Will Czifro

Report

This is a compilation of my notes so far as well as a couple of paragraphs discussing asynchronous technology.

Asynchronous Overview:

Using asynchronous technology was a very explicit decision in this project. Asynchronous work flow utilizes a "non-blocking" method of doing computations. The idea is to maximize CPU utilization while waiting for non CPU based operations to finish and yield control back to the CPU. This is especially useful with IO operations, either to disk or to network.

Choosing a framework for asynchronous tasking was based on the scheduling policies discussed in [1] [2]. Between the two policies, "work-first" seemed like a better policy than "help-first" for this project. The "work-first" policy allows workers to steal pending tasks from each other. By stealing work, a worker is never idle. The .NET framework utilizes this [3]. The TPL library evolved in .NET 4.5 to improve how asynchronous tasking functions through the use of a state machine and parsing and generating code marked as asynchronous. This process optimizes how code is marked as a continuation of a task, especially when a task spawns another task.

Syme et al. do a comparison between languages that support similar asynchronous tasking. F# out performed other frameworks as well as C#, which utilizes the same framework. Even though both C# and F# run on top of the same virtual machine, Common Language Runtime (CLR), the F# implementation of asynchronous tasking and the F# compiler, which also uses parsing and code generation, provides more performant state machines for asynchronous tasking.

Checksumming Performance Overview:

Checksumming is used to ensure the transfer succeeded with zero data loss. The decision was made to use hashing algorithms provided by the .NET Framework. A test was performed to see which of the available algorithms would be the most efficient in three areas: computation time, computation memory usage, and hash length. These are the results:

```
Data Size => 1MB
MD5    Time: 2ms, Memory Usage: 42270 bytes, Hash Length: 16
SHA1   Time: 1ms, Memory Usage: 47226 bytes, Hash Length: 20
SHA256 Time: 2ms, Memory Usage: 42106 bytes, Hash Length: 32
SHA384 Time: 2ms, Memory Usage: 42188 bytes, Hash Length: 48
SHA512 Time: 1ms, Memory Usage: 47226 bytes, Hash Length: 20
```

Time wise, both SHA1 and SHA512 ran the fastest, but had used more memory than the others. SHA256 and SHA384 had the lowest memory use, but generated large hashes, which would eventually have an impact on memory. MD5 seems the most optimal with a memory usage barely more than SHA256 and SHA384, the same computation time, and the shortest hash. Using a short hash is acceptable given how it will be used.

Each partition of the file will have its own checksum. The checksum will be calculated by processing blocks of the partition, referred to as block hashing. These blocks will act like "sub-partitions". If a block does not match, then the block needs to be retransmitted. Block hashing was tested using two different methods, backlogging and incremental. The incremental method was implemented by the C# data structure `IncrementalHash`, which would update the hash as data was given to it. Backlogging required storing `BLOCK_SIZE` worth of data before passing it to the hashing algorithm. Both methods produce a hash once a `BLOCK_SIZE` worth of data has been processed. A simulator was created to show the difference between the two methods. The two methods were also compared against hashing the data all at once, referred to as straight hash. Computation time and memory usage were benchmarked by the simulator. The goal is to have a marginal performance degradation going from straight hashing to block hashing. These were the results of the first test:

```
Data Length => 1KB
Hash Length => 1664 bytes
Number of iterations => 100
Block Size => 10 bytes
Incremental Hash Memory Usage: ~46KB, Time: ~0 ms
Backlog Hash Memory Usage: ~40KB, Time => ~0 ms
MD5 Hash Memory Usage: ~40KB, Time => ~0 ms

Data Length => 1MB
Hash Length => 1632 bytes
Number of iterations => 100
Block Size => 10KB
Incremental Hash Memory Usage: ~86.9MB, Time: 114 ms
Backlog Hash Memory Usage: ~105.5MB, Time: 127 ms
MD5 Hash Memory Usage: ~40KB, Time: 1 ms
```

Early implementations of block hashing was not efficient, there was a test for `1GB`, but that ran for nearly 5 hours without finishing. The F# language lends itself to be written recursively, which is alright in scenarios where the functions do not process a lot of data. After switching from recursion to iterative, these were the results:

```
Data Length => 1KB
Hash Length => 1664 bytes
Number of iterations => 100
Block Size => 10 bytes
Incremental Hash Memory Usage: ~40KB, Time, 3597 ns
Backlog Hash Memory Usage: ~45KB, Time: 2999 ns
MD5 Hash Memory Usage: ~40KB, Time: 1134 ns
```

```
Data Length => 1MB
Hash Length => 1616 bytes
Number of iterations => 100
Block Size => 10KB
Incremental Hash Memory Usage: ~40KB, Time: 5 ms
Backlog Hash Memory Usage: ~40KB, Time: 4 ms
MD5 Hash Memory Usage: ~40KB, Time: 1 ms

Data Length => 1GB
Hash Length => 1600 bytes
Number of iterations => 100
Block Size => 10MB
Incremental Hash Memory Usage: ~259MB, Time: 439 ms
Backlog Hash Memory Usage: ~286MB, Time: 615 ms
MD5 Hash Memory Usage: ~40KB, Time: 618 ms
```

The iterative method has had a tremendous performance gain. As the results indicate, the block hashing matched or marginally did better straight hashing. This was not done asynchronously either. The decision was made to do this synchronously because checksums needed to be generated in order, which is not guaranteed with asynchronous tasking. Since backlogging seemed to have an edge over incremental in performance with smaller files, backlogging will be used for smaller files. Increment will be used for larger files since the test showed a significant performance difference between incremental and backlogging.

