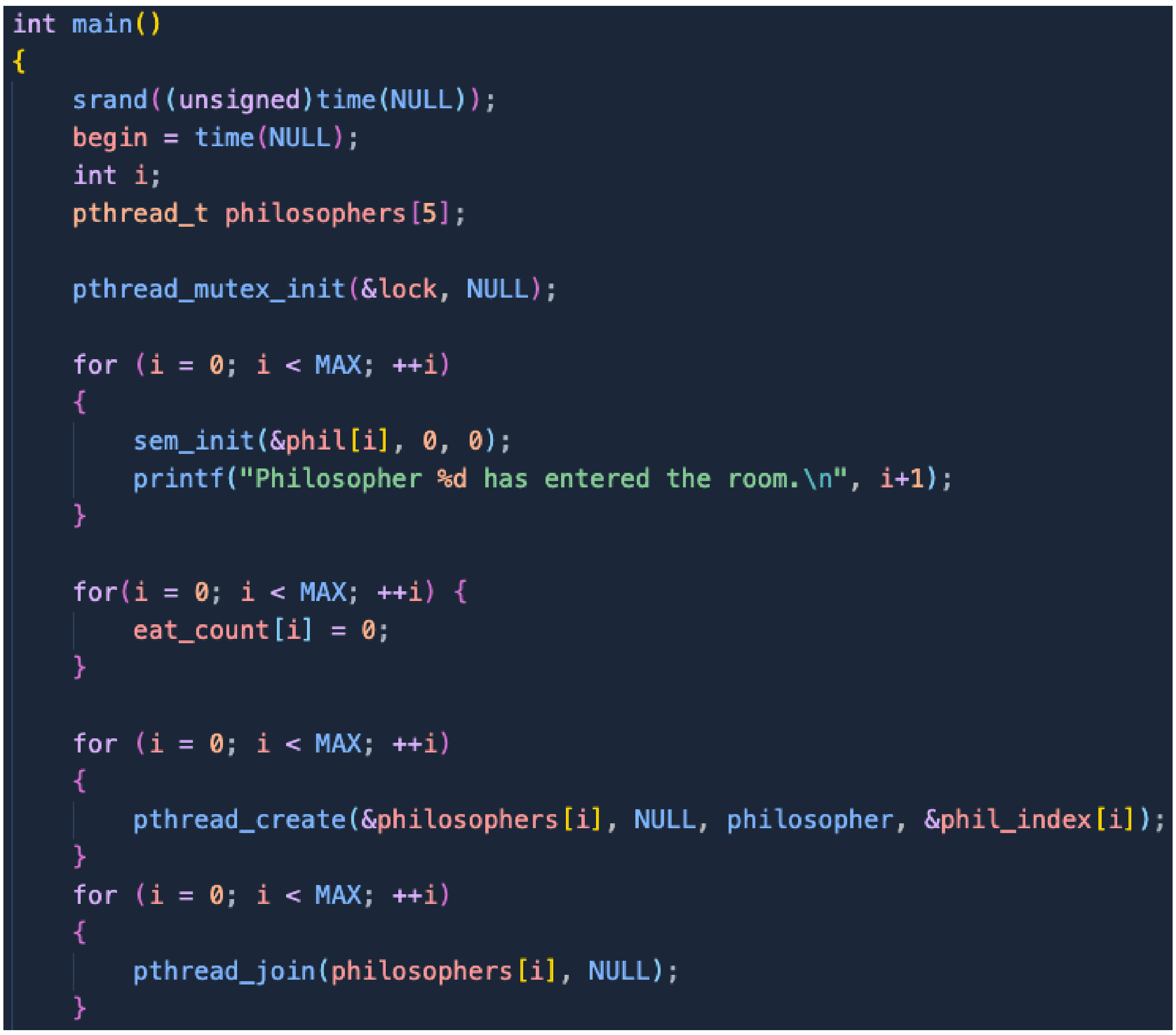
B. **Dining Philosophers Problem**

This problem is given with 5 philosophers circled around a table with 5 chopsticks and a bowl of rice. These philosophers would think and then eat when they're hungry and go back to thinking after they've eaten. This cycle repeats until the restaurant has "closed".

Looking at thus problem, there would be at least one person who is not able to eat if all philosophers eat at the same time. Deadlocks would occur if all philosophers each hold on to one chopstick while waiting for their second chopstick to be available.

