COMP 206 Winter 2018 – Assignment 4

VERSION A – April 3rd, 2018

Objectives:

Gain experience programming multi-process software systems: using semaphores to coordinate a moderately parallel algorithm and experience with distributed software systems.

Hints and Resources

This assignment is very specific towards the semaphore interface we discussed in class during Lectures 16 and 17 (March 27 and 29). I recommend you check those carefully before starting as well as viewing the code on the Lectures GitHub repository. This example will be a good start for Question 2:

https://github.mcgill.ca/COMP206Winter2018/Lectures/blob/master/Lecture17-Coordination/synched_counting.c

Getting started with code:

The A4 repository is at: https://github.mcgill.ca/COMP206Winter2018/Assignment4. This time the code is just a starter for you to clone and develop on your own machine. You will not have to create a mirrored repository on GitHub for marking, but you can if you find this useful.

Read through the provided code. Notice that there are two main functions that you'll interact with:

- sort_single_process.c : used for Question 1 only. Nothing fancy, just helps you get started with the sort routine.
- sort_multi_process.c: used for Questions 2 and 3. This code uses fork to create 32 processes, one to sort the words starting with each letter of the alphabet, one to finalize the process (and 5 that do nothing)

These main files call to the routines in A4_sort_helpers.c. You will only edit that file during this assignment, but make sure you understand all other files present.

The code already has a Makefile. Build the provided code with "$ make". Run some of the tests to see the current functionality (broken, you need to fix it in each question):

- $ bin/sort_single_process test_inputs/words_short.txt
- $ bin/sort_multi_process test_inputs/words_short.txt

Handing In:

Only one file to hand-in again this time. Submit your version of "A4_sort_helpers.c" to My Courses. Your aren't allowed to change any other files while creating your solutions, but you are allowed to modify anything you desire within that file to meet the specifications.

Deadlines:

One deadline: April 13th at 11:59pm.

Intro: Parallel Processing

One of the main reasons we want to break computation up into multiple processes is to take advantages of the potential speed-ups that come with doing multiple operations simultaneously on modern multi-threaded CPUs. We'll look at sorting a text file in a single process and then sorting it with 26 independent processes. Although we're looking at very simple algørithms that are far from the state-of-the-art in single or multi-threaded sorting, we can already see 100's of times speed-up with some simple tricks… if you can get the semaphores right!

Question 1: Single Threaded Sort (30 marks)

Complete one function, sort_words(), within the A4_sørt_helpers.c file. This function is meant to sort the words that have been loaded into the text_array. You can implement any sort function of your choice, but a very simple insertion sort is just fine. Remember that strcmp(a,b) returns <0 if a comes before b alphabetically (0 if they are equal, and >0 if b comes first).

Once you've completed your function, text_array should be sorted correctly. The program should output words sorted in dictionary order to standard output. You can confirm this by using the tests that are in the Makefile:

"$ make test_single_short"
"$ make test_single_long"
When everything is OK, you should not see any error printed.

Question 2: Synchronize the Processes (50 marks)

Add semaphore-based synchronization within 3 functions, initialize(), process_by_leter(), and finalize() such that the 27 active processes launched in sort_multi_process.c execute in the correct alphabetical order. Tips:

- Consider the "notify" semaphore pattern, where only the "a" process starts with a 1-value semaphore and the remainder start with a 0-value (so they block).
- Have each letter "post" on the semaphore for the next letter once its own operations are complete, and have the "z" process "post" on the semaphore that controls finalize().
- Ensure that you use descriptive semaphore names (not just "a", "b" etc), since semaphores can conflict with those of other users and with previous runs of your program.
- Consider calling "sem_unlink(char* sem_name)" just before your "sem_open". This will ensure you start with a fresh value.

When the right order is achieved, the output will be:

This process will sort the letter a.
This process will sort the letter b.
This process will sort the letter c.
… (d through w removed for space) …
This process will sort the letter x.
This process will sort the letter y.
This process will sort the letter z.
Sorting complete!

Question 3: Put it together, multi-process sort (20 marks)

As a final step, complete the sort functionality to make sort_multi_process output the correctly sorted text. You must follow several rules:

- The printing to standard output must only happen inside finalize(). This is because finalize is run by the original parent process and our tests only consider its standard out. The others can also be seen on the console, but they wont be caught in the automated checking.
- In finalize(), you cannot read from the input file or do any actual sorting. Both of those operations must be done by each letter process.

I recommend you implement a temporary file that each letter process writes its sorted results into (using append). The finalize process can then read the file and print the contents.

When complete, the correct dictionary sorted output will display on standard output. You can confirm this works by using the Makefile tests:

"$ make test_multi_short"
"$ make test_multi_long"