Juan Caraballo
10/11/18
Project 4 Report

Test System:

- Quad-Core Intel i5 Processor
- 8GB Ram
- 256GB Hard Disk
- Ubuntu 18

Summarized Results:

There is no clear solution for what equates to the best configuration for running programs and reading files as quickly as possible. However with a combination of multiple process cores that may run and an enlarged file read buffer, seemingly linear performance is obtained. In essence, as file sizes grow by a factor of 10, the time required to read them grows incrementally.

For project 4 the class was tasked with creating a program that would search any given file for a given search string. Implemented through the use of the read() and mmap() system calls, functionality was also expanded through the use of multithreading. Once complete we ran project 1's "doit" program on the proj4 executable to collect usage statistics on the use of mmap() and read(). Unfortunately on my system major page faults are not recorded so I could not supply data concerning these. However, I was able to collect wall-clock times for each test run as shown in figures 1 and 2 below:

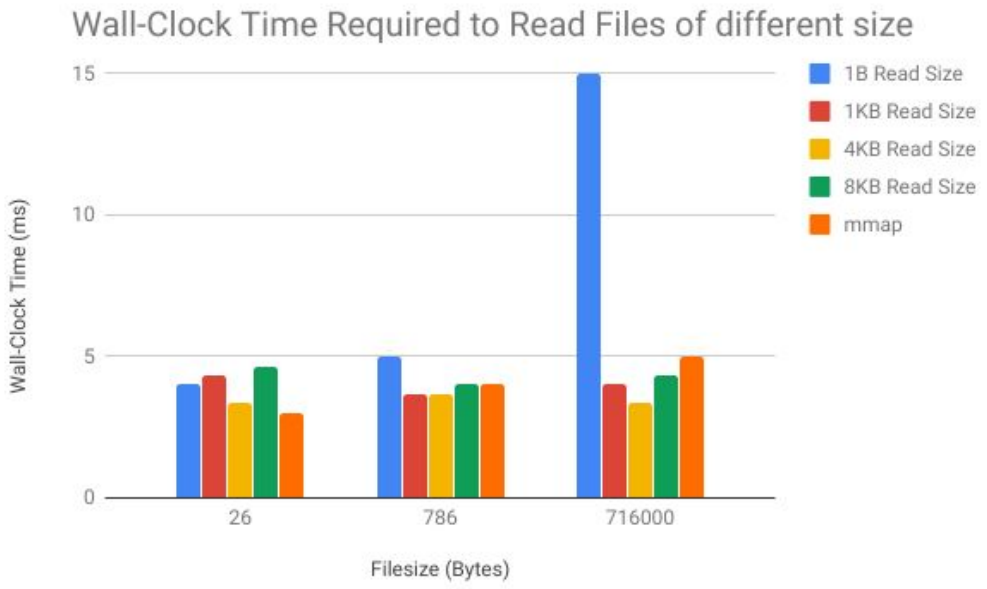

Figure 1: Bar chart of Wall-clock Times taken on different file read implementations

