

CS347: Lab3

Thread Primitives using pthread library

Parth Shettiwar 170070021

Contents

1 Problem Statement	1
2 Thread Ordering	2
2.1 Problem Formulation	2
2.2 Algorithm	2
2.3 Result	2
3 Barrier synchronization	3
3.1 Problem Formulation	3
3.2 Algorithm	3
3.3 Result	3
4 Priority synchronization	4
4.1 Problem Formulation	4
4.2 Algorithm	4
4.3 Result	5
5 Subset Number of Threads Execution	5
5.1 Problem Formulation	5
5.2 Algorithm	6
5.3 Result	6
6 References	7

1 Problem Statement

I have done the Q6 part in this lab. In this part, we had to use the pthread library to implement a set of primitives for managing threads. The following primitives have been implemented:

- Thread Ordering
- Barrier synchronization
- Priority synchronization
- Subset Number of Threads Execution

The following is the implementation and result for each of the primitives:

2 Thread Ordering

2.1 Problem Formulation

Executed in file: `Ordering.c`

This part was formulated as follows:

Given the number of threads and their order of synchronisation, execute the threads in that particular order.

2.2 Algorithm

The following algorithm was used to execute this primitive:

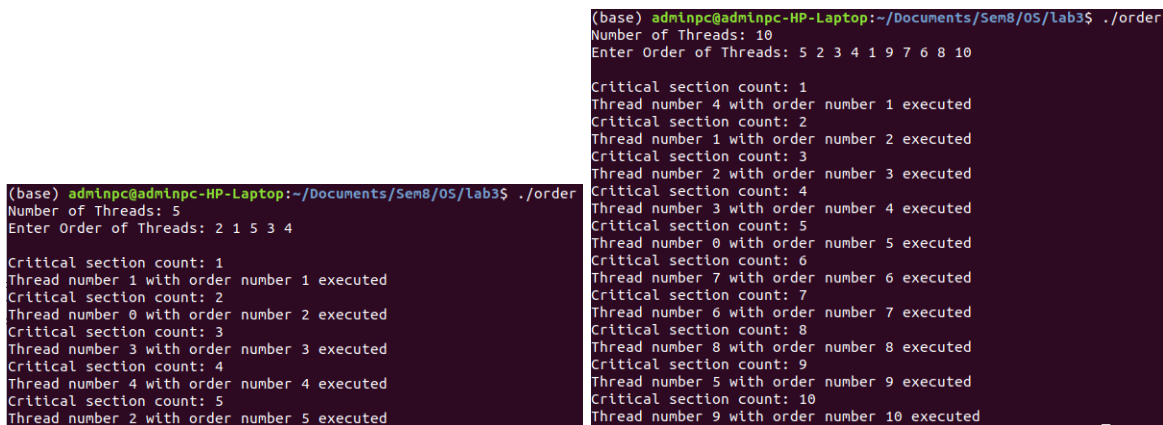
- Take the input of number of threads and their particular order from the user. If N threads are there, order should be any permutation of numbers [1,2,3....N].
- Save the mapping of order number to thread number in an array.
- Run a for loop to create and join threads in the order dictated by above array. Used functions: `pthread_create` and `pthread_join`.
- Every time a critical section is called by the create thread function, which has a shared variable count and it outputs the value of this shared variable after every update by a thread in the ordering passed by user. The critical section also outputs the thread number completed and what was its order number. The critical section has been covered by a mutex lock. Used functions: `pthread_mutex_lock` and `pthread_mutex_unlock`.

