Allison Coopersmith & Diana Del Gaudio

Asst2: Spooky Searching

Implementation Our code is compiled using either the “make proc” or “make thread” command. To execute, the implementation of the driver accepts the array size and the number of processes/threads as an input, and the executable is run as:

```
./a.out arraySize thread/procSize
```

Our code was designed under the assumption that no more than 250 branches would ever be tested, as specified in the assignment instructions. Setting ulimit to be 250 prior to running the executable of the proc code solidifies this fact. A user can input a number of threads > 250; however, our program sets a hard limit of 250 in a conditional and will not allow for a thread/proc size >250 to be tested (that is, it will be restricted to 250 at the most.) We also will not allow an array of size 0, nor 0 threads/processes, as this program is using threads/procs to find the array index a target value is at. Finally, our target value is always 0 (the search function is always looking at 0) because at minimum, 0 will always be a value in the array (array of size 1 has a value of 0 in it, and array of size 20,000 will also have a value 0 in it because of the way our array is generated. Therefore, it will always be searched for, and we move its position at every iteration of the function in the array to guarantee a unique array every time it is searched.

Testing We tested our search functions by having arrays ranging from very small to very large (of size 10, 100, 1000, 10000, and 20000). For each individual workload, we tested branches in iterations of 5 from 1 to 150 (assuming the array was large enough to run that many processes, otherwise smaller branch sizes were used instead--for example, the array of size 10 ran 1, 5, and 10 branches). We used 150 as the upper limit of number of threads/processes in our graph. This allowed us to see how the processes and threads behaved on a large range of input sets, and also let us see what happened when there was a very small, medium, and large amount of branches--sometimes the number branches even increased all the way up to the size of the array. We tested our code on five separate occasions spanning four days, and used the averages in our graphs. Our findings can be seen in our results.pdf file, which highlights when threads are advantageous over processes and vice versa.

Notes We note that the times varied on days when the ilabs were lagging, more people were on a machine, and time of day (the latter also influenced by how many people were on a machine -- for example, at midnight, not as many people were on a machine compared to at five p.m.) Finally, our original assumption that threads would always be faster was not

necessarily correct; in some cases, processes ran faster. Array size and number of thread/proc size influence the runtimes, and it is apparent in certain cases, running multiple processes runs faster than running multiple threads, and vice versa.