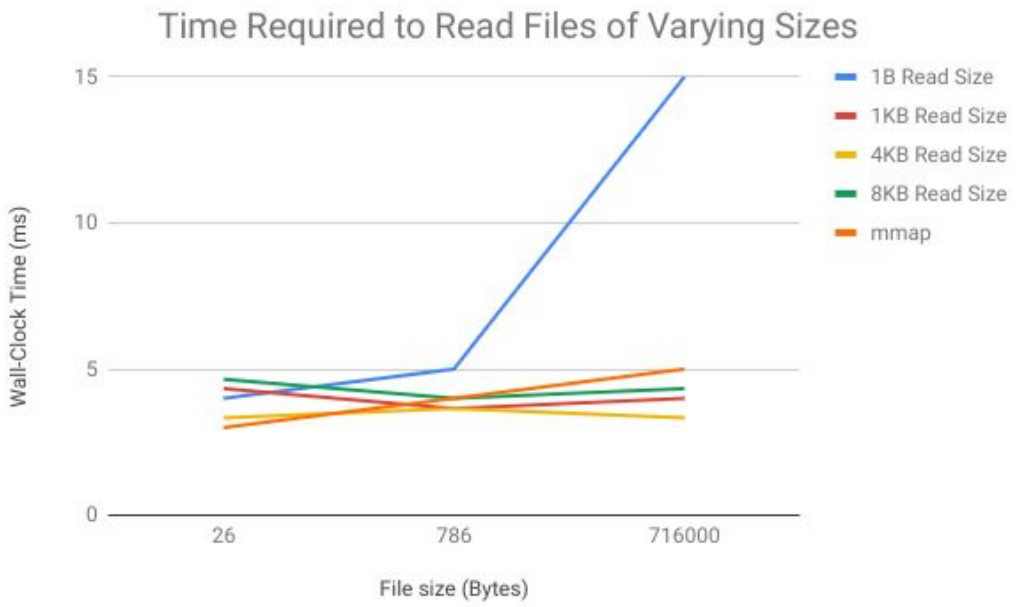

Figure 2: Line plot of Wall-Clock Times taken on different file read implementations.

Comparing the data collected for read size of 1B-8KB and mmap, a conclusion can be drawn that for all file sizes, the size of the buffer used to store read data does not matter so long as it is “large enough”. Essentially, if a buffer is large enough to place full size words into, processes can efficiently read and process files of most sizes in a comparable amount of time. This is suggested by the 716KB file read done with a read buffer of size 1B. On the two charts above this data point persists to be the largest, indicating that time was wasted when buffers have to be continually grown to, at least in our case, search for additional characters. One interesting note however is that although mapping the file contents to memory took approximately the same amount of time as using the read() system call, as the file size grew on the order of 10, the time required to read that file grew by 1ms.

Interestingly as we look at the multi-threaded scenario below it looks like a better performance impact is made with a greater number of threads. However, similar to the scenario where a buffer greater than what is needed makes no difference, once a certain number of threads is reached, additional threads serve to harm performance. As shown in figure 3 below once the number of threads surpassed four, performance began to degrade. This makes sense as my system has four processing cores and only process may be run at a time per core.

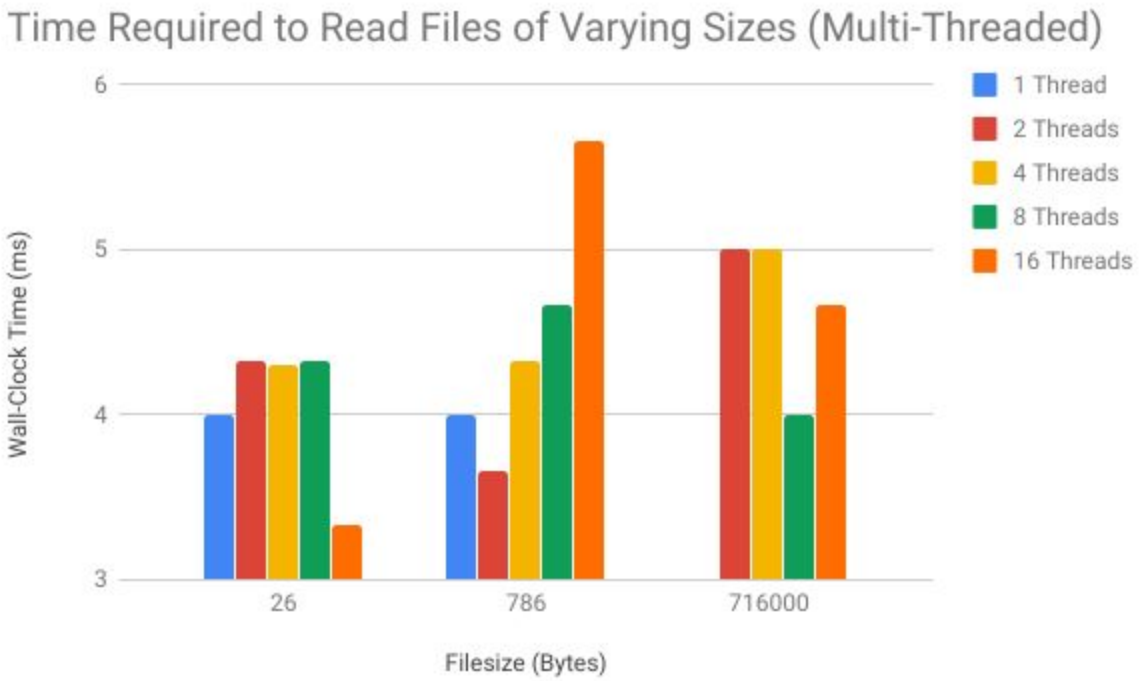

Figure 3: Bar chart of Wall-clock Times taken on different file reads with varying thread counts