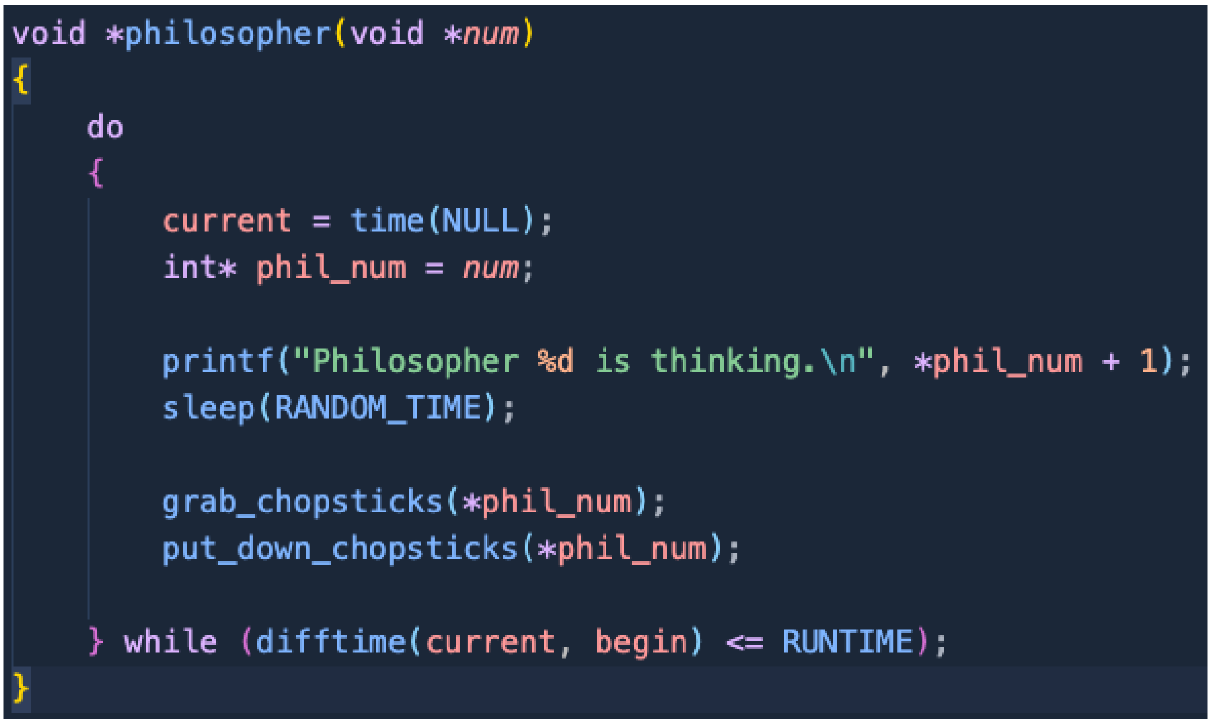
I have come up with a solution to this problem wherein if a philosopher is hungry, they would first check whether his neighbors, that is the philosophers to the left or right of him, is still eating. This is because it would mean that the chopstick that they would want to use is still in use by his neighbor/s. I will go to the algorithm in depth in the following paragraphs below.

In the main method below, a mutex lock for locking critical sections (initialised to 1) of threads later on and a thread of philosophers are initialised. The semaphore array *phil* with size 5 is also initialised to 0. Thi semaphore represents whether a philosopher is allowed to eat which will be seen later on in the process. The number of times each philosopher has eaten is also monitored through the *eat_count* integer array.



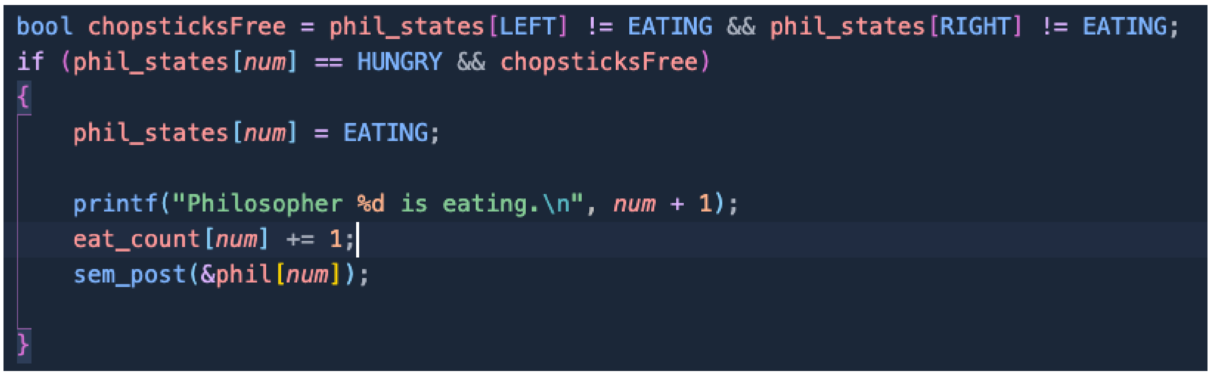
```
int main()
{
    srand((unsigned)time(NULL));
    begin = time(NULL);
    int i;
    pthread_t philosophers[5];

    pthread_mutex_init(&lock, NULL);

    for (i = 0; i < MAX; ++i)
    {
        sem_init(&phil[i], 0, 0);
        printf("Philosopher %d has entered the room.\n", i+1);
    }

    for(i = 0; i < MAX; ++i) {
        eat_count[i] = 0;
    }

    for (i = 0; i < MAX; ++i)
    {
        pthread_create(&philosophers[i], NULL, philosopher, &phil_index[i]);
    }
    for (i = 0; i < MAX; ++i)
    {
        pthread_join(philosophers[i], NULL);
    }
```