File I/O Performance Overview:

File I/O operations are very expensive. Since the most a UDP packet can carry is 512 bytes, very small chunks of the file would be read at a time if every packet was a file I/O operation. The .NET Framework provides a data structure the maps portions of a file to memory [6]. This data structure was the model for the data structure for this project, which required more control over how the file was mapped to memory. The portion in memory acts like a view to whats in the file. The .NET data structure does not allow for views to shift once created. The data structure for this project needs a view that can shift along the file.

A simulation was run testing the performance of this custom data structure. The scenario was a zero latency network transfer with zero data loss. However, checksumming is a part of the process for reading and writing, but the speed of this as shown above makes any latency negligible. The scenario was simulated by simply copying the file from one location on disk to another. The simulator was run with 1, 4, 8, and 12 views, also referred to as partitions. The partitions was disjoint and equal in size with the exception that if the file size did not divide evenly by the number of partitions, then the remainder would go to the last partition. The following were the results of the 4 simulations:

```
File Size => 362310289 bytes => 362.3 MB

Partition Count => 1
Transfer Time: 468 ms
```

```
Partition Count => 4
Transfer Time: 280 ms

Partition Count => 8
Transfer Time: 280 ms

Partition Count => 12
Transfer Time: 247 ms
```

These results were very disappointing. There was not the performance gain that one would expect. Understandably, parallelization of file I/O is not going to have much performance gain since there is only one data bus to send data back end forth to disk. Attempts were made to do random access, but this did not work. This is due to the inability to turn off I/O buffering on Unix based operating systems, which is negating attempts to write in random locations. The feature does exist in the OS, but the .NET Framework only supports this for Windows. The test environment being used is macOS Sierra. This operating system opens files differently than Windows. On Windows, there is a call to `kernel32` using a method called `P/Invoke`. On Unix based systems, files are opened through custom C code that is externalized. In order to be cross platform, Microsoft has left out certain features that Windows supports since most versions of Unix based operating systems do not all operate the same way, especially with respect to file I/O.

To try and remedy this issue, custom C code was written to open a file with no buffering on macOS. The way of doing on macOS uses different flags than on Linux. So in order to support a Linux server for test that will require using the network, C code needs to be written for Linux as well. The problem faced with this is that `P/Invoke` is set up a vary specific way with the .NET Framework. There are problems with locating the C code and documentation on how to properly do this is very minimal. The next available option is to try and see if latency can be reduced by eliminating the in-application buffering used when writing to file. The data structure uses a buffering scheme that ensures the write buffer does not meet capacity and that the read buffer does not go empty before loading more of the file. Perhaps modifying the data structure to just write data and make use of the buffer in the operating system, maybe there will be a significant gain in performance.

What is next:

Next is to work on the network aspect. A protocol was designed and implemented, but a simulation needs to be created to see how performant it is. This simulator will encapsulate using the memory-mapped file as well as the checksumming. The tests will include using a loopback network connection as well as a transfer between the current development environment and a Linux server. Another test will be used to test doing transfers between a Linux server stationed in New York and a Linux server setup on EWU's network, which is claimed to be slow by both faculty and students. These tests should be sufficient to see if multiple asynchronous UDP connections can quicken file transfers.

Resources:

[1] Guo, Y., Barik, R., Raman, R., & Sarkar, V. (2009). Work-First and Help-First Scheduling Policies for Terminally Strict Parallel Programs. Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (IPDPS'09), 1–12.

[2] Blumofe, R., & Leiserson, C. (1999). Scheduling Multithreaded Computations by Work Stealing. ACM, 46(5), 720–748.

[3] D., Schulte, W., & Burckhardt, S. (2009). The design of a task parallel library. ACM SIGPLAN Notices, 44(10), 227.

[4] Ekberg, F. (2013). What does async & await generate?

[5] Syme, D., Petricek, T., & Lomov, D. (2011). The F# asynchronous programming model. In Practical Aspects of Declarative Languages: 13th International Symposium (Vol. 6539 LNCS, pp. 175–189).

[6] Microsoft. Memory-Mapped Files. Retrieved from https://msdn.microsoft.com/en-us/library/dd997372(v=vs.110).aspx.