7376 HW5

Name: Xuebao Zhao NUID: 002108354

Problem1:

a) Nolock

In this problem, we use four threads to enqueue and dequeue 100000 elements respectively without any locks. First, we create all the enqueue threads and join threads. After that we create all the dequeue threads and join them. Every time one element is enqueued/dequeued, we add one on the corresponding count(line 58 and line 42). Finally, we check the number count.

The implementation is not thread-safe, and there is a data race problem that can occur when two threads access the queue concurrently. For example, if two enqueue threads run concurrently, they can both update the tail pointer of the queue, causing one of the enqueued values to be lost.

For example, consider the case where two enqueue threads execute simultaneously. Thread 1 *"executes q->tail->next = tmp"*, setting the next pointer of the current tail to the new node. Then, before Thread 1 can execute *"q->tail = tmp"*, Thread 2 executes the same line, also setting the next pointer of the current tail to the new node. Now both threads have updated the tail pointer to point to the new node, and one of the enqueued values has been lost.

Also, there will be problem when we free a memory address. Consider the case when two dequeue threads execute simultaneously. Thread 1 execute *q->head = new_head*, setting a new dummy head. Then before Thread 1 can execute it, Thread 2 execute node_t *tmp = q->head. When they execute *"free(tmp);"*, the two *"tmp"* point to the same address. It means it will be freed twice which will cause an error.

The count will smaller than 100000 which we want to enqueue and dequeue.

```
-bash-4.2$ ./nolock
Enqueued 9392 times, it should be 10000.
dequeued 129 times, it should be 10000.
```

Attempted to free a memory address that has already been freed or is invalid.

```
-bash-4.2$ ./nolock
*** Error in `./nolock': double free or corruption (fasttop): 0x00007f813c0008e0 ***
======= Backtrace: =========
/lib64/libc.so.6(+0x81679)[0x7f8154cb8679]
./nolock[0x400980]
./nolock[0x400a44]
/lib64/libpthread.so.0(+0x7e65)[0x7f815500ce65]
/lib64/libc.so.6(clone+0x6d)[0x7f8154d3588d]
======= Memory map: ========
00400000-00401000 r-xp 00000000 00:31 3714301006                         /Users/Grad/bobzhao/7376/hw5/nolock
00601000-00602000 r--p 00001000 00:31 3714301006                         /Users/Grad/bobzhao/7376/hw5/nolock
00602000-00603000 rw-p 00002000 00:31 3714301006                         /Users/Grad/bobzhao/7376/hw5/nolock
01dd6000-01df7000 rw-p 00000000 00:00 0                                  [heap]
7f813c000000-7f813c021000 rw-p 00000000 00:00 0
7f813c021000-7f8140000000 ---p 00000000 00:00 0
7f8144000000-7f8144021000 rw-p 00000000 00:00 0
7f8144021000-7f8148000000 ---p 00000000 00:00 0
7f8148000000-7f8148021000 rw-p 00000000 00:00 0
7f8148021000-7f814c000000 ---p 00000000 00:00 0
```

b) One lock VS. Two locks

In this problem, we also use four threads to enqueue and dequeue 100000 elements respectively. First, we create all the enqueue threads and join threads. After that we create all the dequeue threads and join them. Every time one element is enqueued/dequeued, we add one on the corresponding count(line 58 and line 42). “clock_t” is used to obtain the running time. Finally, we check the number count and compare the running time. The only difference between these two codes is one uses lock while the other uses two.

Result:

```
-bash-4.2$ gcc -std=c99 h5-p1-onelock.c -o onelock -pthread
-bash-4.2$ ./onelock
Enqueued 100000 times, it should be 100000.
dequeued 100000 times, it should be 100000.
The program by using one lock takes: 0.330000 seconds
-bash-4.2$ gcc -std=c99 h5-p1-twolocks.c -o twolocks -pthread
-bash-4.2$ ./twolocks
Enqueued 100000 times, it should be 100000.
dequeued 100000 times, it should be 100000.
The program by using two locks takes: 0.340000 seconds
```

Based on the results of the performance comparison, using a single lock for both enqueue and dequeue operations appears to be better in terms of overall performance. This is because using a single lock reduces contention and overhead associated with managing multiple locks, leading to less waiting time and higher throughput.

c) No dummy

In this problem, we only use one enqueue thread and one dequeue to operate one element. We check if the element we enqueue is the same with the element we dequeue.

