**OSP-2150-PrgAsg1**

Student ID: s3850825

Name: You Chan Lee

Github project url: https://github.com/s3850825/OSP-PrgAsg1-s3850825

In this report, I will explain how the algorithm for the Producer-Consumer problem and the Dining Philosophers’ problem works and which real-world industrial or business scenarios each algorithm applies in.

**The Producer-Consumer Problem**

First, I defined NUM\_THREADS as 5 and NUM\_BUCKETS as 10 because they are given. I also defined MAX\_RANDOM\_NUMBER as 99 to generate random numbers between 1 and 99 and SLEEP\_TIME as 100000 to make threads sleep for 100000 microseconds. There are some global variables, bucketIn variable is declared for Producer to see which bucket Producer is in and initialised to 0 as it starts from the first bucket, whereas bucketOut variable is declared for Consumer to see which bucket Consumer is in and initialised to 0 as it starts from the first bucket. An integer NumOfItems is declared and initialised to 0 to check how many items in the bucket. A size 10 integer array bucket is declared to contain the produced random numbers. Four timeval variables are declared to check the start and end time. A double variable elapsedTime is declared to calculate the running time. Lastly, pthread\_cond\_t and pthread\_mutex\_t is declared.

After declared all the global variables, the algorithm starts from the main method. Firstly, A size 5 integer array threadNum is declared and initialised for Producer’s and Consumer’s numbering. An integer variable result is used to check the result of thread creation and t is used in for loop. And 5 producers and 5 consumers are declared by each array producers and consumers. Srand((unsinged) time(NULL)) is used to use of the computer’s internal clock to keep changing the seed for random number generator. Both of pthread\_mutex\_init() and pthread\_cond\_init() are used to initialize pthread\_mutex\_t and pthread\_cond\_t object. After that, using for loop 5 producer threads and 5 consumer threads are created by using pthread\_create(). Own numbering from array threadNum is passed to each producer and consumer thread when they are created. If an error occurs when threads are created, then the error will be printed.

The function producer is used to state what producers need to do and the function consumer is used to state what consumers need to do. Firstly, in producer function all the producers have infinite while loop to keep producing items, but while loop will be broken when the running time is greater than or equal to 10 seconds. Therefore, producers only work for 10 seconds. A producer produces one item which is a random number between 1 and 99. pthread\_mutex\_lock() is used to lock this thread so that no other thread can execute the same region until this thread is unlocked which is pthread\_mutex\_unlock(). This can avoid race condition because only one thread is working at a time. While inside of pthread\_mutex\_lock() a producer checks if buckets are full which means NumOfItems is 10, then this producer thread waits until a consumer thread consumes an item using pthread\_cond\_wait(). If buckets are not full, then this producer thread produces put the item in the bucket and increases NumOfItems by 1. Which producer produces which item and put the item in which bucket will be printed. Now, bucketIn variable is increased by 1 to jump to the next bucket. Once the production is done, then pthread\_mutex\_unlock() is used to unlock this thread so that other threads can execute and also pthread\_cond\_signal() is used to send a signal to another thread who is on waiting. This also can avoid deadlock because sending a signal to other threads prevents making all threads to wait. After send the signal, this thread sleeps 10000 microseconds which are same as 0.01 seconds. This can avoid starvation because while this tread is sleeping, other treads can start working, so this ends up dividing the work fairly.

Now, either a producer or a consumer starts working. Secondly, in consumer function all the consumers also have infinite while loop, so consumers also work for 10 seconds. A consumer consumes one item which is a producer produced in a bucket before. Likewise, pthread\_mutex\_lock() is used to lock this consumer thread to prevent those other threads execute the same region. While this consumer thread is locked, this consumer checks if buckets are empty which means NumOfItmes is 0, then this consumer thread wait until a producer thread produces an item using pthread\_cond\_wait(). If buckets are not empty, then this consumer thread consumes an item and make this bucket empty and decreases NumOfItems by 1. Which consumer consumes which item in which bucket will be printed. Now, bucketIn variable is increased by 1 to jump to the next bucket. Once the consumption is done, then pthread\_mutex\_unlock() is used to unlock this thread so that other threads can start working and pthread\_cond\_signal() is also used to send a signal to another thread who is on waiting. After send the signal, this thread sleeps 10000 microseconds which are same as 0.01 seconds. This can avoid starvation because while this tread is sleeping, other treads can start working, so this ends up dividing the work fairly. After production and consumption run for 10 seconds, pthread\_join() is used to wait all threads to be terminated. Once all the threads are terminated, then checks the total running time and print it. Lastly, pthread\_mutex\_destroy() is used to destroy the mutex object and pthread\_exit(NULL) is used to terminate calling thread.