2.3 Result

Use the command: `gcc -pthread -o order Ordering.c`

Run file using this command: `./order`

I tested this primitive on various input values, i.e. Number of threads and their order as input. The following is the output for various such parameters passed:



```
(base) adminpc@adminpc-HP-Laptop:~/Documents/Sem8/OS/lab3$ ./order
Number of Threads: 5
Enter Order of Threads: 2 1 5 3 4

Critical section count: 1
Thread number 1 with order number 1 executed
Critical section count: 2
Thread number 0 with order number 2 executed
Critical section count: 3
Thread number 3 with order number 3 executed
Critical section count: 4
Thread number 4 with order number 4 executed
Critical section count: 5
Thread number 2 with order number 5 executed

(base) adminpc@adminpc-HP-Laptop:~/Documents/Sem8/OS/lab3$ ./order
Number of Threads: 10
Enter Order of Threads: 5 2 3 4 1 9 7 6 8 10

Critical section count: 1
Thread number 4 with order number 1 executed
Critical section count: 2
Thread number 1 with order number 2 executed
Critical section count: 3
Thread number 2 with order number 3 executed
Critical section count: 4
Thread number 3 with order number 4 executed
Critical section count: 5
Thread number 0 with order number 5 executed
Critical section count: 6
Thread number 7 with order number 6 executed
Critical section count: 7
Thread number 6 with order number 7 executed
Critical section count: 8
Thread number 8 with order number 8 executed
Critical section count: 9
Thread number 5 with order number 9 executed
Critical section count: 10
Thread number 9 with order number 10 executed
```

Figure 1: Thread Ordering primitive result. Number of threads: Left = 5, Right = 10. Order of threads: Left = 2 1 5 3 4, Right = 5 2 3 4 1 9 7 6 8 10

As it can be seen, in left figure, thread 1 with priority 1 was executed first as was the input. (Note: Thread numbering starts from 1). Similarly, Thread 0 with priority 2 was executed next and so on. Similar correct execution of order is seen in right figure where 10 threads were run. Shared variable count is also updated correctly, ensuring that all threads are running one after other and hence count = number of threads after full execution.

3 Barrier synchronization

3.1 Problem Formulation

Executed in file: `Barrier.c`

This part was formulated as follows:

Given the number of threads, implement a barrier such that after performing a task, wait for all threads to complete that task and then perform the remaining task

3.2 Algorithm

The following algorithm was used to execute this primitive:

- Take the input of number of threads from the user. N can be any number greater than 1, where N is the number of threads.
- Same as before, create that many number of threads using function : `pthread_create` and call the `critical_scetion` function Unlike before, dont join the threads now.
- Write a Task 1 inside the `critical_scetion` function, all threads will complete this task irrespective of other threads. In my case, Task 1 is a simple for loop, which assigns some constant value to a variable again and again.
- Barrier implement: Maintain a global variable `left` which keeps count of number of threads yet to complete task 1. As long as left is not equal to 0, put the threads to wait on a conditional variable `cond`. When last thread comes left will become equal to 0, now broadcast all threads to wakeup on same conditional variable. All this is done covered by a mutex lock, so shared variable left is not updated by more than 1 thread at a time.
Functions used: `pthread_cond_wait`, `pthread_cond_broadcast`, `pthread_mutex_lock` and `pthread_mutex_unlock`.
- Threads move on to complete task 2, again same as before, it is a for loop where a variable is assigned some constant value.
- Call the `pthread_join` to wait for all threads to come back in main function.

3.3 Result

Use the command: `gcc -pthread -o barrier Barrier.c`

Run file using this command: `./barrier`

I tested this primitive on various input values, i.e. Number of threads. Each time I have printed, how many threads are remaining and which thread is sleeping. After all threads come, I print which all threads have woken up by broadcast by last thread. Following is the result.

```

(base) adminpc@adminpc-HP-Laptop:~/Documents/Sem8/OS/lab3$ ./barrier
Number of Threads:10
10 Threads remaining to come
Thread 0 sleeping after Task 1 completion
9 Threads remaining to come
Thread 1 sleeping after Task 1 completion
8 Threads remaining to come
Thread 3 sleeping after Task 1 completion
7 Threads remaining to come
Thread 4 sleeping after Task 1 completion
6 Threads remaining to come
Thread 5 sleeping after Task 1 completion
5 Threads remaining to come
Thread 2 sleeping after Task 1 completion
4 Threads remaining to come
Thread 7 sleeping after Task 1 completion
3 Threads remaining to come
Thread 8 sleeping after Task 1 completion
2 Threads remaining to come
Thread 6 sleeping after Task 1 completion
1 Threads remaining to come
Last thread has arrived
Thread 9 wakes up and goes to Task2
Thread 3 wakes up and goes to Task2
Thread 4 wakes up and goes to Task2
Thread 8 wakes up and goes to Task2
Thread 0 wakes up and goes to Task2
Thread 1 wakes up and goes to Task2
Thread 5 wakes up and goes to Task2
Thread 6 wakes up and goes to Task2
Thread 7 wakes up and goes to Task2
Thread 2 wakes up and goes to Task2

(base) adminpc@adminpc-HP-Laptop:~/Documents/Sem8/OS/lab3$ ./barrier
Number of Threads:5
5 Threads remaining to come
Thread 0 sleeping after Task 1 completion
4 Threads remaining to come
Thread 1 sleeping after Task 1 completion
3 Threads remaining to come
Thread 2 sleeping after Task 1 completion
2 Threads remaining to come
Thread 3 sleeping after Task 1 completion
1 Threads remaining to come
Last thread has arrived
Thread 4 wakes up and goes to Task2
Thread 1 wakes up and goes to Task2
Thread 0 wakes up and goes to Task2
Thread 3 wakes up and goes to Task2
Thread 2 wakes up and goes to Task2

