2382854p, Alex Paterson
SP3 AE2

1. Status

To the best of my knowledge, my solution works fully. I have submitted a multithreaded solution that fully conforms to the specification.

My solution works by implementing thread safe structures for s_workQ (a work queue that is a list of file names that have to be processed) and s_theTable (a hash table mapping file names to a list of dependent file names), generating a number of threads depending on the "CRAWLER_THREADS" environmental variable, manipulating these threads to safely and efficiently process the inputted file names and printing out each files' dependencies.

I create and manage my worker threads using a vector of threads. The size of this vector is varied by the value of "CRAWLER_THREADS" using a for loop and "emplace_back". Each worker thread operates until s_workQ is empty, at which point each thread finishes its work, stops operating and eventually finishes. In order to signal to the main thread that all the worker threads have finished, each worker thread is joined using ".join()".

My "process" function operates in a thread safe way by locking and eventually unlocking a global mutex called "process_mutex" when altering "s_theTable" and "s_workQ".

My solution:

- Works with one worker thread.
- Works with multiple worker threads.
- Compiles successfully with no warnings.
- Has appropriate thread safety implemented for Working Queue and Hash Map.
- Has efficient mechanism for determination when worker threads have finished.
- Works correctly with the files in the test folder.

2. Build and sequential & l-Thread runtimes

a.

```
-bash-4.2$ pwd
/users/level3/2382854p/ae2/files
```

b.

```
-bash-4.2$ make
clang++ -Wall -Werror -std=c++17 -o dependencyDiscoverer dependencyDiscoverer.cpp -lpthread
-bash-4.2$
```

c.

```
-bash-4.2$ time ./dependencyDiscoverer_sequential -Itest test/*.c test/*.l test/*.y > temp

real    0m0.056s
user    0m0.008s
sys     0m0.015s
-bash-4.2$
```

d.

```
-bash-4.2$ export CRAWLER_THREADS=1
-bash-4.2$ time ./dependencyDiscoverer -Itest test/*.c test/*.l test/*.y > temp

real    0m0.055s
user    0m0.010s
sys     0m0.015s
-bash-4.2$
```

3. Runtime with Multiple Threads

a.

```
-bash-4.2$ pwd
/users/level3/2382854p/ae2/files
-bash-4.2$ export CRAWLER_THREADS=1
-bash-4.2$ time ./dependencyDiscoverer -Itest test/*.c test/*.l test/*.y > temp

real    0m0.048s
user    0m0.008s
sys     0m0.013s
-bash-4.2$ export CRAWLER_THREADS=2
-bash-4.2$ time ./dependencyDiscoverer -Itest test/*.c test/*.l test/*.y > temp

real    0m0.027s
user    0m0.011s
sys     0m0.012s
-bash-4.2$ export CRAWLER_THREADS=3
-bash-4.2$ time ./dependencyDiscoverer -Itest test/*.c test/*.l test/*.y > temp

real    0m0.022s
user    0m0.016s
sys     0m0.011s
-bash-4.2$ export CRAWLER_THREADS=4
-bash-4.2$ time ./dependencyDiscoverer -Itest test/*.c test/*.l test/*.y > temp

real    0m0.019s
user    0m0.012s
sys     0m0.015s
-bash-4.2$ export CRAWLER_THREADS=5
-bash-4.2$ export CRAWLER_THREADS=6
-bash-4.2$ time ./dependencyDiscoverer -Itest test/*.c test/*.l test/*.y > temp

real    0m0.018s
user    0m0.010s
sys     0m0.020s
-bash-4.2$ export CRAWLER_THREADS=8
-bash-4.2$ time ./dependencyDiscoverer -Itest test/*.c test/*.l test/*.y > temp

real    0m0.019s
user    0m0.014s
sys     0m0.024s
-bash-4.2$
```

b.

CRAWLER_ THREADS	1	2	3	4	6	8
	Elapsed Time	Elapsed Time	Elapsed Time	Elapsed Time	Elapsed Time	Elapsed Time
Execution 1	0.117s	0.029s	0.021s	0.019s	0.016s	0.016s
Execution 2	0.044s	0.027s	0.024s	0.020s	0.018s	0.021s
Execution 3	0.037s	0.037s	0.032s	0.017s	0.018s	0.018s
Median	0.044s	0.029s	0.024s	0.019s	0.018s	0.018s

c.

a. Multithreading takes advantage of multicore computers by running a program on more than one core at the same time. The effects of multithreading are clear to see up until the fifth column of results where CRAWLER_THREADS is equal to 6. We can see that the median execution time decreases as more worker threads are used (until the fifth column) – this is because when two threads are used, these threads can be run at the same time on two cores, and when three threads are used, three cores can be used and so on.
In theory, twice as many cores means twice as much power. However, the experiment does not convincingly affirm this theory, as although it is close, the difference of median execution time between, for example, 2 threads and 4 threads is not exactly half. This is likely due to other factors such as current server CPU load and slight differences occurring during runtime slightly affecting execution time. However, the general trend of the median execution time roughly supports the theory.
The reason the elapsed time stops decreasing significantly after the fourth column is because the university's stlinux server that I ran my solution on has a quad core processor. The solution is running on the maximum amount of cores when working with 4, 6 and 8 worker threads, and therefore cannot get any faster - this is evidenced by the minimal difference between median execution times of solutions with threads 4, 6 and 8.

b. The elapsed times for each thread generally does not vary significantly . This is because the program is essentially always doing the same thing, and any variation in execution time is usually down to external variables such as CPU load.
One exception to this that we can see is the very first time value recorded – Execution 1 for CRAWLER_THREADS 1. This is due to the server needing to cache the program (as it was the first time in a while it was being executed) and thus taking more time to do so.