We can see the producer and consumer problem in real-world industrial. For example, in post office if customers send parcels, then these parcels are kept in post office. While these parcels are waiting to be sent to the recipient, a delivery truck comes to post office and workers load parcels into the truck. Therefore, other workers need to unload these parcels from the full delivery truck to deliver them to the recipient and workers in post office need to wait until next empty delivery truck comes in to load parcels. Here we can clearly see that items are parcels, producers are workers who need to load parcels into an empty delivery truck, consumers are workers who need to unload parcels from a full delivery truck to deliver them to the recipient and buckets are delivery trucks. Sleeping time could be the time delivery trucks spend to get the post office. Pthread\_mutex\_lock() is used when a worker who start loading a parcel into a deliver truck in post office and pthread\_mutex\_unlock() is used when the worker finishes loading. Pthread\_cond\_wait() is used if a delivery truck is full for workers in post office which is for producers and empty for workers for delivering which is for consumers, so workers cannot load more parcels into the delivery truck and deliver more parcels to the recipient. Pthread\_cond\_signal() is used if new empty delivery truck comes in post office or full delivery truck come to workers for delivering. As the time delivery trucks arrive in post office could be varied, workers can notice that new empty deliver truck comes in post office and workers can notice that there are no more parcels to deliver from a delivery truck to the recipient, there is no race condition, deadlock, and starvation for both workers.

Another example is print spooling in Officeworks. As we all know that there is a huge and fast printer in Officeworks, so both staff and customers can ask the print task even though the printer is working. If staff got the print task while the printer is working, then the print task can be added to the printer from staff computer. Customers also can click the print button on their computer. Here we can clearly see that items are print tasks, producers are staff computers and customer computers, and consumer is the printer. Multiple items which are print tasks can be created by staff and customer computers and consumed by a printer. Pthread\_mutex\_lock() is used when a staff or customer clicks print button which is for producers and a printer is printing the task which is for consumers, and pthread\_mutex\_unlock() is used when the task is enqueued by the printer spooler and when the printer finishes printing the task. Pthread\_cond\_wait() is used if the queue for a printer is full which is for producers and empty which is for consumers, and pthread\_cond\_signal() is used if a staff or customer finishes asking the print task and the printer finishes printing the print task. As a result, there is no race condition, deadlock, and starvation for both computers and a printer.

**The Dining Philosophers’ Problem**

First, I defined NUM\_PHILOSOPHER as 5 because they are given. I also defined LEFT as -1 and RIGHT as 1 because it keeps checking neighbour index, and MAX\_DELAY as 500000 and MIN\_DELAY 100000 to generate random delay time between 100000 and 500000 microseconds. INC\_ONE\_INDEX is also defined as 1 because philosophers and forks numbering starts from 1 but actual index in each array starts from 0 so we need to add INC\_ONE\_INDEX when they need to be converted. There is an enum which has 3 states of philosophers, THINKING, HUNGRY and EATING. There are some global variables, a size 5 integer array NumOfEaten is declared to contain how many meals each philosopher had. Three timeval variables are declared to check the start and end time. A double variable elapsedTime is declared to calculate the running time. Lastly, pthread\_cond\_t and pthread\_mutex\_t is declared. pthread\_mutex\_t is forks array to lock and unlock whenever each philosopher uses the fork. Pthread\_cond\_t is wait\_here array to make thread to wait or to stop waiting.

After declared all the global variables, the algorithm starts from the main method. Firstly, A size 5 integer array threadNum is declared and initialised for philosopher’s numbering. An integer variable result is used to check the result of thread creation and t variable for for loop. And 5 philosophers are declared by array philosophers. Srand((unsinged) time(NULL)) is used to use of the computer’s internal clock to keep changing the seed for random number generator. For loop is used to initialize enum array state to THINKING, integer array NumOfEaten to 0, pthread\_mutex\_t object with pthread\_mutex\_init() and pthread\_cond\_t object with pthread\_cond\_init(). gettimeofday() is used to check the start time. After declared philosophers, using for loop 5 philosopher threads are created by using pthread\_create(). Own numbering from array threadNum is passed to each when they are created. If an error occurs when threads are created, then the error will be printed.

