Rutgers ECE 434, Spring 2021 Prof. Maria Striki

Project 2: LINUX Inter Process Communication and Signals

Issue Date: Sat 04-02-2021, Due Date: Thu April 22, 2021, 9.00pm

Total Points (45 points)

Problem 1: Distinct teams of threads sorting & signaling (45 pts)

Write a C program that generates an output with the following desired behavior:
From main function we spawn 4 "teams" of Pthreads (their identities will be held in an array T of elements). The teams will be handling the following signals that may originate either from the keyboard, or from the main function and from being sent from either of the threads:

```
SIGINT, SIGABRT, SIGILL, SIGCHLD, SIGSEGV, SIGFPE, SIGHUP, SIGTSTP.
```

Action 1 (10 pts): Each distinct team of Pthread is spawned with the objective to catch **ONLY THREE** of the above signals (you choose which three will be handled by each team) and handle them with customized handlers. Also, every team should block or ignore the rest.
When one Pthread team (or the main function) catches one of the signals you have selected for them to catch, they should handle the signal with a customized handler. The handler at the minimum will output the signal number and identity of the thread which handles the signal (thread or main).
The main function is set up to ignore the signals until the three threads are created. After the three threads join, then the main function restores default operation for all the modified handlers.
In your code you may use either function `signal` or `sigaction` for setting up the signal handlers.

Action 2 (15 pts): Moreover, each distinct team of Pthreads will also sort a subarray of integers. The main array of O(1,000) – O(1,000,000) elements will be created at main with the use of any random function. And then, each distinct team (out of the 5 teams in total) will be passed a subset of the array to sort (i.e., 1/4 of consecutive array elements to sort).

How many threads are there in a distinct team of threads? You choose. It can be anything from 2 threads to 1,000 threads. You are asked however to run your experiments for 3 distinct numbers of threads in every team: for example: 4, 100, 1,000. It is best that the array of elements has more than O(1000) elements in order for your experiments to make sense.

What sorting algorithm each distinct team will use? Here every group has the freedom to select a different algorithm, a different strategy. The credit is equal for any of the cases you select. Decide based on your familiarity, availability, schedule.

Case 1) Sequentially sort the numbers provided to each distinct team, using one of the well known sorting algorithms: e.g., quicksort, mergesort. Then, one representative thread of their team will sort the numbers of the subarray and produce timing results (or simply ordering results if you do not wish to use timers) once the computation completes. You will do the same for every team of threads and record/report when every team completes.

Case 2) For groups that want to take this project further, you will select a sorting strategy that lends itself to parallel implementation and you will have all threads in one distinct team implement this sorting algorithm in parallel, in addition to sequential execution. For example, you may select all your distinct

teams to run: sequential and parallel quicksort. Or you may select all your distinct teams to run: sequential or parallel mergesort, or you may select to run: sequential and parallel radix sort.

Case 3) For groups who are up to something even more fun to experiment with, you can allow a number of your teams to utilize different sorting algorithms: one team can use sequential and parallel quicksort, the other team will use sequential and parallel mergesort, another team will use sequential and parallel radix sort, another team will use sequential and parallel bubble-sort, or any algorithm you prefer.

Your group must choose to implement either: only Case 1 or only Case 2 or only Case 3.
However, for any of the cases you select you will run experiments with three selections for the number of threads (for those who are implementing also the parallel versions only) and three selections for the size of the original array (for all cases: those implementing only sequential sorting and those implementing parallel sorting). Do not experiment with anything additional for either case. You should document the results of your comparison in your report. What do you observe? Which case of fine-tuning provides the optimal results? Why do you think this is so?

For every parameters you fine tune, you are asked to record which team of threads completes first, which second, which third, which fourth. In order to accomplish that you may use timing functions if you so wish, but you do not have to. You can simply have the team which complete first populate accordingly a data array. For example: if a team completes second, it scans this data array, finds the first NULL element and registers itself there. When all teams are done, we will have a data array with the correct orderings of completion.

Action 3 (10 pts): Take the ordering result from one of your fine-tuning experiments only and implement ONLY one of the two following strategies:

Simple Strategy: The Pthread team that completes first will send "terminating" signals to all the rest of the teams (not to the main thread though). Care must be taken for the proper signals to be send to which threads? How many times? Remember that every team handles/blocks/ignores different sets of signals according to the scenario you selected. Investigate if you need to make any changes to your handlers to achieve correct operation.
The fastest team will collect the status of the slowest team and will output that, and then it will TERMINATE the slowest team with the proper signal. It will do the same for the remaining two teams, terminating first the slowest among them, and then the next. Once this is done, the fastest team will terminate itself with the proper signal.

