

# System Programming 2023 : Programming HW4 Report

Name : 張閔堯

ID : B11902084

---

## 1. Proper References

Programming Part :

曹紹洵(B11902145), 黃昱凱(B11902158), 李冠誼(B11902090), 陳思瑋(B11902027)

Handwritten Part :

APUE section 8.17, 黃昱凱(B11902158), 曹紹洵(B11902145)

## 2. Why Many Possible Answers

Because it's only guaranteed that we accept tasks FIFO, but it's not guaranteed that each task is completed in the order of FIFO because we can't control OS scheduling.

To solve this problem, we can change the implementation of the calling function, for example, build an linked list, when calling this function, allocate the memory to the tail of linked list and preserve a place for it to fill after the function finishes.

After all tasks are finished, print the result from the head to tail.

This guarantees the output is in FIFO of input.

## 3. Mutex and Condition Variable

We use one mutex to protect `pool` from race condition.

The whole pool has one mutex named `mlock`, any thread must get it before doing operation to `pool`

We use one condition variable, there are two usages of it.

First usage is to notify all working thread that there is job to be accepted, and all usable threads would compete for the mutex, the one that gets the mutex would do the work.

The other usage is to notify that all tasks are done, all the threads can exit.

After getting the mutex, the threads would check if it is able to accept the work or it has to exit and do the corresponding commands.

## 4. Avoid Busy Waiting

In the main thread :

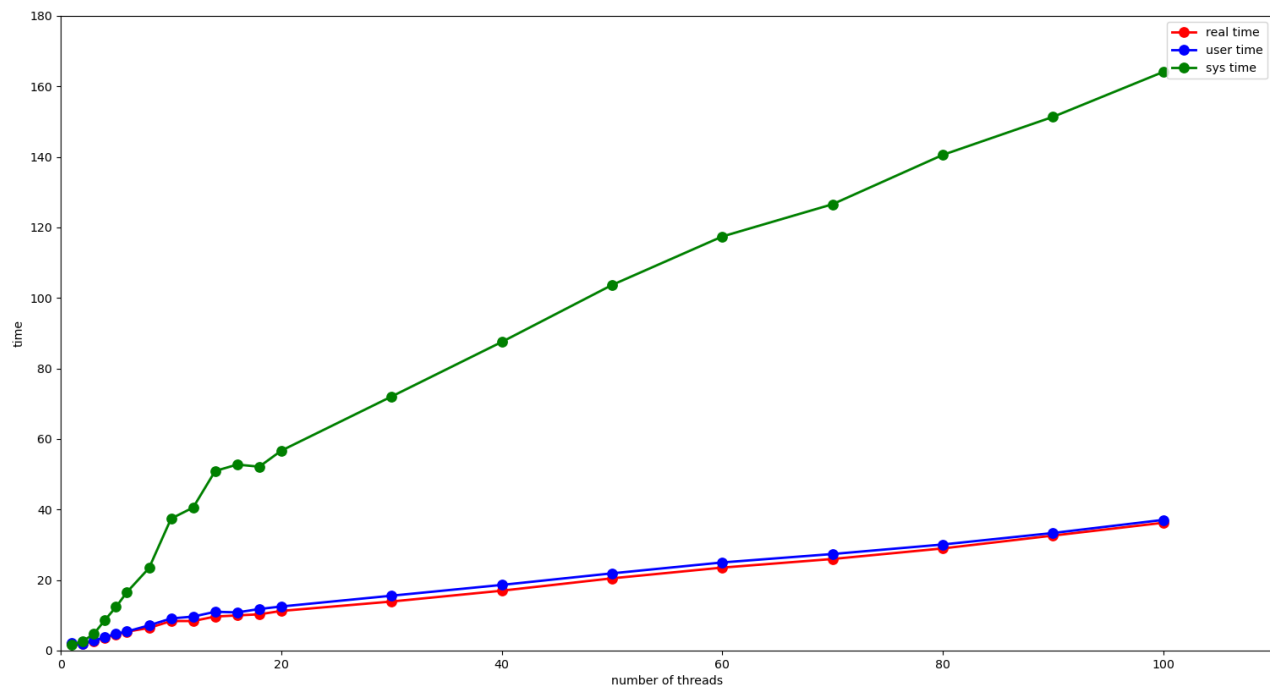
Use `pthread_join` to wait for other threads to exit.

When executing wait, we have to set a bool, and other spawned threads gets the lock and return from `pthread_cond_wait`, it would check the bool. If the bool is modified, that means all jobs are done, it would unlock and tell all threads that the tasks are done, and then exit.

In the working thread :

Use `pthread_mutex_lock` to block the thread until a lock is required and `pthread_cond_wait` to block until the condition variable changes, this guarantees that the process would not keep checking the status.

## 5. Draw a Figure



We use the bench provided in the repo.

The value of user/real time is similar, and it increases with the value of N.

When  $N=8$ , the slope of sys time decreases, the OS may have a different scheduling strategy when N is bigger and smaller than 8.

It's also observed that the system time is much more than user/real time, it may because that the OS spends a lot of time dealing with mutices.

Number of threads	real time	user time	sys time
1	1.91	2.01	1.62
2	1.96	1.9	2.64
3	2.64	2.78	4.72
4	3.65	3.79	8.64
5	4.47	4.77	12.33
6	5.35	5.39	16.56
8	6.4	7.13	23.39
10	8.35	9.08	37.4
12	8.36	9.61	40.54
14	9.63	10.98	50.91
16	9.95	10.8	52.73
18	10.26	11.76	52.13
20	11.22	12.48	56.68
30	13.89	15.54	72.03
40	16.96	18.6	87.51
50	20.47	21.88	103.66
60	23.51	24.97	117.41
70	25.95	27.36	126.56
80	28.95	30.06	140.55
90	32.59	33.32	151.34
100	36.22	37.04	164.08

## 6. ABA Problem

ABA problem is when a location is read by one particular thread twice, if the content doesn't change, we judge that there's nothing changed.

However, there may be another thread changing the content between these two reads.

The implementation prevents this problem because we use mutex.

Using mutex guarantees that only one thread can access the pool at any given time.

Regardless the fact that the status of queue may have changed, there wouldn't a problem because job is taken by another thread, regarding the queue as nothing changed doesn't affect the result.