Assignment 4: Multi-threaded Merge Sort

Due Monday by 11:59pm **Points** 105 **Submitting** a file upload
Available Jul 18 at 8am - Aug 11 at 11:59pm 25 days

Introduction

Write 2 variations of a C program called `uniqify` . The job of `uniqify` is to read a text file and output the unique words in the file, sorted in alphabetic order. The input is from `stdin` , and the output is to `stdout` . In one case, you will be using distinct input and output processes. In the other case, you will have a single process, with 2 threads: one thread for input, and one for output.

Learning Outcomes

- Describe what is mutual exclusion and why is it an important property to maintain when developing programs that may concurrently access shared resources (MLO 2)
- Describe the API you can use to create threads and wait for the termination of a thread (MLO 4)
- Describe what are condition variables and the API related to their use (MLO 4)

Instructions

he purposes of the below instructions, "task" will be used to denote either a thread or a process, depending on which variation you are implementing. If you do this properly, it will be a very simple switch from threads to processes (or the other direction) in your code.

Internally, the program would be organized into 3 tasks. A single task reads the input parsing the lines into words, another group of tasks does the sorting, and a single task suppresses duplicate words and writes the output. You should organize the processes so that every parent waits for its children.

You must use the system sort command (`/bin/sort`) with no arguments to do the actual sorting, and your program must arrange to start the processes and plumb the pipes (this means no popen). The number of sorting processes is a command line argument to the program, with the parser distributing the words round robin to the sorting processes. In both variations, `sort` should be its own process.

The I/O to and from the pipes should be done using the stream functions `fgets` and `fputs`, with `fdopen` for attaching to the pipes, and `fclose` for flushing the streams. After each sorting process is done, work must still be done to merge and uniquify the words prior to printing the list from the suppresser.

In this assignment words are all alphabetic and case insensitive, with the parser converting all alphabetic characters to lower case. Any non-alphabetic characters delimit words and are discarded.

Please provide timings based on the size of the input file. Ensure that you test it with multiple types and sizes of files, including word lists and free form prose. These timings should be plotted in a fashion that makes sense. I recommend using one of `matlab`, `gnuplot`, `R`, `matplotlib`, or `octave` (or some other tool you're familiar with) to generate the plot. Nearly all of these tools are accessible from `python` as well as their own interpreter environment. Please ensure to save the plot as a postscript file.

Some questions you should think on (and answer in your README file) is whether there are benefits to using threads vs processes (and vice versa). Which variation runs with higher performance? Did this surprise you?

For extra credit, provide output similar to `uniq -c`, where each word is output with its frequency. To receive full extra credit, you have to calculate this "on the fly" -- no storing all the words in memory.

Hints & Resources

- Designed properly, the threads and processes versions should look VERY similar, save for different function calls to start them.
- This is fundamentally an implementation of merge sort.
- `#ifdef` is equivalent to `#if defined`
- A thread startup function can be called without starting a thread (for instance, after a call to `fork`).
- Merging should happen dynamically -- you should build the sorted list from the disparate `sort` processes as you get the data from them.
- Command line arguments should be use to define the number of `sort` processes.

What to turn in?

- You can only use C for coding this assignment and you must use the gcc compiler.
- You must use the C11 standard on os1.
- You must use (at least) the flags `-std=c11 -Wall -Werror -g3 -O0` -- this means that all warnings **must** be eliminated in order to compile correctly.
- Variations should be handled within a single C file, using preprocessor macros to select between them. This means making use of `#ifdef/#elif/#else/#endif` constructs.
- Your assignment will be graded on os1.
- Submit a single zip file with all your code, which can be in as many different files as you want.
- This zip file must be named `youronid_program4.zip` where *youronid* should be replaced by your own ONID.
  - E.g., if chaudhrn was submitting the assignment, the file must be named `chaudhrn_program4.zip`.
- In the zip file, you must include a text file called `README.txt` that contains instructions on how to compile your code using gcc to create an executable file that must be named `msort` as well as the answers to the questions posed above.

- When you resubmit a file in Canvas, Canvas can attach a suffix to the file, e.g., the file name may become `chaudhrn_program4-1.zip`. Don't worry about this name change as no points will be deducted because of this.

Grading Criteria

- This assignment is worth 15% of your final grade.
- The process based version is worth 100 points
  - Getting sort to run is worth 40 points
  - Merging is worth 40 points
  - Sending words to the sort processes is worth 20 points
- The threads based version is worth 100 points
  - Getting sort to run is worth 40 points
  - Merging is worth 40 points
  - Sending words to the sort processes is worth 20 points
- Testing, plotting results, and answering the questions is worth 50 points

File Upload Google Doc

Upload a file, or choose a file you've already uploaded.

Upload File Use Webcam

+ Add Another File

Click here to find a file you've already uploaded

Comments...

Cancel Submit Assignment