Systems Programming
Assignment 2: Procs vs Threads
Stephen Petrides
Andrew Macneille
11/21/2016

A multithreaded and a multiprocessed multipart compression program.

INSTRUCTIONS

Compile with make command. A makefile is included.

$ ./compressT_LOLS ./file.txt <int>
$ ./compressR_LOLS ./file.txt <int>

If an incorrect number of parameters are run the program will output an error and exit.

SUMMARY

The multitreaded program will spawn threads for each part (compressT_LOLS) while the multiprocessed program will spawn child processes for each part (compressR_LOLS). In our program, all compression is done in threads and child processes. Each one reads its own input file and writes to its output file.

MULTITHREADED PROGRAM

A few tasks are completed to prepare for the thread creation. The chunk size (number of characters for a thread to compress) is determined from the uncompressed length divided by the number of parts. Any remainder is added to the first chunk. This, along with the starting position, is worked out in the init_chunk() function that is called be any threads are created.

Along with init_chunk(), other important initializations are done before any threads are created. All memory allocation is done before threads are created to prevent from the threads overwriting data or causing segmentation faults. This allows the threads to take the necessary input (chunk size, start position, input file name, output file name) and run the compression algorithm. If we allowed the threads to access the heap at the same time we would have needed to use locks and semaphores. By setting up our memory before creating threads, we avoided the use of those.

MULTIPROCESSED PROGRAM

The multi-processed program is structured similar to the multithreaded program. Similar tasks are completed before child processes are spawned. When spawned, they are given certain values that make the compression easier to complete (chunk size, start position, input filename, output filename). Memory allocation is not an issue in this case because each child process has its own virtual memory space. The challenge was making sure that all processes got the correct

information, would run, and were being waited on by their parent process. In our project all processes (and threads) and waited on (and joined).

TESTING

Our testing includes time tests and special case handling. The special case handling is included below.

To test the speed of the programs we ran the two executables with small and large txt files (50kb and 250kb) with varying parts values (ranging from 1 to 128). This is included in the timetest.sh file. The output of these test are in the timetests.txt file.

In the smaller file (50kb), both the multithreading and multiprocessing runs were the most efficient when broken into 2 or 4 parts. Multithreading dropped an average 0.002 seconds from 1 part to 2 parts and from 2 parts to 4 parts. Multiprocessing dropped 0.002 seconds from 1 part to 2 parts but began to increase after 2 parts.

In the larger file (250kb), the multithreading is most efficient when broken into 4 and 8 parts while the multiprocessing is most efficient when broken into 2 or 4 parts.

This demonstrates an important principle of concurrent programming: for the thread or process to save time, it must work hard enough that it is worth it to add contact switches to the kernel. In this case, the time it takes to spawn child processes and create threads is only worth the time saved from the compression of a few thousand characters. Therefore, as the file size increases, the more threads and processes should be spawned.

Furthermore, the time it takes to initialize a new process is greater than the time it takes to initialize a new thread.

SPECIAL CASE HANDLING

If there an uneven split of characters to parts, we put the entire remainder into the first file.
Non-alphabetical characters are skipped and not considered in the compressed string.
An error is reported and the program exits if the number of parts is less than one or if the number of parts.
An error is reported and the program exits if the number of parts exceeds the length of characters in the string.
An error is reported and the program exits if the parameters are not entered correctly.
An error is reported and the program exits if the input file cannot be opened (if it does not exit on the given path).
Error are reported if the output files cannot be opened.