# Discussion Notes

**04/05/2017**

* Use Clock Monotonic
* Make sure you use gnuplot 5
* Follow the order specified in the spec
* When you turn on yield, the running time of your function slows down
* For sorted list, don’t use spin lock

**Project 2B**

* Use gpreftools
* taskset -> gives number of CPU cores the application could use
* Different devices give a little different outputs  
    
  **Where do you believe most of the cycles are spent in the 1 and 2-thread list tests ?**
* 4 different cases: 1-thread with spin lock -> most operations go to list operations

1-thread with mutex -> intuition: if list large, most cycles in list operations. If list small, then we do not know and we need to write experiments to verify this

2-thread with spin lock, -> we have two threads, when one thread is modifying the lsit, the other one is spinning. Hence, we can say 50% of CPU cycles go to spinning and 50% to list operations

2-thread with mutex -> if list large -> most cycles go to list operations. If list small -> possible that mutex will take most of the CPU cycle

* **Why do you believe these to be the most expensive parts of the code?**
* **Where do you believe most of the time/cycles are being spent in the high-thread spin-lock tests?**
* In spin lock (while spinning)
* **Where do you believe most of the time/cycles are being spent in the high-thread mutex tests?**
* It is possible that this is in the list operations assuming the list is large
* To find gperf -> -v pprof
* Always use the debug option: make sure you have –g turned on when compiling \*.o files
* Use the command to locate the library then provide the location for the library
* In the text output, focus on the first and fourth number
* If you use the sample code, then you should be done with the gperf part of the project
* **Where (what lines of code) are consuming most of the cycles when the spin-lock version of the list exerciser is run with a large number of threads?**

**Why does this operation become so expensive with large numbers of threads?**

* Why is the wait for lock time higher than the completion time? Wait for lock consists of time happening in multiple CPUS. Computing time is just wall time and only consider one CPU
* **QUESTION 2.3.3 - Mutex Wait Time:**
* **Look at the average time per operation (vs. # threads) and the average wait-for-mutex time (vs. #threads).**
* **Why does the average lock-wait time rise so dramatically with the number of contending threads?**
* **Why does the completion time per operation rise (less dramatically) with the number of contending threads?**

**How is it possible for the wait time per operation to go up faster (or higher) than the completion time per operation?**

Addressing the underlying problem: Hash will pick up one key, decide which sublist it should go to

How to get the length of all the sublists? If there are 20 sublists, then there are 20 sublists: just lock them all, calculate the length and release the locks

It is impossible to feel circular dependency here. When you are calculating the list size, make sure all in same order

When we add more sublists, the throughput increases

* QUESTION 2.3.4 - Performance of Partitioned Lists
* Explain the change in performance of the synchronized methods as a function of the number of lists.
  + Throughput increases because there is lest contention
* Should the throughput continue increasing as the number of lists is further increased? If not, explain why not.
  + If we have one list, then the Since there’s so many sublists, then the possibility of two threads modifying the same sublists is low. Once we have reached the perfect parallelism, we cannot continue increasing the throughput
  + If we continue increasing the number of sublists, the list size will keep getting smaller. Insert and lookup are both O(n). The time for each operation is going to keep decreasing, which means we will spend fewer time in the list operation. Eventually we will reach an upper bound for the throughput

It seems reasonable to suggest the throughput of an N-way partitioned list should be equivalent to the throughput of a single list with fewer (1/N) threads. Does this appear to be true in the above curves? If not, explain why not.