ITP 30002-02 Operating System, Spring 2020

Homework 3

Jeon, Yeo Hun, 21500630, 21500630@handong.edu

1. Introduction

In this assignment, I am asked to implement TSP(Traveling Salesman Problem) solver by creating multiple threads to solve subtask of the problem. There are three different types of thread who work for different purpose; the main thread, producer thread and consumer thread.

The main thread is a parent of all other threads. when user gives tsp data and number of threads, this main thread creates a single producer thread and given number of consumer threads. After creating threads, the main thread is waiting for user's input until the program find the final solution to provide the information of each consumer threads and progress of the solution. In addition, when user decrease the number of consumers, some consumer should be terminated. At this moment, the subtask of canceled consumer should be assigned to other consumer, so the program does not skip any possible solutions. I implement queue and thread cancel handler to save the incomplete subtask and assign it to another possible consumer.

The producer thread works for creating a 'prefix', which will be the identifier of subtask and make consumer to check 11! number of possible routes at the same time. To make the prefix, the producer makes permutation with size of 'the total number of cities – 11', in order to make the length of subtask to 11 cities. After creating the prefix, the producer will put the prefix into bounded buffer. If the buffer is full, the producer waits for consumer to take data from the buffer.

The consumer threads take the prefix from the bounded buffer, if there is no possible prefix in the buffer, the threads will wait for it. Since the bounded buffer is shared resource among producer thread and consumer threads, it is necessary to control the possible race condition. Therefore, the bounded buffer needs to have semaphore and mutex lock to guarantee one thread access to the buffer at one time. After taking prefix, the consumer thread starts to make permutation with the prefix. When a consumer finds the optimal solution, it updates the global solution with its solution.

2. Approach

After the main thread creates producer thread and given number of consumer thread, they will start their own action with assigned function by the main thread. When the producer thread starts, it immediately starts to make prefix with permutation. The size will be the total number of cities – 11, so the consumer thread can work for 11 other cities. When the prefix is created, the producer tries to access the bounded buffer to put the prefix into the queue. At this moment, the producer executes semaphore wait(). Since the semaphore for empty is initialized with buffer capacity, the semaphore will be decreased to -1 when there the buffer is full and then the producer thread is blocked. Otherwise it gets into the critical section by acquiring lock by mutex to prevent consumer thread gets into the critical section at the same time. In the critical section, the producer thread puts prefix into the buffer. When the producer finishes its job, it unlocks the mutex and execute signal to increase value of filled semaphore. This semaphore is initialized with integer 0.

The behavior of consumer threads is to take prefix in the buffer and start to make permutations. To get the prefix from buffer, the consumer thread executes wait() for filled semaphore. If the buffer does not have any data, the consumer thread is blocked. Otherwise, it acquires mutex lock and get data from the buffer. When the thread gets data, it needs to read the prefix and put it into its own path. Therefore, before starting the permutation, the consumer thread updates list of visited cities with the prefix. By this, the consumer has 11 unvisited cities and it can start to check 11! permutations.

While solving the problem, user can interact with the program to get solving status and give another option to change the solving condition. The first option is 'threads', which shows each status of all running threads. Since the program creates multiple consumer threads, the thread id is saved in array. Therefore, each thread will have their index number, and this number can be used as their identifier. To keep each number of checked routes of the threads, use another array to save the number of the checked route with corresponding index of the thread id.

Second option is 'stat'. This option gives the best solution up to the point. The minimum distance and the path are saved in global variable. This means that the variables are shared for all threads. Each thread is updating the variables when they found the optimal solution. Therefore, for this option, the program just need to print out the distance and path saved in the global variables.

The last option is num N. With this option, the user can change the number of threads to solve the problem.

The increasing number can be done by simply creating extra threads up to the given number. However, decreasing the number of threads is complicated. First, the program uses pthread_cancel() to terminate the thread. To terminate the thread with this function, the consumer thread function needs to have pthread_setcanelstate() with PTHREAD_CANCEL_ENABLE parameter. In addition, the function also needs to have pthread_setcanceltype with PTHREAD_CANCEL_ASYNCHRONOUS parameter to terminate the thread immediately when the user calls the cancel function. Since the thread terminates immediately, the subtask of the terminated thread will be skipped without checking all the possible route. Therefore, it is necessary to keep the terminated subtasks and assign them to other running threads when they are possible.

For the functionality, the program has specific function that calls before canceling. This function can be set with pthread_cleanup_push() and pthread_cleanup_pop() to release it. In the function, the terminating thread saves the subtask's prefix into queue that collects all the terminated prefix. The queue is implemented with linked list. At this point, the program has two different resource to get prefix data. one is the bounded buffer and one is queue of terminated prefix. Therefore, when consumer function tries to take prefix, it takes prefix from the queue of prefix first to finish up the terminated subtask. If there is no data in the queue, the consumer thread can take data from the bounded buffer.

3. Evaluation

To evaluate the requirements of this program, I need to answer to several questions.

1. Does the program successfully create producer and consumer threads?

I run the program with sample data of 17 cities and 5 starting consumer thread. By printing out 'threads' option, I could observe the 5 different consumer threads are working and the producer thread keeps making the prefix.

2. Does the program find acceptable solution?

In the same condition as above, I could observe that the minimum distance and its path by giving 'stat' option and they keep updating as time goes by.

3. Does the program successfully change the number of threads? Does the program keep the terminated subtask?

To check this requirement, I put debugging code into the program that prints out the content of the data queued and dequeued to and from both the bounded buffer and the prefix queue. And I increase the number of threads to 8, and decrease the number to 2, and then increase the number back to 5. As the result, I could check 3 new threads are created when I give number 8 to the program. When I decreased the number to 2, I could observe that the 6 prefixes are queued into the prefix queue. Finally, when I change the number to 5 again, I could observe that the consumer thread dequeue 3 prefixes from the queue and start to make permutations.

4. Discussion

Controlling the threads from their race condition needs very careful and elaborate work. The most struggle moment for me to implement this program is that It is very hard to find which thread makes what problem. For my case, I tried to use binary semaphore to update the solution from each thread. But the thread who holds the lock got problem of reading data and it unexpectedly got into infinite loop. Therefore, all other threads are stopped finding solution and wait for signal from the never-ending thread. I fixed it by changing loop condition, but It took a lot of time to figure out the problem.

Moreover, for the terminating prefix, it could be much better program if the producer can assign exactly same state of terminated thread to the new consumer thread. I tried this by put all the visited city, length, prefix, and path data into structure and save it in queue, but I faced the problem to restarting exactly same point of the terminated subtask.

5. Conclusion

In this assignment I implemented a TSP solver by using multiple threads. There are three different types of thread, one for providing UI, one for making prefix and assigning subtasks to consumer threads, and one for getting the subtasks from producer to find the solution. Since the prefix and the optimal solution data are needed to be shared among the threads, I used bounded buffer with semaphore, so that the producer thread puts prefix in, and the consumers takes the prefix out. The semaphore in the buffer works as the mutual exclusive lock and makes only one consumer thread takes the data at the same time. The most important part of this program besides finding a solution is to interact with user. User can give three different options to check the thread status, optimal solution and the user also can change the number of running consumer thread. The considerable part is that when user decrease the number of threads, the thread terminates without checking all the possible routes within its subtask. Therefore, the program needs to keep the prefix of subtask into specific queue and assign the prefix to consumer when they are ready to start the new subtask. The saving action before terminating can be done by assigning function called when the thread is canceled.