Homework 4 Theory

Michael Tang

due June 5, 2019

1 Exercise 108

After the locks are grabbed, the only value checked within curr is curr.key, to ensure that it is not the same as the key of the item being added. Since no operation ever alters the key, we would not need to worry about concurrent changes to curr.key. curr.next is never referenced but pred.next is, because pred.next is the only one that needs to be altered to add the item. Therefore a lock for curr is never actually needed for add().

2 Exercise 109

Contains() is not linearizable because without locks there is no linearization point. Without lock and validation the operator would go straight from setting pred and curr (line 55) to checking if curr.key == key (line 60), and there is no way to ensure at line 60 that curr even still exists in the set. For example, Thread 1 could set pred and curr at line 55 and before it executes line 60, Thread 2 removes curr. Then if curr.key == item.key Thread 1 will return true, which is incorrect since curr is not in the set.

3 Exercise 121

A similar implementation as the node-based queue, except instead of dynamically generating nodes we create an array that is the size of the bound. Head and tail will be index numbers that are continually incremented with deq() and enq(). To obtain the item that head or tail are referring to, we would index into the array by head/tail modulo bound. enq() fails when $tail - head = bound$ while deq() fails when $tail - head = 0$.
To satisfy (1), the appropriate head/tail lock would be obtained before a thread is allowed to increment either.
To satisfy (2):
It is simple enough to do away with the head lock; a deq() operation would entail taking the element at the head, then atomically incrementing head. At least quiescent consistency is maintained: there is no guarantee of which order three concurrent deq() calls would execute in, but at the end of those three calls, head will be correct.
For the tail lock, we could follow the same model of logical enqueue followed by physical enqueue, where the logical enqueue would be adding the item to index $(tail+1)\%bound$ (so long as $tail-head < bound-1$) and the physical enqueue would be changing the tail pointer. Again we run into the problem of these two

steps not being atomic, and the fix that we did for lock-free queues does not seem to work as well. Unlike simply looking ahead to tail.next we would need a way to know that a logical enqueue happened. If we do, and check an index position n indices beyond tail, we also need to check that $tail + n - head < bound$.

4 Exercise 124

1. Yes, because the read and the change of the head pointer is done in one atomic step with CAS. So, the order in which deq() appear to happen depends on the order of calls on the CAS step.
2. No, because the true effect of enq() is the logical enqueue, actually adding an item to the tail end of the queue. So, the instantaneous order that enq() calls will appear to happen in is tied to the order of logical enqueues, not the actual change of the tail pointer. The tail pointer may be changed in different order.

5 Exercise 125

1. deq() is only lock-free if the queue is never empty; otherwise, it will block the system looping until the queue isn't empty. Even when the queue is never empty deq() is still only lock-free and not wait-free, because if there are multiple calls to deq() and one is slower than all the rest, it will be continually forced to keep going forwards looking for a non-null item to dequeue. enq() is lock and wait-free because its execution does not affect the system (there are no loops) and does not hinder other threads.
2. enq()'s linearization point is line 7, where it inserts the item in to the array. This way the order in which enq()'s appear to happen is visible by the order of the items in the queue. deq()'s linearization is line 13, but only if it obtains a non-null item because that would be the only case where it returns the desired visible result.

6 Exercise 127

1. We initialize the array with all null contents, and initialize the *top* pointer to be the last index in the array. push() calls would decrement this pointer while pop() calls would increment. We are able to tell if the stack is full by seeing if $top == 0$, and empty if $top == bound$. Both push() and pop() calls would attempt to grab the single lock and push/pop the top of the stack. If a thread performing push() encounters a full stack or pop() encounters an empty stack, the thread releases the lock and goes to a sleep pool designated for push or pop. When a thread successfully performs a push call, they send a wake signal to the pop sleep pool, and vice versa.
2. We can use atomic increments and decrements on *top*. The problem arises from trying to read elements at the top in the same atomic step. Atomic get and increment operations would only get the index of the array, not the actual element. A potential fix is having *top* reference the actual address of the top element and not the array index, but then the point of using a bounded array would be partially defeated as new addresses must be allocated for new items to swap into head.

7 Exercise 129

In the LockFreeStack object, it is better for push to have its own backoff object and pop to have its own as well. This is because failure of *trypop*() is not only around the change of *top* but also due to empty stack. If pop and push use the same backoff object then multiple pop calls that fail due to empty stack will significantly slow the adding of elements via push calls.
For the EliminationBackoffStack, threads that fail to acquire the lock can enter the EliminationArray. Also, threads holding WAITing slots in the EliminationArray can sleep instead of spin to free system resources. This sleep would last until a thread that is trying to pair with the slot issues a wake signal. Then, that pairing thread would perform a CAS. The LockFreeExchanger is still linearizable on the CAS with this.

