CSCI 5103 – Assignment 4

Released Oct. 16th, 2018

Due Oct. 29th, 2018

Can be completed in group size of at most 2 students

Submission

Submit a `zip` or `tar` archive containing your solutions to this assignment. You are allowed to change or modify your submission until the due date, so submit early and often. **Verify that all of your files are in the submission**. Failure to submit the correct files will result in a score of zero for all missing parts.

- Only **one** person needs to submit. If both partners submit, then the submission with the latest timestamp will be graded.
- Include your names and x500s at the top of each submitted file.
- Please make sure your submission contains all of the necessary files to **compile and run** the programming portion of the assignment. Your program **must compile and run** correctly on a CSE-Lab Linux machine as grading will be performed on these machines.
  - Include a `README` to list any known issues and or assumptions about your programs.

Grace Day usage

If you are using your grace days, indicate how many you are using **at the top of each submitted file** in your submission, e.g. `.pdf` ,`.c`. The following is a sufficient enough indication of grace days being used: `Using <#> grace days`

Late submissions which omit the indication will be graded with the following penalties:

1-day late	10% penalty
2-days late	25% penalty
3-days late	50% penalty
$\geq$ 4-days late	No credit

Objectives

The objectives of this assignment are the following:

- Acquire familiarity with using POSIX thread management primitives
- Understand and use LINUX functions for creating shared memory between two LINUX processes
- How to write POSIX thread programs, where threads execute in different processes and communicate through shared memory
- Read the manual pages for the LINUX primitives for creating shared memory between two LINUX processes (`shmget`, `shmat`)
  - Please study and execute the sample programs posted on the course webpage to create shared memory between two LINUX processes using these functions.

For sample programs on how to use the POSIX thread library and its shared memory management, look at the *Examples* section of the course webpage.

The rest of this page is intentionally left blank

Problem Statement

Consider a system comprising of **three** producer processes and **three** consumer processes depositing and consuming "colors" from a shared buffer of length $N > 0$. Each producer is assigned a specific color to deposit into the buffer:

- `producer_red` will only produce `"red"`
- `producer_green` will only produce `"green"`
- `producer_blue` will only produce `"blue"`

Similarly, each consumer is assigned a specific color to consume from the buffer:

- `consumer_red` will only consume `"red"`
- `consumer_green` will only consume `"green"`
- `consumer_blue` will only consume `"blue"`

Associated with each item a producer deposits into the buffer is a timestamp, in microseconds, of when that item was deposited into the buffer; you may use the `gettimeofday()function`. Associated with each item a consumer consumes from the buffer is the timestamp at which the item was consumed.

After a producer deposits an item in to the buffer, it will write that item along with its timestamp into a file called `producer_<color>.log`, e.g. `producer_red.log`. Similarly, after a consumer consumes an item, it too will write the item it consumed to a log file called `consumer_<color>.log`, e.g. `consumer_red.log`.

You will solve this problem in two different ways, one using POSIX threads and the other using LINUX processes.

Requirements

1. Items in the buffer must be ordered as such: `red, green, blue, red, green, blue`....
2. Producers will only produce 1000 of their assigned colors and log their deposits into the buffer in their respective log files. Each line of the log file **must** be in the following format without quotation marks: `"<color> <deposit timestamp>"`, e.g. `"red 12345"`
3. Consumers will only consume their assigned colors and log their consumptions from the buffer into their respective log files. Each line of the log file **must** be in the following format without quotation marks: `"<color> <deposit timestamp> <consumed timestamp>"`, e.g. `"red 12345 23456"`
4. The size of the buffer must be configurable through arguments passed to your program (more on this later).
5. Upon termination of your programs, the six log files with their respective contents as described in the Problem Statement. The output files should be named as such and the contents should adhere to the format described above.

Problem 1 (POSIX Threads (50 points))

Solve the problem using **one LINUX process** containing **six POSIX threads**, three of which are assigned to the producers and the other three assigned to the consumers, each of which then are assigned to their respective colors.

Requirements

1. All of your code needed to run your program should be placed in a file named `prodcons.c`
2. A `makefile` to compile your code into an executable called `prodcons`
3. Your executable must be able to take in one argument to configure the size of the buffer. You may assume that the size will be greater than 0.
4. The program must terminate gracefully, i.e. it should wait until all threads complete before terminating.

Problem 2 (LINUX Processes (50 points))

Solve the problem using **six LINUX processes**, one for each of the three producers and one for each of the three consumers. All of these processes should run on the same machine and will therefore use a shared memory space to maintain the bounded buffer. In this shared memory, you will also have to maintain synchronization variables such as mutexes, counters, condition variables, etc.

Requirements

1. All of your code for the producers should go in a file named `producer.c`
2. All of your code for the consumers should go in a file named `consumer.c`
3. You must have a file called `main.c` that is responsible for forking all six processes and having them execute their corresponding responsibilities.
4. A `makefile` to compile your code into three executables, `main, producer,` and `consumer`
5. Your `main` executable must be able to take in one argument to configure the size of the buffer. You may assume that the size will be greater than 0.
6. The program must terminate gracefully, i.e. it should wait until all processes complete and **deallocate** the shared memory before terminating.

Useful Tips

1. Use `shmget` to create a shared memory
2. Anything that needs to be shared across processes should be placed in the shared memory. If you need more than one element shared across processes, consider using a `struct`
3. Shared mutexes and condition variables should be initialized using the `PTHREAD_PROCESS_SHARED` flag.
4. Look into `shmdt` and `shmctl` to detach and deallocate shared memory.
5. Look into `execl` to see how to execute files via a running C program.

 Continued on next page ⟶

Grading Criteria

The following is the grading criteria for each of the two problems.

1. Required files are named accordingly **[3 points]**
2. Upon termination, six log files are created with their contents in the format specified the Problem Statement. Moreover, the log files are named accordingly **[2 points]**
3. Documentation of non-trivial pieces of code **[5 points]**
4. Thread/Process creation is correct **[3 points]**
5. Programs take in a single argument defining the size of the buffer and a buffer of such size is created. **[2 points]**
6. Program terminates gracefully as defined above **[5 points]**
7. Overall correctness of program. This includes but not limited to, proper synchronization, ordering of elements in the buffer, correctness of the contents in each file, etc. **[30 points]**
   (a) Partial credit will be given accordingly.