CSE 5031 Operating Systems
2019/20 Fall Term

Project: Bonus #2

Topic: Multiple Producers – Single Consumer Paradigm

Date: 16 - 25.12.2019

Objectives:

- to implement multiple producers - consumer paradigm with **POSIX Pthreads**
- to coordinate actions of threads of execution with **POSIX Semaphores**

References:

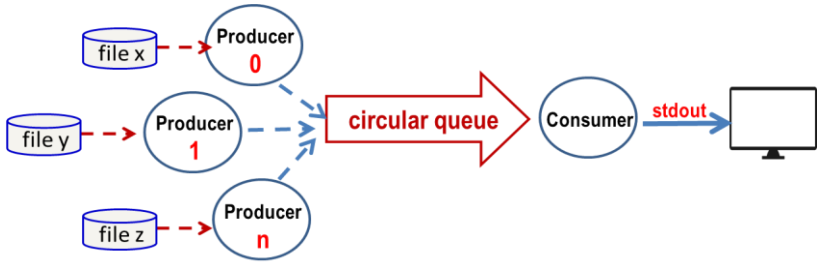
- **Lawrence Livermore National Laboratory Computing Center Pthreads tutorial portal,** https://computing.llnl.gov/tutorials/pthreads/#Pthreads
- **Linux System Programming 2d ed., Robert Love, O'Reilly 2013** (course web site, or http://pdf-ebooks-for-free.blogspot.com.tr/2015/01/oreilly-linux-system-programming.html
- **The GNU C Library Reference Manual** (course web site, or http://www.gnu.org/software/libc/manual/pdf/libc.pdf)

Section I. Project Definition

I.1 Problem Statement

The project aims at implementing **multiple producers – single consumer** scenario depicted here after.

- ✓ **Producers** read records from their respective file; process and store them in a **shared circular queue**, along with a tag that identifies the producer process and/or the file they belong to.
- ✓ The **consumer** retrieves the records from the **queue**, processes them to generate statistics, and displays the results when all the **producers** have finished their reading and the queue is empty.



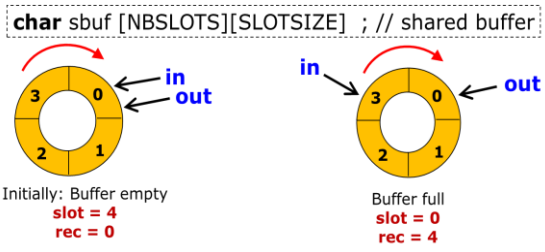
I.2 Implementation Constraints

The project requires the use of:

- ➢ **POSIX Pthreads** to model produces and consumer threads of executions;
- ➢ **POSIX semaphores** to synchronize thread of executions, and implementation of critical sections.

Critical Section Issue

Lecture slides and reference programs analyzed in the lecture hours provide you with several examples of **single producer – single consumer** scenarios implemented with **POSIX** tools. In these scenarios the producer and the consumer are not in race conditions on the use of the subscripts "**in**" and "**out**" while **enqueuing** or **dequeuing** a record to/from the shared buffer, as shown on the drawings at the right hand side.



In this project, **multiple producers** will be contending for the use of an input slot; thus they will be racing to acquire their own subscript "**in**". You have to implement the critical section that will prevent producers from acquiring the same input slot and incrementing the shared subscript at the same time!

Tagging the Records

Producers read their input from different files and enqueue their records in random order. As such, the **consumer** cannot tell which record belongs to which file or processed and stored by a given **producer**! An obvious solution is to prefix each record by the **id** of its **producer** e.g. **0,1,2 …n**; or the identifier of the file it belongs to (how?).

However, prefixing records alters their size and the resulting record may not fit in the fixed size slots of the queue. A feasible alternative is the use of an integer array having the same number of slots as the shared buffer, and storing the **id** of its **producer** e.g. **0,1,2 …n** along with the record. The C code here after provides an example of how tagging can be implemented.

```
char  buf [NBSLOTS] [SLOTSIZE];   // shared buffer of NBSLOTS each of SLOTSIZE characters
int     tag [NBSLOTS];            // record tags of NBSLOTS
//  if a producer acquired the index "in" to enqueue it's record; record's tag can also be stored in parallel
memcpy ( buf [in], producerbuf, SLOTSIZE ) ;  // enqueue the record in shared buffer
        tag [in] = poducerID ;                // store producer id e.g. 0,1,2,….
```