Once each thread start running, the philosopher method is ran. In this method, this is where the functions (action) of grabbing the chopsticks, eating, and then putting down the chopsticks is called. The philosophers repeat this until the run time of 10 seconds has been reached.



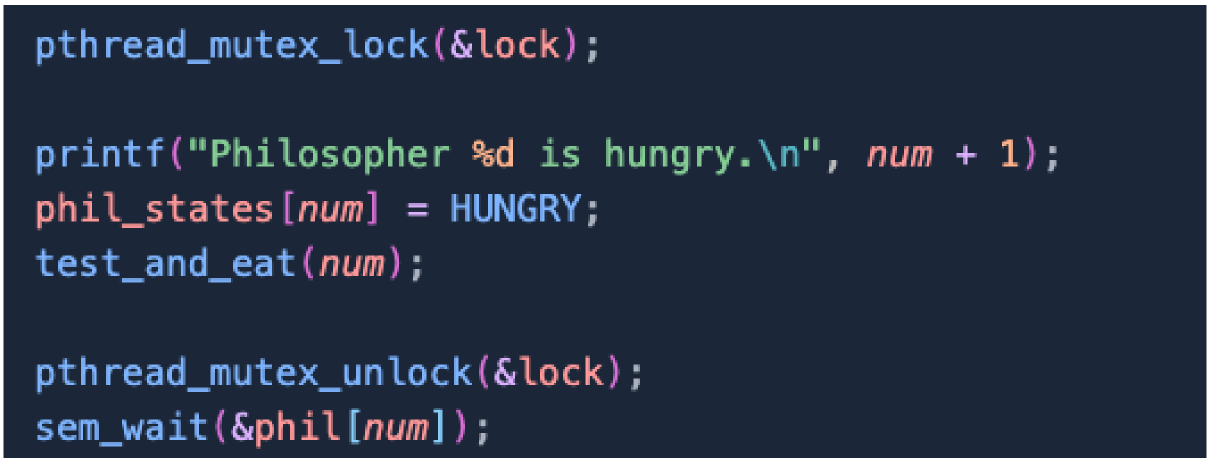
```
void *philosopher(void *num)
{
    do
    {
        current = time(NULL);
        int* phil_num = num;

        printf("Philosopher %d is thinking.\n", *phil_num + 1);
        sleep(RANDOM_TIME);

        grab_chopsticks(*phil_num);
        put_down_chopsticks(*phil_num);

    } while (difftime(current, begin) <= RUNTIME);
}
```

In the *grab_chopsticks* method, the *test_and_eat* function is called where the checking of neighbours eating is done. If the neighbours are not eating, then the chopsticks are free for the current philosopher to use. The state of the philosopher is then changed from “hungry” to “eating” and the semaphore is set to 1. Note that I did not use an actual variable to represent the chopsticks as no neighbours can eat the same time with the if condition I have used.



```
bool chopsticksFree = phil_states[LEFT] != EATING && phil_states[RIGHT] != EATING;
if (phil_states[num] == HUNGRY && chopsticksFree)
{
    phil_states[num] = EATING;

    printf("Philosopher %d is eating.\n", num + 1);
    eat_count[num] += 1;
    sem_post(&phil[num]);

}
```

