CS342 Operating Systems – Fall 2018
Project 2: Task Communication and Synchronization

Assigned: Oct 27, 2018
Due date: Nov 12, 2018, **23:55** (Moodle)

You will do this project in groups of *two*. You have to program in C and Linux. You are recommended to use the following distribution of Linux: **Ubuntu 16.04 – 64 bit**.

Part A: Processes [35 points]
Objective: Practice process IPC, shared memory, synchronization.

In this part, you will develop the same application in project 1, but this time you will use shared memory, instead of intermediate files, to pass information from child processes to the parent process. Hence, again you will develop a **multi-process** application that will generate a value *histogram* for the values sitting in a set of input ascii text files, one value per line. Values can be an integers or real numbers. The program will be called **syn_phistogram** and will take the following parameters:

syn_phistogram *minvalue maxvalue bincount N file1 ... fileN outfile*

Here, *minvalue* is the minimum value that exists in the given set of input files, and *maxvalue* is the maximum value. *bincount* is the number of bins in the histogram. Let *w* denote bin-width. Then w = (*maxvalue* - *minvalue*) / *bincount*. The first bin will give the count of values in range [*minvalue*, *minvalue*+*w*); the second bin will give the count of values in range [*minvalue*+*w*, *minvalue*+*2w*); and so on. The last bin will be inclusive [] in both ends. *N* is the number of input files. *file1* … *fileN* are the names of these input files. *outfile* is the output file. An example invocation of the program can be like the following:

syn_phistogram 0 500000 20 3 f1.txt f2.txt f3.txt out.txt

As in project 1, your program will create another child process for each input file to generate a histogram for the values in that input file. Hence there will be *N* child processes working concurrently on the *N* input files. Each child will generate a histogram for the values in its input file. The generated histogram will be put into a *shared memory* this time, instead of outputting to an intermediate file. There will be one shared memory region that is shared between parent and all children. The shared memory segment will be *just large enough* to hold *one* histogram. Each child will use the same shared memory segment. When a child has the histogram ready, it will put the histogram into shared memory, and the parent will receive the histogram from the shared memory. In this way the histogram content will be passed from child to the parent. Parent will process the histogram (merge it to its histogram or copy it to its own memory). Then, another child can use the shared memory, When another child has finished preparing its histogram, that child will put the content into the shared memory (and overwrite the previous content in the shared memory). Then the parent can receive the content form shared memory again and process it. In this way, each child will prepare a histogram and will pass the content to the parent. Parent will

collect those and will build a single combined histogram. Then the parent will print out the combined histogram into the output file.

Parent and children must be synchronized in accessing (reading from or writing into) the shared memory. For example, while a child is copying its histogram into shared memory (i.e., accessing and writing into shared memory), another child should not access and write into the shared memory at the time. Similarly, while a child is copying its histogram into shared memory, the parent should not access and read the shared memory at the same time. After copying of histogram into shared memory is finished, the parent can access the shared memory and read the content. You should also not allow a child to overwrite the histogram of another child before parent has received the histogram. That means, after a child *X* has put its histogram into shared memory, another child *Y* can not write into shared memory, until the parent has retrieved the histogram of child *X*.

You will use POSIX semaphores for synchronization. Type "man sem_overview" at commad line of Linux to get information about POSIX semaphores. You can also find a lot of information in Internet about POSIX semaphores and their usage.

You will use POSIX shared memory as IPC among processes. Type "man shm_overview" at commad line of Linux to get information about POSIX shared memory interface. You can also find a lot of information in Internet about POSIX shared memory and its usage.

The output file will contain the combined histogram. Each output line will contain information about a separate bin in the following format: *binnumber*: *count*. Bin numbers will start at 1. For example, if the are 10 bins, the following can be a possible output.

```
1: 30
2: 50
3: 60
4: 40
5: 30
6: 20
7: 15
8: 10
9: 5
10: 3
```

Part B: Threads [35 points]

Objective: Practice developing multi-threaded applications, sharing information among threads, and synchronization.

In this part, you will develop a similar application as in part A. But this time, for each input file, there will be a separate worker thread reading the input file. This time, a worker thread will not build a histogram, but will just pass the values that it reads from its input file to the main thread. The main thread will build a combined histogram. To pass values form worker threads to the main thread, a single shared *linked list* will be used. That means in your program, you will define a linked list as a global variable (structure) to be accessed by all threads (worker threads and main thread). Initially, the list is empty.

A worker thread will read the values from its input file in batches of *B* values each. When a worker thread has retrieved *B* values from its input file, the worker thread will add (i.e., append) these values into the linked list. Then, the worker thread

will read another batch of *B* values from its input file, and then add them again. A worker thread will continue reading and passing values like this until the file is completely read.

Meanwhile, the main thread will receive the values from the linked list in batches of *B* values each and will generate the combined histogram. All worker threads and the main thread will run concurrently. After values are received and the histogram is build, it will be written to the output file.

The program will be called **syn_thistogram** and will be invoked with the following parameters:

syn_thistogram *minvalue maxvalue bincount N file1 … fileN outfile B*

All parameters are the same with **syn_phistogram** program except we have now another parameter *B*. This is the batch size. It can be a value between 1 and 100. As said, a thread will collect *B* values from its input file before appending them to the linked list. If *B* is 1, values are read and appended one by one. If *B* is bigger than 1, The last batch may contain less than *B* values.

You will use POSIX pthreads *mutex* and *condition variables* to synchronize the threads. That means, one at a time the threads should access the shared linked list. For example, while a thread is appending a batch of *B* values into the linked list, another thread can not concurrently append, or the main thread can not concurrently retrieve. After the addition of *B* number of values (a batch) is finished, then another thread can add its batch of values, or the main thread can retrieve a batch of values.

An example invocation can be as follows:

syn_thistogram 100 40000 6 3 file1.txt file2.txt file3.txt file4.txt o.txt 50

Part C: Experiments [20 points]
Objective: Practice designing and conducting experiments and applying knowledge and skills acquired in the Probability and Statistics course.

Do some timing experiments to measure the running time (turnaround time) of the **syn_thistogram** program for various values of *B*. You can use various input sizes. At the end, decide whether batch size affects the running time of the application or not. If you find that batch size affects, try to find out and discuss the relationship.

Submission

Put all your files into a project directory named with your ID (one of the IDs of team members), tar the directory (using **tar xvf**), zip it (using **gzip**) and upload it to Moodle. For example, a student with ID 20140013 will create a directory named 20140013, will put the files there, tar and gzip the directory and upload the file. The uploaded file will be 20140013.tar.gz. Include a **README.txt** file as part of your upload. It will have the names and IDs of group members, at least. Include also a **Makefile** to compile your programs. We want to type just **make** and obtain the executables. Do not forget to put your report (PDF form) into your project directory.

Additional Information and Clarifications

- POSIX thread synchronization functions to solve critical section problem are pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock. To solve other synchronization issues you may use condition variables with the following functions: pthread_cond_init, pthread_cond_wait, pthread_cond_signal, pthread_cond_broadcast.