Modified Strategy: **If your group is up for some fun with signals, then instead of the Simple Strategy you may implement the Modified Strategy for the same amount of credit.** The fastest team, instead of a terminating signal, it will send the other three teams some type of stop signals. Again you have to check if these can be handled and investigate what changes you need to make for these signals to be handled. The fastest team sends a message to the next fastest team which has been stopped to now resume. The newly resumed team now terminates the fastest process and collects and outputs its status. Then, the resumed process identifies the next faster team, resumes that team. The newly resumed team also resumes the next fastest team. The next fastest team that is now resumed sends a terminating signal to the team that resumed it, collects and outputs its status. We keep repeating the process, until the slowest team is the last remaining team which will also terminate itself. The main thread will collect its status and output it.

Assume that signal does not automatically re-installs itself. Even though in most recent LINUX version the signal function does re-install itself, I would like you to distinguish those two cases and explain what happens next (in Part 2) for both cases: 1) signal does not re-install itself, 2) signal does re-install itself. You may run the one case experimentally, and for the second case you may manually revert to the old handler using programming or attempt to tackle this question theoretically.

Action 4 (10 pts): Now ignore Action 2 and Action 3 in your code, only focus on Action 1: Make the following experiments and answer the questions:

Q1: (5 pts) What do you observe when the process receives one of the following signals externally (from terminal) during execution?

`SIGINT, SIGABRT, SIGILL, SIGCHLD, SIGSEGV, SIGFPE, SIGHUP, SIGTSTP.`

Briefly describe each case. What does your results depend on? Please justify your answer.

Q2: (5 pts) What will be observed when one thread sends one of the signals separately to a specific thread id (to one thread at a time)? Study each case separately.

Briefly describe each case. What does your results depend on? Please justify your answer.

.

Remark: Experiment with each signal at a time, record the messages you get on the screen and describe the behavior of your program. Does your program completely stop at all times, for all signals? Does one thread only get suspended or stopped or all the threads do, or the main process does? Report your findings. One way to check on the status of your threads (if your process is still alive) is to ask the threads send printouts with their id and status. Those that do are still alive and unblocked. Of course, bear in mind that some of the signals you sent completely terminate your program. In this case, re-run the program, and send one by one the rest of the signals.

Please provide detailed code for this problem (you have ample degree of freedom) and a thorough report with your results and your justifications of the results.

Solution:

What to turn in:

- C files for each problem
- A makefile in order to run your programs.
- Input text file (your test case)
- Output text file (for your test case), or printscreens…
- **Report**: Explain design decisions and interpret results. Also, please consider providing a very detailed report, as along with your C file deliverables, it corresponds to a substantial portion of your grade.

Logistics:

- You are expected to work on this project using LINUX OS
- For those that do not have access to LINUX in their laptop, you may use one of the solutions posted online regarding how to get access to a LINUX platform.
- Make ONE submission per group. In this submission provide a table of contribution for each member that worked on this project.
- Do not collaborate with other groups. Groups that have copied from each other will BOTH get zero points for this project (as a warning) no matter which copied from another, and will also incur more substantial consequences.

APPENDIX

Useful Links:

https://www.gnu.org/software/libc/manual/html_node/Generating-Signals.html#Generating-Signals

How to write a MakeFile (Example):

```
$ cat Makefile
# a simple Makefile

CC = gcc
CFLAGS = -Wall -O2

all: fork-example

fork-example: fork-example.o proc-common.o
 <Tab> $(CC) -o fork-example fork-example.o proc-common.o

proc-common.o: proc-common.c proc-common.h
 <Tab> $(CC) $(CFLAGS) -o proc-common.o -c proc-common.c

fork-example.o: fork-example.c proc-common.h
 <Tab> $(CC) $(CFLAGS) -o fork-example.o -c fork-example.c

clean:
 <Tab> rm -f fork-example proc-common.o fork-example.o

$ make
gcc -Wall -O2 -o fork-example.o -c fork-example.c
gcc -Wall -O2 -o proc-common.o -c proc-common.c
gcc -o fork-example fork-example.o proc-common.o
```