CIS 505: Software Systems

Spring 2017

Assignment 1: Processes and threads

Due February 3, 2017, at 10:00pm EST

1 Overview

For this assignment, you will build a program that can sort numbers while taking advantage of concurrency. Sorting itself is not rocket science – you were probably exposed to a range of different sorting algorithms in your introductory classes. The `real' goal is to familiarize yourself with some basic UNIX primitives (processes, threads, pipes, basic I/O), and to get some hands-on experience with concurrency and its impact on application performance.

We have put together a small amount of code to get you started. This code has been checked into your Git repository. If you have already created a local clone of your repository as described in the HW0 handout, you can download the HW1 code simply by opening a terminal, changing to the directory that contains your local copy of the repository, and running `git pull`, like this:

```
cd ~/git/
git pull
```

The HW1 directory will contain a basic Makefile, a README file, a placeholder for your sorting program (`mysort.cc`), and a small tool that can generate example inputs (`makeinput.cc`). We strongly recommend that you check in your code frequently using `git commit` and `git push`; that way, if your code breaks, you can easily go back to a previous version. Also, this will give you a backup of your code in case something happens to your VM.

2 The sorting program

As a first step, you will build a version of the program that uses a (configurable) number of processes for concurrency. Suppose N is the number of processes. Then the program should read the input data from a number of files, break it into N pieces of roughly equal size, fork N subprocesses, and send each piece to a different subprocess. The subprocesses should then sort their pieces and send the results back to the main process, which should merge then into a single sorted sequence and then output the result to `stdout`.

The communication between the main process and its subprocesses should be implemented using a pair of pipes (an 'up' pipe and a 'down' pipe) for each subprocess. The main process should hold the write end of the down pipe and the read end of the up pipe; the other ends should be held by the subprocess. Once the main process has forked a subprocess, it should write the numbers that it wants that subprocess to sort to the down pipe; the subprocess should read the numbers from this pipe, sort them, and write the sorted numbers to the up pipe; the main process should then read them from that pipe. The easiest way to write and read the numbers is using `fprintf()` and `fgets()`; however, keep in mind that the `pipe()` system

call returns a pair of file handles and not `FILE` objects, so you will have to use the `fdopen()` function to convert them first.

To make things simple, we will make two simplifying assumptions: 1) The input consists of text files that contain signed 64-bit numbers, each on a separate line; and 2) all the input files together contain at most one million numbers.

The program must accept the names of several input files on the command line, as well as an optional '-n' parameter that specifies the number N of subprocesses that the program should use. (By default, the program should use N=4 subprocesses.) For instance, the command line could look like this:

```
./mysort -n 3 foo.txt bar.txt
```

If the user sets N=1, the program should sort the numbers directly and not fork any subprocesses. If the user sets N>1, the program must use *true concurrency*, that is, all the subprocesses must be able to work in parallel if your machine has at least N CPU cores. If the program is invoked without any command-line arguments, it should output the full name and SEAS login of its author to `stderr`. (Please consider using the `getopt()` function for parsing the command-line arguments. If you are unfamiliar with this function, please have a look at https://www.gnu.org/software/libc/manual/html_node/Getopt.html). If anything goes wrong, the program should output an informative error message to `stderr` and terminate with an exit code other than zero. The program should only return zero as an exit code if it has output the sorted numbers.

In order to see something interesting in the benchmarking phase (see below), your program should use the Bubblesort algorithm (!) to sort the numbers. (Yes, this is slow, but that is the idea!) If you are not sure how to merge several sorted sequences, please have a look at the description of Mergesort in any good textbook.

2.1 Testing and debugging

You should test your solution carefully. One good way to do this is to use the `makeinput` tool, which comes with the HW1 framework code, to make several test inputs of different sizes, and to then run these inputs through the program with several different settings of N. To find out what the correct output is, you can use the UNIX sort command. For instance, you could try the following:

```
./mysort -n 4 input1 input2 >myoutput
cat input1 input2 | sort -n >goodoutput
diff myoutput goodoutput
```

If the diff command returns nothing (i.e., there are no differences), this is a good sign.

If your program crashes or otherwise misbehaves, please consider using the gdb debugger; there are lots of good tutorials available on the web. Also, you may want to add a flag (say, -v) that enables a special verbose mode, in which your program outputs additional debug information. (By default, your program should *not* output any debug information!)

2.2 Adding thread support

Once your process-based solution works, the next step is to add thread support. You should add a new command-line option (`-t`). If the user supplies this option, your program should use pthreads instead of processes. As before, the process should create N threads, which should each sort a roughly equal portion of the data in parallel and then report back to the main thread, which should merge the portions into a single sorted sequence after all the threads have finished. If the `-t` option is not given on the command line, your program should continue to work as described above (i.e., fork subprocesses to do the sorting).