8 Exercise 132

1. The two steps of getAndIncrement/getAndDecrement and inserting/referencing the actual item in the array are not atomic. For example, a call to *pop*() could getAndDecrement *top* to get its needed index i, but then before it is able to reference $items[i]$, another thread could call *push*(), increment *top*, and change $items[top]$. Then $items[i] = items[top]$ and not was actually supposed to be dequeued in that space.
2. For both *push*() and *pop*(), before the retrieval and change of *top* the thread would try to enter a push or pop room for its respective operation. Following the rules of the Rooms interface, only at most one of those rooms can be occupied at all times. If the thread fails to get into its needed room, it sleeps until the *onEmpty*() handler is called and tries again. If a thread successfully gets into its room, it does the indexing and element retrieval operations and then exits the room. Concurrent *pop*() and *push*() calls are already linearizable with respect to calls of the same type, and this prevents concurrent *push*() and *pop*() calls from overwriting each other incorrectly.

9 Exercise 159

If no sentinel nodes are used and an entry that is being pointed to by a bucket is deleted, then problems can arise even if it's the only entry of that hash. Two CAS's must be performed: first on the *.next* of the node preceding the one we want to delete, then on the bucket pointer to remove it. There is no way to make those two steps atomic, so concurrent calls could potentially, for example, *add*() new entries of the same hash by calling the still-extant bucket and tacking them onto the node that has already been logically removed from the list.
Also, in this case, if the bucket concerned is parent to other buckets, re-adding it causes inefficiencies not intended in the recursive split algorithm.

10 Exercise 160

Assuming $N = 2^i$, this worst-case scenario could happen if you want to add a bucket whose bit representation is i 1's, and there are no initialized buckets whose least significant bits are 1. Then, you need to initialize $i - 1$ parent buckets of form 1...0, where 1... consists of $i - 1$ 1's. So in total $i \approx log(N)$ buckets will need to be initialized.

The probability of the above worst-case happening is approximately $2^{-(i-1)}$, which is very low if i is high. Much higher probability is that more recent parent buckets are already initialized, so based off of this, new bucket initializations should on average take $O(1)$ time. This proof was given by Shalev and Shavit.

11 Exercise 185

$S_p(n)$ is the amount of steps required to mergesort a list of length n with p processors.
For work:
$S_1(n) = 2S_1(n/2) + \theta(1) = \theta(n)$
Critical section length:
$S_\infty(n) = S_\infty(n/2) + \theta(1) = \theta(log(n))$
since sorting on each new division of list can be handled fully in parallel. So, parallelism is $S_1(n)/S_\infty(n) = \theta(n/log(n))$.

12 Exercise 186

The students' version is optimized by adding more operations to the critical section that make 1-processor operation more efficient, and predictably make the critical section longer. At a lower amount of processors like 32, the substantial non-critical improvement is higher than the loss of critical section brevity. But at 512 processors:

$$T_P = 2048/512 + 1 = 5$$
$$T'_P = 1024/512 + 8 = 10$$

So, visibly, the original implementation scales better to the much higher 512 processors.

13 Exercise 188

First, we can apply the last bound for greedy schedulers to both the 4 processor and 64 processor measurements. We have:

$$80 \leq \frac{T_1 - T_\infty}{4} + T_\infty \text{ and } 10 \leq \frac{T_1 - T_\infty}{64} + T_\infty$$
$$T_1 + 3T_\infty \geq 320 \text{ and } T_1 + 63T_\infty \geq 640$$

Subtract those two equations from each other to get:

$$60T_\infty \geq 320$$
$$T_\infty \geq 5.333333... \approx 5.33$$

We can see from the greedy scheduler inequality that to get the lowest possible upper bound on T_P for any $P > 1$, we would pick the lowest possible $T_\infty \geq 5.33$, which is $T_\infty = 5.33$.
Applying this T_∞ back into the greedy scheduler inequalities for 4 and 64 processors and solving for T_1 gives us $T_1 \geq 303.97$ and $T_1 \geq 303.57$ respectively. We need the higher of the two bounds for the selection of T_1 to be correct, and want the lowest possible T_1 given those bounds to minimize T_P, so $T_1 = 303.97$.

Just to check that these selections are correct, we observe from the first inequality ($T_P \geq T_1/P$) that $T_1 \leq 320$, which our selection satisfies. Also, the second inequality states that $T_\infty \leq T_P$, for which the lowest bound we have is $T_P = 8$. Our selection of T_∞ is ≤ 8.
Plugging our selection of the best possible T_1 and T_∞ back into the greedy scheduler inequality, we get:

$$T_{10} \leq \frac{303.97 - 5.33}{10} + 5.33$$
$$T_{10} \leq 35.19$$

The fastest the professor's computation can possibly run on 10 processors given the data is approximately 35.19 seconds.

14 Exercise 192

Measuring queue sizes twice is not atomic, so taking those measurements outside of a lock can be problematic. For example, $q0$ could be originally smaller than $q1$ at line 1, but then before line 2 is run several $enq()$ calls could have been made to $q0$ to make it bigger than $q1$, in which case the thread will mistakenly have $q0$ as both $qMin$ and $qMax$, diff will incorrectly equal 0, and the load balancing will not happen. Even if $qMin$ and $qMax$ are correct after line 2 the sizes can still change before the lock is called.