The function philosopher is used to state what philosophers need to do. Firstly, integer variable item is declared and in philosopher function all the philosophers have infinite while loop to keep trying to think and pick up left and right forks, but while loop will be broken when the running time is greater than or equal to 10 seconds. Therefore, philosophers only work for 10 seconds. A philosopher needs to recognize index of left fork which is same as philosopher’s index and right fork which is same as philosopher’s index plus one, but we only need index between 0 and 4, so modulo operator is used. A philosopher also needs to recognize index of left philosopher which is one index smaller and right philosopher which is one index bigger. Modulo operator is also used here with the same reason. It goes into infinite while loop for a repeat of thinking, picking up forks, eating and returning forks. Firstly, in while loop it checks the running time with method check\_running\_time(), if the running time is greater than or equal to 10 seconds, then it returns true so ends up breaking the while loop and the philosopher finishes working. Now, he is time to think, so sleeps for random time between 100000 and 500000 microseconds and display that he is thinking. After thinking, his state is changed to HUNGRY, so he tries to pick up two forks first. Here, pthread\_mutex\_lock() for left fork is used to pick up left fork and display that he picked up left fork. If another philosopher is using the fork, then he needs to wait until another philosopher finishes eating. Once he gets left fork, then phtread\_mutex\_lock() for right fork is used to pick up right fork and display that he picked up right fork. This can avoid race condition because a fork is used by only one thread at a time. Once he picked up both forks, he needs to check neighbours state whether they are eating or not. If both neighbours don’t have EATING state and he has HUNGRY state, then his state is changed to EATING to prevent that both neighbours start eating. If both or either of two neighbours have EATING state, then he needs to wait using pthread\_cond\_wait() with two forks. Therefore, other philosophers cannot pick up these two forks. Once he gets a signal from others, then increase NumOfEaten by 1, so that how many meals each philosopher had can be checked at the end and he eats a meal for random time between 100000 and 500000 microseconds and display that he is eating a meal and how many meals he had as well. Once he finishes eating, then check left and right philosopher with method check\_neighbours() so that he can let them know he has done and after that he returns two forks using pthread\_mutex\_unlock() with two forks. Now, his state is changed to THINKING and keep going on the same process and also display that he returned left and right forks. Therefore, this can avoid deadlock because sending a signal to other philosophers prevents making all philosophers to wait. Sleeping random time after eating can avoid starvation because if one philosopher had a meal, then it needs to think for random time after eating, so others can pick up forks while he is sleeping. As a result, this ends up giving a fair chance to all philosophers. After the dining is running for 10 seconds, pthread\_join() is used to wait all threads to be terminated. Once all the threads are terminated, then checks the total running time and print it. And print the result how many meals each philosopher had. Lastly, pthread\_mutex\_destroy() is used to destroy all 5 forks threads and pthread\_exit(NULL) is used to terminate calling thread.

We can see the dining philosophers’ problemin real-world industrial. For example, in car rental service if a customer rent a car for 2 days, then no other customers can rent the car during the period, so they must wait. No two customers can rent the same car on the same day. Once a customer gets a car, then only the customer can use the car for the period. No one can force the customer to return the car earlier. Here we can clearly see that philosophers are customers and forks are cars in car rental service store. Thinking time could be the time customers need to spend to get the car rental service store. Eating time could be the period customers can use a car. Once a customer gets a car, then it is same as using pthread\_mutex\_lock(), so no other customers can access to this car. Also pthread\_cond\_wait() is used for other customers who want to rent the same car so that that they can know this car is now inaccessible. Once the customer returns the car, then it is same as using pthread\_mutex\_unlock(), so another customer can use this car. Also pthread\_cond\_signal() is used for other customers who are waiting to rent the same car so that they can know this car is now accessible. Therefore, many customers can share the limited resources which are cars without race condition, deadlock, and starvation like our algorithm above.

Another example is hotel accommodation system. If a customer book a room for 5 days, then no other customers can book the same room during the period. Similar with car rental service, no two customers can book the same room on the same day. Once customer gets a room, then only the customer can use the room for the period. No on can force the customer to return the car earlier. Here we can clearly see that philosophers are customers and forks are rooms in hotel. Thinking time could be the time customers need to spend to get the hotel. Eating time could be the period customers can use a room. Once a customer gets a room, then pthread\_mutex\_lock() is used to prevent other customers are using the same room. Also pthread\_cond\_wait() is used for other customers who want to book the same room so that they can notice that this room is not usable. Pthread\_mutex\_unlock() is used when the customer finishes using the room, so now another customer can use this room. Also pthread\_cond\_signal() is used for other customers who are waiting this room to be usable so that they can know this room is now usable. Therefore, many customers can share the limited rooms in hotel without race condition, deadelock, and starvation like our algorithm above.