```

Figure 2: Barrier synchronization primitive result. Number of threads: Left = 10, Right = 5

As it can be seen, in left figure, I have printed the number of threads yet to come and then printed which thread has completed its Task1 and it goes to sleep. After Last thread arrives, I print all the threads which have woken up and they move on to complete Task 2. N can be any number greater than 1 and we can see it works for any number of threads.

4 Priority synchronization

4.1 Problem Formulation

Executed in file: Priority11.c

This is the part 3 and formulated as follows:

Given the number of threads N, and their given particular priority which can be either High = 1, or Low = 0, both given by user, execute the threads in their priority order as they come and contend for lock. If no high priority threads are present in queue, then only low priority thread should acquire the lock

4.2 Algorithm

The following algorithm was used to execute this primitive:

- Take the input from user of number of threads and the priority the threads should while contending for lock. Priority can be either 0 or 1, with high priority attached to 1.
- As before threads are created in main function
- A barrier is put in the critical section part, to allow all threads to arrive at a junction.
- Now if, a higher priority thread arrives, increase the global variable `remain1`, which keeps track of high priority threads currently in queue or execution. If low priority thread comes, dont increase this variable. This part is Covered by mutex `pthread_mutex_lock(&m4)`.

- Now if `remain1` is not equal to 0, then sleep the thread if its a low priority thread, otherwise if its high priority thread, give thread the lock (in code its mutex `m3`). If `remain1` equal to 0, broadcast all the sleeping low priority threads and wake them up.
- If high priority thread, finishes its job, reduce the `remain1` variable by 1. If low priority threads finishes its job, dont update this variable.
- Last 2 steps are covered by mutex `m2` to take action (sleep or acquire lock) on 1 thread at a time.
- Join all the threads once the finish in main function.

