# 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**

Firstly, I defined NUM_THREADS as 5 and NUM_BUCKETS as 10 because they are given. I also defined EMPTY_BUCKETS as 0 to check if buckets have no items in, 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. MAX_RUN_TIME is defined as 10.0 to check the running time and CONVERT_USEC_TO_SEC is defined as 0.000001 to convert microsecond to second. There are some global variables, bucketIn variable is declared for producers to see which bucket a producer is in and initialised to 0 as it starts from the first bucket, whereas bucketOut variable is declared for consumers to see which bucket a consumer is in and initialised to 0 as it starts from the first bucket. An integer Num_Items is declared and initialised to 0 to check how many items in the bucket. A size 10 integer array buckets is declared to contain items which are produced random numbers. Four timeval variables are declared to check the start, end, and run time. A double variable elapsed_time 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 declared to check the result of thread creation and t is declared for parameter in for loops. 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 producer function is used to state what producers need to do and the consumer function is used to state what consumers need to do. Firstly, integer variable item is declared to take a random number. All the producers have an infinite while loop to keep producing an item, but the while loop will be broken when the running time is greater than or equal to 10 seconds. The function gettimeofday() is used to check the current time and elapsed_time is calculated by subtracting the start time from current time. Therefore, producers only can 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. This can avoid race condition because only one thread can work at a time. While inside of pthread_mutex_lock() a producer checks if buckets are full which means Num_Items 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 Num_Items 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 next. This can avoid deadlock because sending a signal to another thread prevents making all threads to wait. After send the signal, this thread sleeps for 100000 microseconds which are same as 0.1 seconds. This can avoid starvation because while this tread is sleeping, other threads can start working, so this ends up dividing the work fairly. For instance, if producer1 is sleeping for 0.1 second, then producer2 can start producing. After that, while producer1 and producer2 are sleeping, producer4 can start producing. Now, producer1, producer2 and producer4 are sleeping, so either producer3 or producer5 can produce an item. As a result, all producers will get a chance to produce an item.

In consumer function all the consumers also have an infinite while loop, so consumers also work for 10 seconds. A consumer consumes one item which is a producer produced in a bucket before. Like producer function, the function gettimeofday() and elapsed_time is used to check the running time and pthread_mutex_lock() is used to lock this consumer thread to prevent that other threads can execute the same region. While this consumer thread is locked, this consumer checks if buckets are empty which means Num_Itmes is 0, then this consumer thread waits 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 Num_Items 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 next. After send the signal, this thread sleeps 100000 microseconds which are same as 0.1 seconds. This can avoid starvation because while this tread is sleeping, other treads can start working, so this ends up dividing the work fairly like the producer’s example. 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 who are outside 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 other 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 a break time for workers who are inside and outside. In terms of a producer, pthread_mutex_lock() is used when a worker who start loading a parcel into a delivery 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, so workers in post office have to wait for another empty delivery truck. Pthread_cond_signal() is used if a new empty delivery truck comes in post office. In terms of a consumer, pthread_mutex_lock() is used when a worker who start delivering a parcel to the recipient and pthread_mutex_unlock() is used when the worker finishes delivering. Pthread_cond_wait() is used if a delivery truck is empty, so workers who are outside of post office have to wait for another full delivery truck. Pthread_cond_signal() is used if a new full delivery truck comes to the worker so that the worker can start delivering. As pthread_mutex() and pthread_cond() are used like our algorithm, there is no race condition and deadlock. As a break time for workers who are inside and outside could be varied, all workers who are inside can start loading parcels into a new empty delivery truck while others are taking a break. All workers who are outside usually work alone with own delivery truck but if more than two workers work together, then they can start unloading parcels from a delivery truck while others are taking a break, there is no 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 a print task while the printer is working, then the print task can be added to the printer from the staff computer. Customers also can click the print button on their computer while the printer is working. If there is nothing to print, then the printer needs to wait until someone asks the print task. Here we can clearly see that items are print tasks, producers are staff computers and customer computers, and consumer is a printer. Multiple items which are print tasks can be created by staffs and customers' computer and consumed by a printer. Sleeping time could be the time a staff or a customer are ready for asking the print task. In terms of a producer, pthread_mutex_lock() is used when a staff or customer clicks print button and pthread_mutex_unlock() is used when the task is enqueued by the printer spooler. Pthread_cond_wait() is used if the queue for a printer is full, so no one can add the print task anymore and pthread_cond_signal() is used if the printer finishes one print task, so another task can be enqueued. In terms of a consumer, pthread_mutex_lock() is used when a printer is printing one task and pthread_mutex_unlock() is used when the print finishes printing the task. Pthread_cond_wait() is used if the printer finishes all the tasks, so there is no more task to print and pthread_cond_signal() is used if one task comes into the queue, so the printer can start printing the task. As pthread_mutex() and pthread_cond() are used like our algorithm, there is no race condition and deadlock. As the time a staff or a customer are ready for asking the print task could be varied, all of them can ask print tasks while others are still being ready and the printer can keep printing the task as it is alone, there is no starvation for both staffs, customers, and the 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's 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. MAX_RUN_TIME is defined as 10.0 to check the running time and CONVERT_USEC_TO_SEC is defined as 0.000001 to convert microsecond to second. There is an Enum array state which contains 3 kinds of states of philosophers, THINKING, HUNGRY and EATING. There are some global variables, a size 5 integer array Num_Meal is declared to contain how many meals each philosopher had. Three timeval variables are declared to check the start, end, and run time. A double variable elapsed_time is declared to calculate the running time. Lastly, pthread_cond_t and pthread_mutex_t are 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 declared to check the result of thread creation and t is declared for parameter in for loops. 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. A for loop is used to initialize Enum array state to THINKING, integer array Num_Meal 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 thread when they are created. If an error occurs when threads are created, then the error will be printed.

