# Nov / 02 / 2020

## Multithreading Part ll

* Linux Source Files and how to measure them
	+ Source program of linux-5.8-rc7
		- Many files are small in size
		- Most header file <2.3 and 6.25 KB
		- Most .c files are <100KB
	+ Linux Command Line to find the above :
		- Compute number of files of a given type: find [directory] -name \*.h | wc -l (-l:count the number of lines)

Compute file size: find linux-5.8-rc7/ -name \*.c| xargs du -b | awk ‘{print $1}’ | sort -n | uniq -c > c\_file\_stats , du-: estimate file space usage , sort -n : sort in numerically

* + wc++ example

 Worker thread 1

Main thread Worker thread 2 Final merge & sort

(collect the path names; Worker thread 3

Spawn worker threads) …

 (process the file)

* + Measuring Performance
		- Not printing results on console(which may affect the performance)
		- Evaluation: local laptop (4 physical, 8 logical cores) vs. remote server (32 physical, 64 logical cores)
	+ Bash script to measuring the performance

(taskset: define which threads to use)

Taskset 0x01 [./wc++ ]

Questions:

1. 4 threads use 4 physical cores?
2. 4 threads using 1 core? Hyperthreading, performance not necessarily improve, since it doesn’t mean 4 times parallel
	* Breaking down process of wc++:
		+ Find\_all\_files (sequential) → t\_process\_file (parallel) → t\_merge (seq <-> parallel) → n\_files\_processed → n\_bytes\_processed → t\_sort
		+ based on the breakdown, could decide how to optimize it
		+ 1~4 threads time spent breakdown
		+ Too large number of threads will change the bottleneck to merging and sorting. For example, 64 threads to run wc++ (introduce contention to enter the same critical section)
	* Changes and comparison
		+ How to change and access the map
		+ How to parse and read the file content: word-by-word
	* Static file assignment to thread:
		+ Statically partition and assign each file to the threads
		+ Advantage: don’t need std::atomic<uint64\_t> to find the unique file indexes

Files may exit at different times, which may be good to avoid contention at the merging step.

* + - Disadvantage: if the files are unevenly distributed, then many threads may need to wait for a very slow thread.
	+ Performance improvement
		- Measure and compare the time spent for different file sizes
		- Gprof
		- Idea 1: batch file processing --- merge multiple files into a single string and process them -no apparent improvement
		- Idea 2: parallel merge --- thread 0, thread1 merge ; while thread2, thread3 merge … → thread 0, thread 2 merge ; while thread 4, thread 6 merge …..
	+ How to measure the performance ?
		- Self-define: util get\_time

uint64\_t util::get\_time(){

 static const auto start\_time = std::chrono::stead\_clock::now();

 const auto end\_time = std::chrono::stead\_clock::now();

 uint64\_t nanoseconds\_elapsed = std::chrono::duration<double, std::milli> (end\_time - start\_time);

 return nanoseconds\_elapsed;

}

* + - No printing during the measurement (Printing is expensive to perform)
	+ Additionally:
		- Mutex: use lock( ... );
		- Shared\_mutex : use shared\_lock(mutext\_type& m);