4.3 Result

Use the command: `gcc -pthread -o prior Priority11.c`

Run file using this command: `./prior`

I tested this primitive on various input values, i.e. Number of threads and priority of threads. Each time I have printed, which threads have arrived, how many high priority threads are currently in queue, whether a particular thread acquires a lock or sleeps and completion of work when thread is done its work. Following is the result.

```
(base) adminpc@adminpc-HP-Laptop:~/Documents/Sen8/OS/lab3$ ./prior
Number of Threads: 5
Enter Priority of Threads: 0 0 1 1 1
Thread 4 arrives
Number of current High priority threads in queue or execution = 1
Thread 4 acquires lock with priority 1
Thread 0 arrives
Number of current High priority threads in queue or execution = 1
Thread 1 arrives
Number of current High priority threads in queue or execution = 1
Thread 2 arrives
Number of current High priority threads in queue or execution = 2
Thread 4 work done with priority 1. High priority threads in queue or execution = 1
Thread 3 arrives
Number of current High priority threads in queue or execution = 2
Thread 0 sleep with priority 0
Thread 2 acquires lock with priority 1
Thread 2 work done with priority 1. High priority threads in queue or execution = 1
Thread 1 sleep with priority 0
Thread 3 acquires lock with priority 1
Thread 3 work done with priority 1. High priority threads in queue or execution = 0
Thread 1 acquires lock with priority 0
Thread 1 work done with priority 0. High priority threads in queue or execution = 0
Thread 0 acquires lock with priority 0
Thread 0 work done with priority 0. High priority threads in queue or execution = 0

(base) adminpc@adminpc-HP-Laptop:~/Documents/Sen8/OS/lab3$ ./prior
Number of Threads: 6
Enter Priority of Threads: 0 1 0 1 0 1
Thread 5 arrives
Number of current High priority threads in queue or execution = 1
Thread 5 acquires lock with priority 1
Thread 2 arrives
Number of current High priority threads in queue or execution = 1
Thread 5 work done with priority 1. High priority threads in queue or execution = 0
Thread 4 arrives
Number of current High priority threads in queue or execution = 0
Thread 4 acquires lock with priority 0
Thread 0 arrives
Number of current High priority threads in queue or execution = 0
Thread 3 arrives
Number of current High priority threads in queue or execution = 1
Thread 4 work done with priority 0. High priority threads in queue or execution = 1
Thread 1 arrives
Thread 2 sleep with priority 0
Number of current High priority threads in queue or execution = 2
Thread 0 sleep with priority 0
Thread 3 acquires lock with priority 1
Thread 3 work done with priority 1. High priority threads in queue or execution = 1
Thread 1 acquires lock with priority 1
Thread 1 work done with priority 1. High priority threads in queue or execution = 0
Thread 2 acquires lock with priority 0
Thread 2 work done with priority 0. High priority threads in queue or execution = 0
Thread 0 acquires lock with priority 0
Thread 0 work done with priority 0. High priority threads in queue or execution = 0
```

Figure 3: Priority synchronization primitive result. Number of threads: Left = 5, Right = 6. Priority of threads: Left = 0 0 1 1 1, Right: 0 1 0 1 0 1

Explanation for Left figure: First thread 4 arrives which has priority 1 given by user. So it acquires the lock. Then thread 0 arrives with priority 0. Doesn't get the lock as thread 4 is in progress still. Thread 1 arrives with priority 0. Then thread 2 arrives with priority 1. High priority threads in queue or execution now become 2 (thread 4 and thread 2). Thread 4 is done with work. High priority threads in queue or execution now become 1 (thread 2). Thread 3 arrives with priority 1. High priority threads in queue or execution now become 2 (thread 3 and thread 2). Thread 0 sleeps now since high priority threads still in queue. Thread 2 gets the lock. It is done with its work. Thread 1 also sleeps now, since high priority threads still in queue. And so on. In the end low priority threads 1 and 0 are executed after getting woken up. Similarly for Right figure.

5 Subset Number of Threads Execution

5.1 Problem Formulation

Executed in file: `Subset_Thread.c`

This is the part 4 of q6 and formulated by myself as follows:

Given the number of threads N , and subset number of threads P , execute a primitive where maximum of P of the N threads would run at any moment

The intuition is that, usually nowadays many threads have to be executed in one moment for doing same task. This increases the load on CPU. By passing this input parameter P , we will only allow maximum of P threads to be executed at any moment putting a cap on maximum CPU load which can be possible at any moment, and clients can be rest assured that a particular person wont occupy the complete CPU at any moment. Note: There is no thread ordering assumed, threads can come in any order and fill the buffer.

5.2 Algorithm

The following algorithm was used to execute this primitive:

- Take the input of number of threads from the user. N can be any number greater than 1, where N is the number of threads. Take Number of Subset of Threads working at one time as input.
- Same as before, create that many number of threads using function : `pthread_create` and call the `critical_section` function.
- Write a Task 1 inside the `critical_section` function, all threads will complete this task irrespective of other threads. In my case, Task 1 is a simple for loop, which assigns some constant value to a variable again and again.
- Subset number of threads execution: A global variable `left` is defined which tracks the size of the buffer left. Working threads reduce the value of this variable as they come and Once the buffer becomes 0, sleep all the remaining threads who have completed Task1.
- Now working threads would be executing Task 2, once it is finished, they will increase the buffer size by 1 and broadcast all sleeping threads to wakeup.
- Due to our implementation, only 1 thread will wakeup, reduce the `left` variable to 0 again and rest of the threads will sleep again. In this way, at any moment only P threads will be executing. The shared variable `left` and sleeping and broadcasting calls have been put in 2 different mutexes. 1st mutex for reducing the `left` variable (Buffer size) and making threads sleep and second mutex to increase the buffer size and broadcasting.
Functions used: `pthread_cond_wait`, `pthread_cond_broadcast`, `pthread_mutex_lock` and `pthread_mutex_unlock`.
- Call the `pthread_join` to wait for all threads to come back in main function.