Result:

```
-bash-4.2$ ./nodummy
Enqueued 10
Dequeued 1551404796
```

It can be seen that the dequeued element is a wrong value. When using two locks for a concurrent queue, one lock protects the head pointer while the other lock protects the tail pointer. In this case, a dummy node is needed so that the update of the tail pointer is separated from the update of the head pointer. This is important to ensure that a dequeue operation does not encounter a null pointer exception when it tries to access the next field of a node whose next field is currently being updated by an enqueue operation.

Without the dummy node, it would be impossible to differentiate between an empty queue (where head and tail both point to NULL) and a queue with one element (where head and tail

both point to the same node). By inserting a dummy node at the front of the queue, we can ensure that head always points to a valid node, even when the queue is empty. This allows us to distinguish between an empty queue (where head points to the dummy node and tail points to NULL) and a non-empty queue (where head points to the first real node in the queue and tail points to the last real node in the queue).

Problem2:

This program takes two arguments *"num_elements"* and *"num_threads"*, where *"num_elements"* is the number of elements in the array to be summed and *"num_threads"* is the number of threads to be used for the summation.

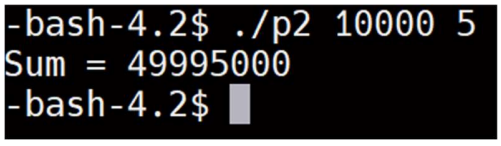
The program first initializes a vector v of size *"num_elements"* and populates it with integers from 0 to *"num_elements-1"*. It then initializes a mutex *"sum_mutex"* for thread synchronization.

The program creates *"num_threads"* threads, where each thread sums a subset of the array v. The runner function is the function that each thread runs. The *"runner_args_t"* struct holds the arguments for each thread, which includes a pointer to the array v, an index indicating the starting position of the subset of the array to be summed, and a size indicating the number of elements in the subset.

Each thread computes the partial sum of its subset of the array and then acquires the mutex lock to update the global variable sum by adding its partial sum to the global sum. Once the update is complete, the thread releases the mutex lock and frees its argument struct.

The main thread waits for all the threads to complete using pthread_join() and then frees the resources and prints the final sum.

Result:



Problem3:

a)

6 4 4 2

Processes	Allocation				Max				Need				Available			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	2	0	1	1	3	2	1	1	1	2	0	0	2	2	2	0
P_1	1	1	0	0	1	2	0	2	0	1	0	2				
P_2	1	0	1	0	3	2	1	0	2	2	0	0				
P_3	0	1	0	1	2	1	0	1	2	0	0	0				

b)

	Processes	Allocation				Max				Need				Available			
		A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
①	~~P_0~~	2	0	1	1	3	2	1	1	1	2	0	0	2	2	2	0
④	~~P_1~~	1	1	0	0	1	2	0	2	0	1	0	2	4	2	3	1
②	~~P_2~~	1	0	1	0	3	2	1	0	2	2	0	0	5	2	4	1
③	~~P_3~~	0	1	0	1	2	1	0	1	2	0	0	0	5	3	4	2
														6	4	4	2

It is safe. The order is $P_0 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1$

c) Assume that a request from P_2 of (2,2,0,0) is granted

6 4 4 2

	Processes	Allocation				Max				Need				Available			
		A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
②	~~P_0~~	2	0	1	1	3	2	1	1	1	2	0	0	0	0	2	0
④	~~P_1~~	1	1	0	0	1	2	0	2	0	1	0	2	3	2	3	0
①	~~P_2~~	~~1~~3	~~0~~2	1	0	3	2	1	0	0	0	0	0	5	2	4	1
③	~~P_3~~	0	1	0	1	2	1	0	1	2	0	0	0	5	3	4	2
														6	4	4	2

It is still safe if the request is granted. After P_2 get (2,2,0,0). It can finish its job and return all the resources. And the available will be (3,2,3,0). It is enough for other processes. The order will be $P_2 \rightarrow P_0 \rightarrow P_3 \rightarrow P_1$