# 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.

**Friday 18th May Notes**

* Copy the temperature code from the following link: <http://wiki.seeedstudio.com/Grove-Temperature_Sensor_V1.2/>
* Sudo aptget to install the library
* Login with root in order to access the temperature
* The reading is non blocking
* -DDUMMY #ifdef DUMMY, then do something
* gettimeofday, set the second one to
* Turn the switch to 5v and not 3v

**Friday 25th May Notes**

* There is a timeout in the test script, if your code takes a long time, change the timeout and see if your code passes the cases with that
* [**http://web.cs.ucla.edu/classes/cs111/Samples/**](http://web.cs.ucla.edu/classes/cs111/Samples/)-> to download files one by one
* If there is no newline, don’t have a new line. Need the **EXACT SAME OUTPUT**
* [**http://www.nongnu.org/ext2-doc/ext2.html#DISK-LAYOUT-SAMPLE-20MB**](http://www.nongnu.org/ext2-doc/ext2.html#DISK-LAYOUT-SAMPLE-20MB)Table 3-1. Is our case for the basic output
* Pread explanation: offset specifies from which point you start to read
* (struct \*) super = (struct \*) buf
* If the value is 1, then it is used, if not, then it is not used

DUMMY CODE FOR READING

char \*buf = malloc(1024);

pread(fd, buf, 1024, 1024);

(struct superblock \*) sp = (superblock \*) buf;

Getting bitmaps -> use a bit operation to read a specific bit from the byte. Be careful!

INODE SUMMARY

* Store all the information about files, you need to do the typecasting again
* Size of each inode entry is a static number
* 13. Print out the pointer information. **(Missing in the spec??)** For each inode, there are 15 pointers. The first twelve are direct pointers, they point to some data block. 13: Points to other block numbers, which point to data. 14: One indirect pointer 15: Indirect pointer to an indirect pointer (triple I believe)
* Important to pay attention to:
* For ordinary files (type 'f') and directories (type 'd') the next fifteen fields are block addresses (decimal, 12 direct, one indirect, one double indirect, one triple indirect). Symbolic links are a little more complicated. If the file length is less than the size of the block pointers (60 bytes) the file will contain zero data blocks, and the name is stored in the space normally occupied by the block pointers. If this is the case, the fifteen block pointers need not be printed.
* Follow the allocated definition for checking inodes: non-zero mode and non-zero link count

DIRECTORY ENTRIES

* Valid still means non zero inode number, follow the definition.
  + **How do we find the parent inode:** when you scan through the inode, go to data block.
  + **DO NOT GO DEEPER INTO THE SUBDIRECTORY**
  + For data block, size of each entry is not the same
  + For each inode, we need to scan every data block. But we have 15 pointers, when do we stop? We try to scan through all 15. First 12 -> go to data block. If 13 is 0 (empty), then go to 14. If it is not 0, go deeper to find the entries there.
  + The file is broken, so there might be some invalid information, before you encounter valid information.
* Scan through all pointers including the indirect pointers.
* For printing out the name, follow the name length!

INDIRECT BLOCK REFERENCES

* Logical byte offsets are the most misleading ones, If the referenced block is data block, logical offset means position in the file (because if you put all data blocks together you form a file) For the second data block, the logical offset is 1024 (because first 1024 is contained in first data block)
* Drawing: Block 0: 0~1023 Block 1: 1024~2047
* If the referenced block is a single or double-indirect block, this is the same as the logical offset of the first data block to which it refers: **MEANS:** For block 13, we point to somewhere, and then somewhere else. The final destination follows the 12th block in the file. So offset of 13th is 1024 \*12.
  + For 14th one: 1024\*12 + 1024\*256 (bc 13th one points to 256 data blocks)
  + Logical offset: How many data blocks are located before the current block, you concatenate them all!