Structuring the Circular Queue

The **shared circular queue** is a complex object comprising: *buffer area, in and out subscripts, mutual exclusion and counting semaphores etc.* Its implementation using separate data items is prone to errors; and in case you need to define several **circular queues** in your application their handling becomes problematic. Thus you are asked to define the **shared circular queue** as a **structure**, such as the one shown here after.

```
typedef struct {   // shared queue structure definition
        char  buf [NBSLOTS] [SLOTSIZE] ; // shared data buffer
        int     tag [NBSLOTS];           // shared data tag array
        int     in, out;                 // input and output slots indexes
        sem_t  mutex_in;                 // mutex semaphore for producers
        sem_t   free_slot;               // counting semaphore free slots
        sem_t   filled_slot;             // counting semaphore filled slots
} SharedQ_Def;
```

It is also a recommended to define an initialization function for this structure such the one shown here after.

```
void init_queue ( SharedQ_Def  *pq )  {
        pq->in = 0;                              // init enqueuing index
        sem_init ( &(pq->mutex_in),   0, 1);     // init mutex for enqueueing
        ……….
        sem_init ( &(pq->free_slot),  0, NBSLOTS);  // nb of free slots
}
SharedQ_Def    sq;              // define the shared circular queue 'sq'
 init_queue (&sq);              // initialize shared queue
```

Defining the File Path and the Record Tags

It is also a recommended to define a structure that aggregates input file attributes e.g. its path and tagging identifier, such as the one listed here after.

```
typedef struct {          // input file descriptor
        int id;           // file or producer id { 0 .. PRDMAX }
        char *fn;         // input file path
} File_Def;
// initialize input file descriptors
File_Def     f1 = { 0, "/etc/passwd"};    // 0 is the tag associated with this file
File_Def     f2 = { 1, "/etc/group" };    // 1 is the tag associated with this file
  // examples of input file descriptors
  pthread_create ( &pTID[0], NULL, producer, (void *) & f1);
  pthread_create ( &pTID[1], NULL, producer, (void *) & f2 );
```

Section II. Implementation Guidelines

II.1 Producer Function

Producers read records from an input file, process and store them in a shared queue, along with a tag that identifies the file and/or the producer. Organize your flow of actions using the guidelines here after.

- ✓ open the input file; and display starting messages for the producer
- ✓ while (not EOF read the input file) {
  - P (free buffer slot)
  - use a critical section to acquire the **enqueuing** subscript "**in**" and to increment it
  - sleep 1 second to emulate the processing
  - copy the record to shared buffer
  - store the tag of the record
  - V(filled_slot)

  }
- ✓ close the file
- ✓ decrement the number of active processes
- ✓ exit

II.2 Consumer Function

The consumer retrieves records from the shared queue; counts the number of records in each file; and lists this statistics at the end. Organize your flow of actions using the guidelines here after.

- ✓ display starting messages for the consumer
- ✓ while ((**number of active producers > 0**) || (**shared buffer has records to display**) {
  - P (filled_slot)
  - sleep 1 second to emulate processing
  - use record's tag of the record to increment record count for this file
  - display processed record along with its tag
  - increment **dequeuing** subscript "**out**"
  - V(free_slot)

  }
- ✓ display record number of each file
- ✓ Exit

Section III. Project Testing - Control and Report Submission

III.1 Test Steps

Test your application:

- ✓ first using first **1 producer** reading the **"/etc/passwd**" file
- ✓ when successful, add the second producer reading the file **"/etc/group**" file.
- ✓ Perform stress tests by:
  - 'increasing the number of producers;
  - increasing the number of slots of the shared queue;
  - removing processing delays successively;
  - suppressing the display of the records
  - etc.

III.2 Project Control

The bonus should be result of your **individual work**. You are strongly advised to attend any laboratory session on **Friday December 20**, and give your Teaching Assistant the opportunity to check your personal work.

Those students who could not finish their project on that day, may ask their TA for an individual appointment until the dead line. Late applications will not be considered.

Students, who were not present in the laboratory session, should contact the instructor their eventual acceptance to a control session.

III.3 Project Submission

Following your TA's consent**,** add a comment line consisting of your name and student-id**;** and store the **source code** in the "**PrjBonus-2**" folder, located at the course web site under the tab **CSE5031-X/Assignment**; where "**X**" stands for (A,B,C,D) your laboratory session group you are registered.

Warning

You are encouraged to discuss the implementation procedures and general concepts behind the projects with your fellow students. However, **plagiarism is strictly forbidden**! Submitted report should be the result of **your personal work**!

Be advised that you are **accountable** of your submission not only for this project, but also for the mid-term, and final examinations. Your project grade may be reevaluated retrospectively, had you fail to answer correctly the same or a similar examination questions that you have solved with success in your submissions.