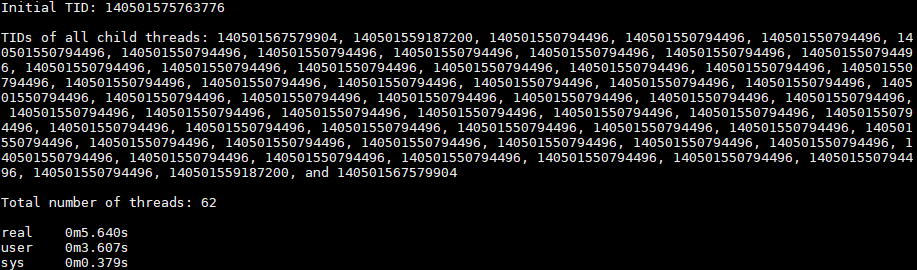
Analysis.pdf

1. Is the comparison between run times a fair one? Why or why not?

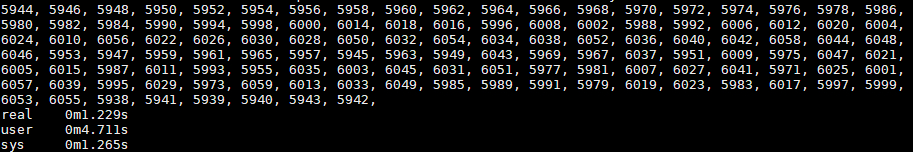
Using the project descriptions, no, the comparison isn’t fair. Project 2 requires you to merge all the individually sorted files into one big file, while Project 1 only requires the sorting to be done and outputted. In order to make the comparison fair, I modified the multithreading code to output all files to one specific folder, and modified the multithreaded code to do exactly the same thing. However, depending on how efficient your multithreaded code is compared to the multiprocessed code, it really shouldn’t make much of a difference in making a comparison, because the multithreaded code should be miles faster than the multiprocessed code anyway.

1. What are some reasons for the discrepancies of the times or for lack of discrepancies?

MULTITHREADED OUTPUT:



MULTIPROCESSED OUTPUT:



The main cause for these discrepencies is that in the multithreaded version of the code, we had to use locks to make sure the data was being accessed and written to without any issues or race conditions. Ideally, multithreaded should be faster since it doesn’t have to spend as much time copying the entire heap over to a new process like forking does.

1. If there are differences, is it possible to make the slower one faster? How? If there were no differences, is it possible to make one faster than the other? How?
2. Is mergesort the right option for a multithreaded sorting program? Why or why not?

Absolutely. The great thing about mergesort is that each section of data that is being sorted is being worked on separately. This means that once the entire array to be sorted is split up, every single thread can do comparisons in parallel, so there’s no waiting on the other half to complete its comparisons to then merge. This is theoretical though, since in practice there’s no way you can have as many processors as threads you create. So theoretically, if you DO have as many processors as threads you create, the big O of mregesort would be O(logn). But in practice, it’s somewhere between O(nlogn and logn), and depends on the number of processing cores your computer has. On computers with anything more than one core, yeah, mergesorting with multiple threads is a great option.