Read The Art of Multiprocessor Programming by Maurice Herlihy & Nir Shavit

Read several articles on Locklessinc.com/articles

Including Spinlocks and Read Write locks

23/01/14

Agreed with David Gregg to implement a Ring Buffer as the first data structure. I decided to go for a simple design in C to begin with. I implemented it using pthreads and two modes of operation; the first used mutexes in a simple lock and unlock fashion while the second used a spinlock. However, the implementation of the spinlock proved tricky due to the low level nature of C and so I decided early on to start using C++, in order to access the higher level intrinsics, since my focus was not on the implementation but rather the testing of these structures.

<http://people.csail.mit.edu/edya/publications/OptimisticFIFOQueue-DISC2004.pdf> --Gives details on implementing a buffer using CAS.

27/01/14

Researched several papers on the topic of concurrent data structures, placed them into the research folder

Amended the output of the program to be easier to paste into excel for graph generation

Gathered data from stoker for both locked and spinlock modes of ring buffer

Need to gather data from my machine, spoon and ducss

28/01/14

Gathered data from ducss, netsoc and local machine. VPN has only 1 core so no point testing on that. Tested assembly spinlock against C++ version, performance negligible.

29/01/14

Added in several references to the report

30/01/14

Added in a CAS lock with back off to the ring buffer and collected data from all 4 machines

02/02/14

Added a TAS and ticket lock to all 4 architectures and collected data

03/02/14

Got perf + nice working. Came to the conclusion that a graph should be done for each size and mode of operation possible. In addition, started work on a linked list.

06/02/14

Added to the ring buffer section of the report, gathered data on the CAS lock with no back off.

11/02/14

Implemented an assembly version of the ticket lock to the ring buffer to compare against the c++ ticket lock. The difference was negligible.

Started gathering data from stoker using perf

Started gathering data from cube, perf not installed not, have requested

12/02/14

Gathered data on the ring buffer from cube using perf. In addition I …

…implemented a spsc lockless ring buffer and gathered data on it from stoker, cube and my local machine both normally and with perf

13/02/14

Finished off the locked version of the linked list and messed around …

…was CAS in test.cpp

17/02/14

Added modes of operation TAS with pause and TTAS with no pause for ri…

…ng buffer, collected data on stoker,local and cube

18/02/14

[Finished locked linked list, add modes of operation and gathered data…](https://github.com/gibsonma/FYP/commit/2a2e7cf234a6634fe4b9357f2ffda65a83fa0de9) […](https://github.com/gibsonma/FYP/commits/master)

… from stoker, local and cube. Added CASLOCKND which is a CAS lock but with no delay. In addition, I ran tests to determine if locked performance differed with different key ranges. In most cases it does not, though with a very small key range such as 10, progress slows drastically.

19/02/14

Added code for a lockless linked list using compare and swap and gathered data for three key ranges, 100, 100000 and 100000000 from stoker, cube and local.

22/02/14

Added to the report

25/02/14

Created graphs for ring buffer and linked list both locked and lockless

26/02/14

Gathered data from linked list using perf

01/03/14

Implemented four more modes into linked list, TTAS\_RELAX, TAS\_RELAX, CASLOCK\_RELAX & TICKET\_RELAX which all use \_mm\_pause() instead of sleep. Note that CASLOCK does well and TICKET is now working as expected.

Did the same for locked Ring buffer, not lockless and added the data to the sheet

04/03/14

Implemented MPMC, a lockless linked list, with a head and a tail. Nodes are constantly added onto the head and removed from the tail. This is to eliminate the time used for a list to be traversed to find the point of entry. I also created its own data table. Stoker currently busy but will use averages to get data if necessary.

05/03/14

Gathered data from the lockless MPMC buffer, no need to use averages as stoker was not in use.

Implemented locked version of MPMC buffer along with all modes of operation. Also gathered data from it

07/03/14

Implemented mpmc\_alt\_lockless, an alternative to the linked list buffer which switches the head and tail so remove the need for a prev pointer. I now need to test this, implement a locked version, gather data on both and compare to the original version.

08/03/14

Implemented the locked version of mpmc\_alt and gathered data on both. Doesn’t appear to be much difference between the two…

09/03/14

Updated data for linked list, spsc and ring buffer, ensuring all had data for 128 key range and perf data

13/03/14

Tried to encapsulate head into a List struct but ran into problems calling list->add in pthread\_create

15/03/14

Started implementation on the lockless hash table. Wrote the add but ran into a strange occurrence, where occasionally the iterations posted for one loop would be very low, going from the hundreds of thousands down to the thousands. I added a fail variable to see if the cause was perhaps linked to an unusually high failure rate in the atomic compare\_exchanges but that was not the cause.

I also decided to add in both head and tail atomic variables into the List object, as if I had not, removing a node would involve traversing the list, which may not best highlight the performance of the lockless algorithm. Hence, the linked lists used almost identical code from the standalone linked list that I wrote.

I ran into an issue where it occasionally seg faulted at high thread counts. I noticed that I was directly accessing the hash table a lot during both add & remove in the form of htable->table[hash]…. I thought that this may be the reason for the seg fault, as halfway through an add, another thread may change the value of hash among other things. I decided to combat this by assigning htable->table[hash] to a List pointer and in addition, I added checks for several of the conditionals, checking that tmpList still pointed to the same point as htable->table… and that another thread had not interfered. To test to see if this had fixed the problem I set the program to run itself 20 times, ensuring that by the end if a seg fault was going to appear that it would do so at some point. This was not the case and so it would seem to be working correctly now.

Implemented the locked version of the hash table, hash\_locked.cpp which I implemented and quickly tested to check if it worked. I now need to implement a resize method for both versions and gather data on the two. I may implement it in a new file and compare the two.

16/03/14

Added the resize functionality to the locked hash table today. I define a global constant and when any list in the table exceeds that in length, a new table of 2\* the size it created and the lists are copied over. I ran into issues as using with the pthreads, so in the end I chose to have the resize function non-threaded but since it will only be called in a critical section it does not matter.

21/03/14

I decided that to properly replicate the behaviour of an actual hash table, I needed to add in a contains function which searches for an item within the table. This needed to be done for both locked and lockless versions. As a possible additional metric I added in two variables, pSearches and nSearches to track the number of successful/failed searches. I plan on comparing the its/s when they are used, especially for the lockless version as they will need to be atomically incremented. Adding a third method has additional problems, I can no longer say that ½ of the threads should add and the others should remove as there are now three actions. For now, what I have done is that each thread now calls the choose function which randomly assigns them a function which I can use to determine the number of threads performing an action as I can do rand() % num and then if x > 4 do contains else if num > 2 do add etc and change the % of threads doing which task.

I added in the ability to count how many contains, adds and removes were performed by adding in some counts.

I then began modifying the resize method to use the one described in “Concurrent Hash Tables”, with the plan being to implement this and in addition modify it so that each list has a lock.

22/03/14

I decided to put the resize function on hold and divert my attention to modifying the locked list so that each list had its own lock, so when a thread entered a critical section it need only lock the list it was in, and not the entire table, as with hash\_locked.cpp. This proved relatively easy as, instead of locking down the whole table, I instead just locked by list. I now must compare both hash\_locked and hash\_locked\_per\_bucket to see if there is a performance difference. Initially there does not appear to be a major difference when using plain mutex locks, though per bucket seems to have a slight advantage at higher thread counts.

I have attempted to add a resize functionality to the lockless hash table but as of yet I have been unsuccessful, I feel that this may be a step too far and that I may need to leave it unimplemented, or failing that, implement the locked resize function that I have already.

Started work on my presentation slides.