The philosopher function is used to state what philosophers need to do. Firstly, philosopher_number is declared to take the passing value that numbering for threads. A philosopher needs to recognize index of left fork which is same as philosopher's index and index of right fork which is same as philosopher's index plus one, but we only need index between 0 and 4, so a 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. A modulo operator is also used here with the same reason. In philosopher function all the philosophers have an infinite while loop to keep trying to think and pick up left and right forks, but the while loop will be broken when the running time is greater than or equal to 10 seconds. Therefore, philosophers only work for 10 seconds. In a while loop a philosopher reiterates thinking, picking up forks, eating and returning forks. Firstly, in a while loop it checks the running time with function 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 he sleeps for random time between 100000 and 500000 microseconds and display that he is thinking. After thinking for a random time, his state is changed to HUNGRY, so he tries to pick up two forks with function pickup_forks(). 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 the 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 philosopher at a time. Once he picked up both forks, he needs to check neighbours' state whether they are eating or not with function check_neighbours(). If both neighbours don't have EATING state and he has HUNGRY state, then his state is changed to EATING and send a signal to him with pthread_cond_signal(), so that he stops waiting. This can avoid deadlock because sending a signal to a philosopher prevents making all philosophers to wait. If both or either of two neighbours have EATING state, then he needs to wait using pthread_cond_wait() with two forks until both left and right neighbour finish eating. Once he gets a signal from neighbours, then he start eating with function eat(). The value of array Num_Meal of his index is increased by 1, so that how many meals each philosopher had can be checked at the end and he eats a meal for a 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 checks left and right philosopher with function check_neighbours() so that he can let them know you can start eating. Now, he needs to return the forks with function return_forks(). His state is changed to THINKING and keeps going on the same process and displays that he returned left and right forks. 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 runs 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 prints it and prints 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' problem in real-world industrial. For example, in a car rental shop 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, other customers must wait for the car. Here we can clearly see that philosophers are customers and forks are cars in a car rental shop. Thinking time could be the time customers need to spend to get the car rental shop. Thinking state could be the state before getting to the car rental shop. Hungry state could be the state that the customer has decided to rent which car for how long. Eating time could be the period customers can use a car. Eating state could be the state that the customer is using the car. When a customer tries to get a car, pthread_mutex_lock() is used, so no other customers can access to the same car. Also pthread_cond_wait() is used for the customer so that the customer can know that this car is now inaccessible, so must wait. When the customer returns the car, pthread_mutex_unlock() is used, so another customer can use this car. Also pthread_cond_signal() is used for the customer so that the customer can know that now this car is accessible. As pthread_mutex() and pthread_cond() are used like our algorithm, there is no race condition and deadlock. As the time customers need to spend to get the car rental shop could be varied, many customers can share the limited resources which are cars in the car rental shop without 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 the 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, other customers must wait for the room. 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. Thinking state could be the state before getting to the hotel. Hungry state could be the state that the customer has decided to book which room for how long. Eating time could be the period customers can use a room. Eating state could be the state that the customer is using the 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 the customer so that the customer can notice that this room is not usable, so must wait. 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 the customer so that the customer can know this room is now usable. As pthread_mutex() and pthread_cond() are used like our algorithm, there is no race condition and deadlock. As the time customers need to spend to get the hotel could be varied, many customers can share the limited resources which are rooms in the hotel without starvation like our algorithm above.