Robert Chung

Dr. Dingler

CSC 3350

Due 6/1

Homework 5

1. The current version of the threaded program (file\_search\_threaded\_1) works by scanning the top-level directory and spawning threads that recurse on each of the files/directories found in that top-level directory. We also use 4 threads to accomplish this task. The difference is that each thread carries out a recursion from the top-level directory, whereas in the original file\_search, we essentially used a blocking single-thread to accomplish all the recursion.   
     
   In terms of performance, I saw no real difference between the threaded version and the non-threaded version. In finding around 6700 items, both the threaded *and* non-threaded applications had a running-time of around 500ms. There was no real clear distinction on which was faster.
2. One reason is clearly listed in the Extra-Credit. Indeed, in this case, the program (at least the my current implementation) is *IO-Bound*. Therefore, the benefits from multi-threading are not too noticeable.   
     
   Furthermore, I argue multi-threading must be implemented efficiently; in my current implementation, there are clear bottlenecks (as explained in II). Therefore, a 4x speed-up is not necessarily going to result from using more threads. The implementation has to be very efficient and carefully thought out in order to utilize the benefits of using threads.
3. Yes, because the way that I implemented multi-threading, the threads are only created from the top-level directory. Therefore, in a hypothetical scenario, there could be such a structure that thread 1 is created to recurse on a very nested directory, whereas threads 2-4 are merely files (would only recurse to depth 0). Since pthread\_join() is a *blocking* function, even though threads 2-4 are finished, it would still have to wait for thread 1 to finish before being assigned any meaningful tasks.  
     
   In another implementation, we could have it so that as soon as a thread is done, main gives it a new task from the top-level directory. In this case, there can also be a performance hindrance if say—the first thread is recursing on a very nested structure, whereas the rest of the items in the top-level directory are merely files. In this case, even if threads 2-4 are non-blocked and given tasks while thread 1 is still trying to complete its task, thread 1 might take significantly more time after threads 2-4 are completely finished going through the top-level directory. Therefore, at least in my current implementation, the file structure does matter in regards to performance.
4. Indeed. The 2nd paragraph of II points out a different approach that would be more optimal. This is more optimal because the threads are non-blocked, meaning they can get useful work while other threads may be finishing their tasks. This is different in that my original version waits for *ALL* the threads to be completed before I assign the threads new tasks. Obviously, this is a bottleneck, since not all threads finish at the same time; there is an obvious waiting time for the thread before it is assigned any new task.  
     
   Another implementation could be that the threads themselves spawn threads as they see fit. For example, thread 1 could be used on the first file found in the top-level directory. Then, thread 2 would be used for the first file found in the potentially nested file found. This can go on. In this manner, the threads would always have useful work and would have to be non-blocking. However, this would be optimal in that we would solve the problem of nested file structures hindering the performance. This is because no matter the depth of the recursion, a thread could be created within the nested structure.
5. I believe one workload that will outperform the original with threads is sorting (the full path). Thus, if the objective was to recursively discover all files, sort, and print the result, it should be faster to have it be threaded. This is because sorting will be CPU intensive with all the string comparisons that it must do. Indeed, string comparisons are a CPU designated task, which means for big enough workloads, the program will end up being CPU bound.  
     
   Another workload that will outperform the original with threads might be something like finding the shortest path between two files. The program would recursively discover all files from the starting directory and create a graph structure that represents the file structure. Then, it would take two file names and see what the closest path between those two are. This would be CPU intensive because:  
   1) The program must create a graph  
   2) The program would need to find the starting file (starting node)—which *could* be O(1) with hash-tables, but that would mean more memory used. We can maybe assume it is going to be O(n) for search so that we don’t use extra memory.  
   3) It would run Dijkstra’s algorithm for shortest path between two nodes, which is indeed CPU intensive.  
   Therefore, with a big enough file structure, it will be more CPU intensive—in other words, it would be CPU bound.
6. In section 40.7, it mentions that reading a file is expensive; it states that “with a long file name, the file system would perform hundreds of reads just to open the file.” Therefore, the OS uses caching and DRAM to speed up the task for subsequent reads. We know from previous readings and general information about caches that it is most definitely faster than reading from disk. Therefore, if the cache is found to have information about the file, it is a lot faster. With this in mind, the subsequent reads after the first read are faster because the information is likely in the DRAM, which means there is a high chance of not needing to perform the actual IO operation.