Please keep in mind that, unlike the subprocesses that `fork()` creates, the pthreads share a single address space with the original thread that came with the process. Thus, if one of the threads makes changes to a shared data structure, these changes will be visible to the other threads. If you need to synchronize the threads, please use the proper primitives from the pthreads library – do not 'roll your own' synchronization primitives.

3 Evaluation

Once you have completed your implementation, the final step is to do a small quantitative evaluation. First, use the `makeinput` tool to create data sets of various different sizes, and find one that takes roughly ten seconds to sort with no concurrency (`-n 1`). You can measure the completion time using the UNIX `time` command, like this:

```
time ./mysort -n 1 in1 >/dev/null

real 0m10.441s
user 0m19.347s
sys  0m0.055s
```

Next, vary the parameter -n from 1 to 20 and measure the completion times with and without the -t option. (You will notice that the values reported as 'user' will start to become larger than the values reported as 'real'; this is because the latter measures wall-clock time, whereas the former measures total time spent in user level on *any* core. You should write down the 'real' values, not the 'user' values!) If the values do not start dropping substantially for N>1, you either have a bug in your code, or you are measuring on a single-core machine. If the latter, please measure on a machine with multiple cores.

Finally, draw a graph that has N on the horizontal axis and the measured 'real' times on the vertical axis. The graph should have two lines, one for threads (`-t`) and one for processes. Please describe and explain 1) any nontrivial differences between the two lines; 2) any major trend within each line. You should submit the graph as a file called `graph.jpg` or `graph.pdf`, and a text file with your descriptions and explanations as `description.txt`. The file `description.txt` should be short and concise; a few sentences will be sufficient.

4 Submitting your solution

Before you submit your solution, please make sure that:

- ☐ Your solution compiles properly.
- ☐ Your code contains a reasonable amount of useful documentation (required for style points).
- ☐ You have completed *all* the fields in the `README` file.
- ☐ You have checked your final code into the Git repository *before* you submit.
- ☐ You are submitting a `.zip` file (e.g., one created by running `make pack` from the `HW1` folder)
- ☐ Your `.zip` file contains all of the following:
  - ☐ all the files needed to compile your solution (especially all `.cc` files!);
  - ☐ a working Makefile;
  - ☐ the files `graph.jpg` (or `graph.pdf`) and `description.txt`; and
  - ☐ the `README` file, with all fields completed
- ☐ Your `.zip` file is smaller than 250kB. Please do not submit binaries or input files!

Once you have your `.zip` file ready, please open https://alliance.seas.upenn.edu/~cis505/cgi-bin/submit.php in your browser, authenticate with your PennKey, select HW1 from the dropdown box, and then upload the `.zip` file. This page also allows you to request an extension if you need one.

5 Extra Credit

5.1 Arbitrary input (+7%)

Extend your sorting program to sort arbitrary lines of text, and text with more than one million lines. You may assume that the input fits into memory – there is no need to implement external sorting – but your solution must not require an upper bound on the number of characters per line. To maintain compatibility with the main assignment, your solution should still use numeric sorting by default, and only switch to lexicographic sorting when a special `-L` option is given on the command line. For instance, if the input is a file with four lines: "1", "10", "2", "11", then the output should be "1", "2", "10", "11", by default, and "1", "10", "11", "2" when the `-L` option is given.

5.2 Benchmarking (+3%)

Extend your program to handle more numbers (say, 100 million), and then benchmark runs with inputs of different sizes. Produce a graph that has the input size on the horizontal axis and the runtime on the vertical axis, and plot at least 8 lines for different configurations (using both processes and threads, as well as different settings for `N`). Your graph should contain at least 10 data points per line. Describe and explain any nontrivial differences between the lines. You should submit the graph as a file called `graph2.jpg` or `graph2.pdf`, and a text file with your descriptions and explanations as `description2.txt`. The file `description2.txt` should be short and concise; a few sentences will be sufficient.

5.3 Shared memory (+10%)

Extend your process-based solution to use shared memory for the communication between the processes. You should add a new command-line option (`-s`). If the user supplies this option, your program should use shared memory instead of pipes. We did not cover shared memory in class, but you can find many tutorials available on the web; for example, a good introduction with examples to using shared memory in C can be found at https://users.cs.cf.ac.uk/Dave.Marshall/C/node27.

For this extra-credit task, you may make the same simplifying assumptions as described in the main part of the assignment, i.e., 1) the input consists of text files that contain signed 64-bit numbers, each on a separate line; and 2) all the input files together contain at most one million numbers.