5.3 Result

Use the command: `gcc -pthread -o subset Subset_Thread.c`

Run file using this command: `./subset`

I tested this primitive on various input values, i.e. Number of threads and subset of threads to run each time (Buffer size). Each time I have printed, which thread numbers are executing at any moment and buffer size. As and when thread finishes, I again print that and also print the new buffer size. Following is the result.

```

(base) adminpc@adminpc-HP-Laptop:~/Documents/Sem8/OS/lab3$ ./subset
Number of Threads: 8
Number of Subset of Threads working at one time: 2

Thread 1 running. Buffer Size = 1
Thread 3 running. Buffer Size = 0
Thread 2 sleeping after Task 1 completion
Thread 5 sleeping after Task 1 completion
Thread 7 sleeping after Task 1 completion
Thread 4 sleeping after Task 1 completion
Thread 6 sleeping after Task 1 completion
Thread 3 Finished. Buffer Size = 1
Thread 7 running. Buffer Size = 0
Thread 5 sleeping after Task 1 completion
Thread 2 sleeping after Task 1 completion
Thread 6 sleeping after Task 1 completion
Thread 0 sleeping after Task 1 completion
Thread 4 sleeping after Task 1 completion
Thread 7 Finished. Buffer Size = 1
Thread 2 running. Buffer Size = 0
Thread 5 sleeping after Task 1 completion
Thread 6 sleeping after Task 1 completion
Thread 1 Finished. Buffer Size = 1
Thread 4 running. Buffer Size = 0
Thread 0 sleeping after Task 1 completion
Thread 5 sleeping after Task 1 completion
Thread 6 sleeping after Task 1 completion
Thread 2 Finished. Buffer Size = 1
Thread 6 running. Buffer Size = 0
Thread 4 Finished. Buffer Size = 1
Thread 5 running. Buffer Size = 0
Thread 0 sleeping after Task 1 completion
Thread 6 Finished. Buffer Size = 1
Thread 5 Finished. Buffer Size = 2
Thread 0 running. Buffer Size = 1
Thread 0 Finished. Buffer Size = 2

(base) adminpc@adminpc-HP-Laptop:~/Documents/Sem8/OS/lab3$ ./subset
Number of Threads: 7
Number of Subset of Threads working at one time: 3

Thread 0 running. Buffer Size = 2
Thread 1 running. Buffer Size = 1
Thread 4 running. Buffer Size = 0
Thread 5 sleeping after Task 1 completion
Thread 2 sleeping after Task 1 completion
Thread 6 sleeping after Task 1 completion
Thread 3 sleeping after Task 1 completion
Thread 0 Finished. Buffer Size = 1
Thread 5 running. Buffer Size = 0
Thread 2 sleeping after Task 1 completion
Thread 6 sleeping after Task 1 completion
Thread 3 sleeping after Task 1 completion
Thread 4 Finished. Buffer Size = 1
Thread 2 running. Buffer Size = 0
Thread 1 Finished. Buffer Size = 1
Thread 6 running. Buffer Size = 0
Thread 5 Finished. Buffer Size = 1
Thread 3 running. Buffer Size = 0
Thread 2 Finished. Buffer Size = 1
Thread 6 Finished. Buffer Size = 2
Thread 3 Finished. Buffer Size = 3

```

Figure 4: Subset Number of Threads Execution primitive result. Number of threads: Left = 8, Right = 7. Number of Subset of Threads working at one time (Buffer Size): Left = 2, Right = 3

As it can be seen, in left figure, in left figure, when buffer becomes 0, all threads start sleeping after completing their Task1. As and when a particular thread is finished, buffer size is increased and some other thread wakes up and other threads again sleep.

6 References

References

- [1] https://docs.oracle.com/cd/E26502_01/html/E35303/sync-21067.html
- [2] <https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html>
- [3] <https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html>