1. Proper Reference.

https://nachtimwald.com/2019/04/12/thread-pool-in-c/?fbclid=IwAR1EedTlbDSrolKM-Y0lL7W0KQxghFeQRMNzsU8VWYf5j2_aTFTR6DEMau0

2. Specify why there are many possible answers in Sample Testcase? Provide a solution e.g. a `main.c` that can generate output in deterministic order and explain it. (1pt)

Explain:雖然 work 的實作方式是 FIFO，但這只是**確保「先被加進 work queue 的工作先被執行」**，並**沒有保證「先被執行的工作會先被完成」**，因為 thread programming 是由 CPU 去決定現在是哪個 thread 要動作，有可能比較慢拿到工作那個 thread 會不斷被 CPU 排程到，於是反而比較早完成工作。

Solution:我的實作方式是讓 worker(thread)只要在 threadpool 拿到工作就 unlock 了，這樣就有機會在第一個 thread 還在工作時，讓第二個 thread 也拿到工作，就可能產生上述 race condition 的情況。因此，若想避免這種狀況，可**將 unlock()的部分移至處理完工作的時候**。這樣即可確保「**在一個工作處理完之前，不會有任何 thread 去拿另一份工作**」

3. Explain the function of each mutex or condition variable if any. (0.5pt, if no mutex, condition variable, why you don't need them)

I use **mutex** to protect my shared memory(threadpool) from being accessed by multiple threads. When main thread want to add some work into job queue or some thread want to take some work from the job queue(those behavior will affect share memory), I will use mutex to ensure that there will only one thread access shared memory at one time.

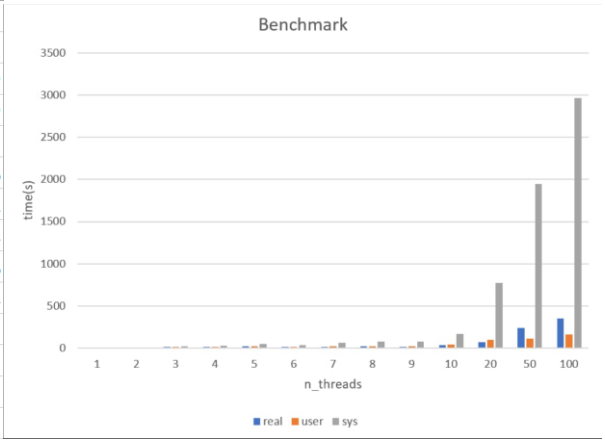
I have two cond var: 1. work_cond 2.working_cond
work_cond is for worker(except main thread) to sleep. The worker will sleep when there are no job to take from queue. And main thread will wake them up when any job be added into job queue.
working_cond is for main thread, main thread sleep when it finishing adding work to job queue, and main thread will be waken up if any worker finish their work, but if there are still job in the queue waiting to be process, then main thread will sleep again. Otherwise, main thread wake up and preparing to destroy threadpool.

4. How do you avoid a busy waiting in both main thread (accept submissions, wait for children threads ...) and different judge threads(take jobs...)? (0.5pt)

For both case, I use **pthread_mutex_lock(),** instead of pthread_mutex_trylock(), that is. When some thread is having a lock and accessing shared memory, then other threads **won't repeatedly check** the availability of that lock. Instead, the calling thread will block until the mutex is unlocked. And **pthread_cond_wait()** will also block(sleeping) calling thread instead of letting that thread repeatedly checking whether the desired condition is satisfied.

5. Plot a *t-n_threads* graph with at least 10 data points, where t is real time, user time, and sys time to deal with numerous time-consuming tasks. The data points in x-axis(n_threads) must follow the constraint:{data points} $\supset \{1, 2, 5, 10, 50, 100\}$. Discuss what you observe. Note: `time` command is your friend. And you can test with `bench/main.c` in the repository for testing. (1pt)

n_threads	real	user	sys
1	3.31	4.14	2.03
2	5	5.66	5.9
3	12.61	14.49	22.79
4	11.32	12.21	27.97
5	17.48	20.22	45.6
6	11.8	13.33	37.42
7	16.2	18.71	61.14
8	18.44	21.03	75.76
9	16.35	18.39	76.34
10	36.44	43.57	169.63
20	72.93	98.21	772.51
50	237.39	109.39	1943.03
100	349.45	159.23	2965.25



First, I notice that for any thread_num, **real_time < user_time + sys_time**, This happen when multiple threads is working on different cores. Since those threads are woking parallely, Their overlapping time will o**nly be count once for real_time**, but for **user and sys, those overlapping part will be count multiple times.** Thus, unlike previous programming(only has one thread(main thread), and there is dependency in it), **in multiple thread programming. The real_time will be less than user_time + sys_time.**

Second, for real, user, sys time, there are trend that the more thread there are, the more time it use.

I had two possible explanations for this:

1. **Synchronization Overheads**: Increasing the number of threads might require more synchronization mechanisms, like **locks**. Causing overhead, and thus decreasing efficiency.

2. **System Resource Limits**: There' limit to the number of threads a system can handle efficiently. Too many threads can lead to **excessive context switching**, thus decreasing efficiency.

6. Bonus: What is ABA problem? Does your implementation resist against ABA problem? Why / Why not?(0.5pt)

I use an example to illustrate what ABA problem is:
Suppose you have a shared memory location containing a value:
Thread 1 reads the value '10' from this memory location.
Thread 1 then changes the value to '20' and performs some operations.
Meanwhile, Thread 2 reads the memory location and sees '10'.
Later, Thread 1 changes the value back to '10'.
Now, if Thread 2 checks the value again, it still sees '10', assuming that nothing has changed, unaware that there was an intermediate '20' state in between.

I do think **my implementation has ABA problem**. In my tpool_worker(), I let worker_thread to sleep when there are no job in job queue. However when some job Are added in job queue, a thread will be waken up by main thread and take a job from job queue, causing job queue be empty again. Under this circumstance, other threads that don't get the lock and sleep again **won't notice that the job queue had been modified**, thus **causing ABA problem.**