 There are expected discrepancies between the multi-threaded application vs the multi-process one. Creating a new thread is relatively inexpensive because most of the memory is shared between threads. A new thread just needs to copy over its own stack. On the other hand, when processes are created, the whole memory segment must be duplicated, which takes a significant amount of time. Similarly, context switches between threads is much faster than for processes because the caches and the program context can stay the same.

However, multithreading can be slower in several conditions. If threads don’t take advantage of the shared memory space, then it loses an advantage of not having to copying over lots of data. Also, since threads may access and change the same variables, there many need to be locking mechanisms, which cause slowdowns because threads need to wait to continue execution. This similarly holds for the semaphore, which ensures no more than 1000 files are open.

In our case, multithreading is slower, as shown in the graphs.

The time is most directly correlated with the number of total threads/number of total processes. The time increase for each new threads is about an order of magnitude greater than for processes. This is likely because for our threading application, we merge all the sorted csvs in a massive csv, which also explains why the time is linear with number of files. If we didn’t put all the csvs in one massive csv, then our threading application would take a lot less time. Theoretically speaking, since merge sort is O(n\*log(n)), and we consider doing sorting on a files, then for the multithreading application, since we sort ALL the data, the big O is a\*n\*log(an), and for the multiprocessing application, since we sort a files each with O(n\*log(n)), the big O is a\*n\*log(n). The ratio of these two complexities is [a\*n\*log(a\*n)] / [a\*n\*log(n)] = [log(a) + log(n)] / log(n). Since the number of entries in each file, n, is very large compared to the number of files, n, this ratio is approximately unchanging, which explains why the first two graphs are BOTH linear (and off by a factor).

The multithreading can be faster if when we use mergesort to sort each csv, we spawn two more threads to mergesort each half of the dataset. This way, more operations are parallelized. However, if there are too many threads, then the creation of threads and context switching will destroy the benefits of multithreading. There is a point of diminishing returns with respect to adding more threads to the program. Another way to speed up the multithreaded algorithm is to keep a fixed number of threads with which to do sorting, instead of creating and destroying a thread for each file. We would first index all the files to be sorted, then pass pointers to these files into each of the threads. This would also remove the need to use a semaphore to restrict the total number of threads created, and speed the total program up.

       Mergesort is definitely the best sorting algorithm for multithreading. This is because mergesort divides the task of sorting into parts, which can individually be delegated to separate threads. These threads are independent from other threads in that all they need is a set of data to sort on. Then their results can be aggregated and sorted by yet another thread.