OS Assignment-4

Mergesort Comparison Report

After running the two codes (one using simple mergesort, and the other using forking in merge sort) on an input of 1000 randomly generated integers (check file.txt), the following performance chart was obtained.

Sr. No.	Normal Mergesort	Mergesort using fork	Mergesort using threading
1	0.006	0.270	0.032
2	0.009	0.187	0.031
3	0.010	0.236	0.047
4	0.005	0.182	0.033
5	0.007	0.218	0.035
6	0.012	0.319	0.037
7	0.011	0.198	0.047

Thus, as is obvious from the chart, simple merge sort is much faster than the one using forking and a little faster than the one using threading.

Reason:

1. Sequential Mergesort is faster than than the one using fork
   - When the left child accesses the left array, the array is loaded into the cache of the processor. Now when the right array is accessed (because of concurrent accesses), there is a cache miss since the cache is filled with left segment and the right segment is copied to the cache memory, thus overwriting the left segment. This process goes on and on and eventually it increases the time so much that the normal merge sort performs much better.
   - Though cache misses can be reduced by controlling the workflow of the code, they cannot be avoided completely!
2. Threaded Mergesort is slightly slower than the Sequential merge sort
   - Right now, we are using a base case of 5 for selection sort.
   - We are creating a large number of threads, each of which then does little work. To sort, say, 1000 integers, we create about 200 threads in total that spawn other threads and merge their results.

- The overhead from creating these threads somewhat outweighs the gain we get from parallel processing.

  - A solution for this would be to use sequential merge sort for a larger base case. Sequential merge sort is preferable for smaller size arrays. While sorting say, a million integers, using a base case of 1000 will speed things up a lot.