Alexander Nguyen

Multicore HW 4

1.

    a. It can spawn two more threads to run in parallel while a game is playing. More threads can be spawned but they would run concurrently instead of in parallel. This limit is from having a 4-core machine in which only two more threads are available after two are taken to play the video game.
    b. I would optimize for throughput given that there is a large number of prime numbers that need to be dealt with and the game. With more throughput, we are able to begin progress on the other prime numbers sooner. From a system-wide perspective, we are able to use more cores to process more threads and more prime number factors which is more efficient if we optimized for latency.

2. A cause of the ABA problem is allowing a CAS to succeed because a pointer has the correct address but may not be the correct pointer temporally. A simple example is using CAS on the front of a queue and having the front change to another node and then change back to the original node. Because CAS would still succeed since the original node points to the correct address, we must prevent it by:

    a. Additional Pointer Information:

    With a 64-bit x86 computer, we are able to include a 64-bit counter and can then manipulate to further challenge when a CAS should and should not succeed regardless of seeing the same address pointer

b. Free List Tracking:

Track the freed pointers in the freed list and increment the pointers if they are ever reused.

The above methodology prevents the ABA problem by ensuring that we are considering the correct pointer both spatially as an address and temporally via counter before allowing a CAS to succeed.

3.

a. Muticies

i. Push(): Mutex around the function to prevent additional data race pushing or reading.

ii. Pop(): Mutex around the function to prevent data race readings.

iii. Size(): Mutex around the function to guarantee the size returned is the

actual value for the issue that the size may change while accessing the size

of the circular queue. Prevents read and write pointers from moving.

b. Yes, it can be slightly more performant with Reader-writer locks

i. Push(): Lock a global read lock to allow push and pop at the same time. Lock a local write lock to prevent other pushers from appending to the buffer and altering what the write pointer points to.

ii. Pop(): First check if the size is larger than 1. Lock a global read lock to allow push and pop at the same time. Lock a local write lock to prevent other poppers

from moving the read pointer as data is being read and is meant to be moved forward appropriately.

iii. Size(): Lock a global write lock to prevent pushing and popping. Have a local read lock to allow other readers to read the size at the same time and prevent writers from either pushing or popping the circular queue.

c. No, it can't be more performant with CAS

i. Push(): Cache the current write pointer and first check if it has changed. If it has not changed by the first CAS, then write to it. Do a second CAS and if the write pointer still has not changed then move the pointer forward. If it has changed, update the cached write pointer to the current write pointer and restart from the beginning.

ii. Pop(): Cache the current read pointer. Cache the current value from the read pointer. Perform CAS, if the read pointer has not changed, move the pointer forward and return the cached current value. If it has changed, restart from the beginning.

iii. Size(): It is not more performant to use CAS to find the size of the buffer given that we must check if both write and read pointers have not moved. However, when we check if one has moved, the other may also move and vice-versa, causing a loop. In keeping track of two pointers, it may not be feasible to use CAS to return the current size with thread safety. Because size is not achievable with CAS, it cannot be more performant than the previous two methods.

4. Bugs

a. After thread 1 passes the if statement, thread 2 can change thr_glob to be null

which can lead to issues in fputs()

b.Fixes

i. Use a mutex surrounding the if statement and surrounding changing thr_glob to null

ii. Use a reader-writer lock. Read lock for the method in thread 1 and write lock for changing the thr_glob to null.