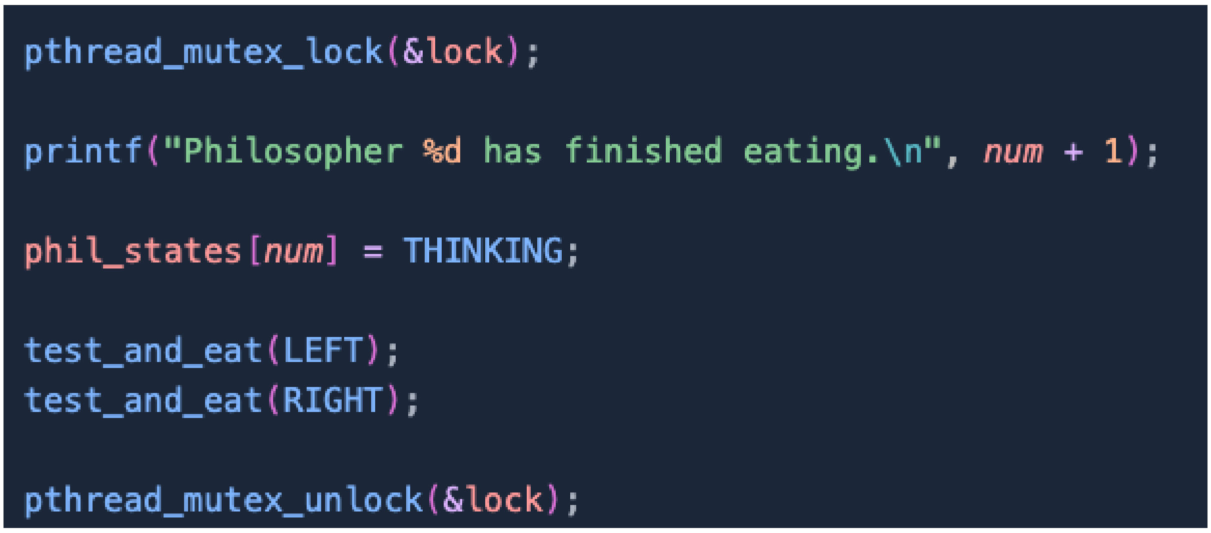
If the philosopher is not able to eat, once the process goes back to the *grab_chopsticks()* function, it will wait using the *sem_wait()* until such time



```c
pthread_mutex_lock(&lock);

printf("Philosopher %d is hungry.\n", num + 1);
phil_states[num] = HUNGRY;
test_and_eat(num);

pthread_mutex_unlock(&lock);
sem_wait(&phil[num]);
```

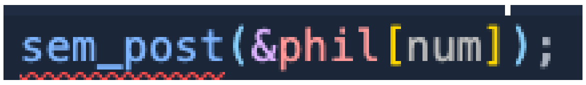
the philosopher is able to eat through the following process I will be explaining further.

This is done so when after a philosopher has finished eating, it will call *put_down_chopsticks()* and test if the neighbours are able to eat after him, calling the test_and_eat() for his neighbours.



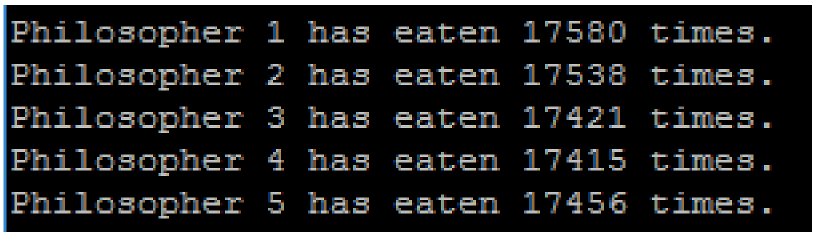
```c
pthread_mutex_lock(&lock);

printf("Philosopher %d has finished eating.\n", num + 1);

phil_states[num] = THINKING;

test_and_eat(LEFT);
test_and_eat(RIGHT);

pthread_mutex_unlock(&lock);
```

As you can see, inside this function, the semaphore for the given philosopher will be available which will alert the other philosophers to resume eating.



```
sem_post(&phil[num]);
```

In terms of fairness, I could say that the solution implemented is fair as the range between how many times each philosopher has eaten is not huge. I believe this is because of the process I have mentioned wherein a philosopher checks its neighbours whether they could eat or not. Below is the given output.



```
Philosopher 1 has eaten 17580 times.
Philosopher 2 has eaten 17538 times.
Philosopher 3 has eaten 17421 times.
Philosopher 4 has eaten 17415 times.
Philosopher 5 has eaten 17456 times.
```

In the real world, this problem is seen in a banking system in which two accounts are executing a transaction. In this example, the chopsticks would be the two accounts as you need to lock the accounts to ensure the correct amount is taken from one account to another. If locking the resources are not done, an account may run out of money even before it is credited which will cause the transaction to fail.

Another scenario would be a hostage taking incident. The hostage taker would not release his/her captive until they are given the money they want to receive. If in the instance the authorities are withholding the ransom money in hopes of trying to talk to the taker, the hostage taker will not let go of the captives, which the authorities are waiting to be released. Hence, the authorities must give the hostage taker the money before they achieve their goal. A deadlock occurs when both the parties are waiting, hence one must release the resource the other party needs before they can do their part.