James Maciej Krywin
Multicore Programming

Homework 3

1. Consider the following two threads accessing shared global variable `thr_glob`. (a) What kind(s) of concurrency bug(s) are present in this code? (b) **Describe** (ie, don't just list) two possible ways to fix this. If it helps, use code to demonstrate why these are good fixes.

Thread 1	Thread 2

`if (thr_glob->proc_info) {`
`  fputs(thr_glob->proc_info);`
`}`
	`thr_glob->proc_info = nullptr;`

2 Solutions:

a. Mutecies – place a **mutex_lock** before and a **mutex_unlock** after the presented code for threads 1 and 2 or even for a better concurrency the programmer can utilize a reader/write lock. The read lock would encompass the code for thread 1 while the write lock would surround the code for Thread 2.

Thread 1	Thread 2

pthread_mutex_lock(&mutex);
if(thr_glob->proc_info) {
  fputs(thr_glob->proc_info);
}
pthread_mutex_unlock(&mutex);
	pthread_mutex_lock(&mutex);
	thr_glob->proc_info = nullptr;
	pthread_mutex_unlock(&mutex);

b. Compare And Swap – introduce a compare an atomic CAS method for fputs. [Depending on the CAS implantation, if it returns true, the user may double check the values and print.]

Thread 1	Thread 2

temp = thr_glob->proc_info;
if(CAS(temp, thr_glob->proc_info, thr_glob->proc_info){
  fputs(temp);
}
	thr_glob->proc_info = nullptr;

2. Why does adding a sentinel simplify a concurrent queue? How does it make it possible to construct a lock-free queue?

Adding a sentinel node to a concurrent queue removes the variance or every having a head or tail pointer pointing to null. Thus, the algorithm only needs to address how to efficiently switch targets of a pointer. Additionally, since the addition and removal of nodes will never require to adjust more than one pointer per action (modify only the head or tail at a time), both actions can occur without locks. The pulling threads will only be modifying the head pointer while the pushing threads will only be using the pointer. Simply, put it prevents from altering the head and tail pointers from ever overlapping.

3. What causes the ABA problem? How can you solve it, and why does that solve it? What extra hardware and software primitives, if any, are required to implement a fix to the ABA problem?

The ABA problem occurs when a node or memory block becomes reused. A pointer can only differentiate whether it is pointing to a different memory location but not if the memory object is different. Thus, if a node is reused to represent a new node of information, it may still be treated as the old object in a CAS. Therefore, in 64-bit systems, 128-bit pointers exist that have a free list tracker as well as additional pointer information stored in the additional bits.

4. What is linearizability? If an algorithm or technique is linearizable, what does that mean in practice? For example, you could motivate this with a CAS example, or another example of your choosing.

Linearizability is the characteristic that allows a set of operations to occur in an atomic and synchronized manner. An operation or computation is linearizable if its resulting state is comparable to the state after a linear and sequential round of the operations. In practice, this translates to lock-free queue containing a sentinel node, being able to being shared by parallel threads for multiple actions. This guarantees some form of order and where the state of each action is comparable to the actions from a parallel run(i.e. there is no intermediate state for actions).

5. Speaking of CAS:

a. What are the strengths (useful features) and weaknesses (limitations of what you can express) in thread-safe algorithms using CAS?

b. Review our CAS-based queue enqueue() method, and to the best of your ability, explain why we need to allow the `tail` pointer to point to either the last or second-to-last entry in the queue instead of only the last one.

Think about the properties of CAS, what we talked about in class, and the slides. Show how you think: if you still don't quite understand it from class, you may want to explain your thought process as you try to work through examples of how two threads might interleave their enqueue() operations (not just the one explained in the slides). You should come up with some fundamental deductions about this and other CAS-based algorithms and the global states visible to other threads in a CAS-based algorithm that wouldn't be visible with mutex-based critical sections. Again: when in doubt, show your work!

a. One weakness of the CAS method leads to all cache stores needing to flush out their value and update it from the main memory. Having a program that heavily relies on such atomic operations would lead to a dramatic hit to performance. `A strength of using the CAS operation is that it does not use any locks resulting in a n implementation that cannot end up deadlocked. Use of locks runs the risk of a thread becoming starved by having another thread never release the lock.

b. The reason for using the following two pointers instead of just a tail pointer is in order to ensure another enque() did not act on the tail element:
elem* cur_tail = tail; elem* next = cur_tail->next;
This would lead to the method updating the "next" pointer of the last element to node A while it is already pointing to node B. Since the updating of both pointers (next and tail) is not an atomic operation, an enqueue'd node has the potential of turning into a stray orphan node.

<u>Example (different than in class):</u>

<- Tail (pointing at Z)
Head -> [X] -> [Y] -> [Z] -> NULL

Thread 1 reads in the tail pointer as Z.
Thread 2 reads in the tail pointer as Z.
Thread 1 adds [B] as a the next node to [Z] resulting in the following:

<- Tail (pointing at Z)
Head -> [X] -> [Y] -> [Z] -> [B] -> NULL

Thread 2 adds [A] as the next node to [Z] resulting in the following:

<- Tail (pointing at Z)
Head -> [X] -> [Y] -> [Z] -> [A] -> NULL [B] -> NULL

Thread 1 updates tail pointer to node [A]
Thread 2 updates tail pointer to node [B] resulting in the final state:

<- Tail (pointing at A)

Head -> [X] -> [Y] -> [Z] -> [A] -> NULL [A] -> NULL
