COMP604: Operating Systems

Semester 2, 2020

Lab-5

Due Date: 23-Aug-2020 (till 11:59pm

Producer Consumer Problem

Objective : The goal of this lab is to develop a better understanding of multithreading and process/thread synchronization using semaphores. You can do this assignment either using **Java** (`Runnable` interface and `java.util.concurrent.Semaphore`) or **C/C++** (`Pthreads`).

Overview

In Lecture 5, we have discussed a semaphore-based solution to the producer-consumer problem using a bounded buffer. In this lab we will design a programming solution to the **bounded-buffer problem** using the producer and consumer processes. The solution presented in Lecture 5 uses three semaphores: `empty`and `full`, which count the number of empty and full slots in the buffer, and `mutex`, which is a binary (or mutual exclusion) semaphore that protects the actual insertion or removal of items in the buffer. For this lab, standard counting semaphores will be used for `empty`and `full`, and a binary semaphore will be used to represent `mutex`. The producer and consumer--running as separate threads--will move items to and from a buffer that is synchronized with these empty, full, and mutex structures. You can solve this problem using either Java concurrent API or POSIX Pthreads API.

Task 1: The Buffer (or Buffer.java)

Internally, the buffer will consist of a fixed-size array of type `buffer_item` (which will be defined using a `typedef`). The array of `buffer_item` objects will be manipulated as a circular queue. The definition of `buffer_item`, along with the size of the buffer, can be stored in a Java class file or C header file such as the following:

Example snippet of buffer.h or constant.java file

```
/* buffer.h */                  // Java: constant.java
typedef int buffer_item;        // Java: omit it in Java or use
                                //       wrapper class
#define BUFFER_SIZE 5;         // Java: public static final
                                //       int BUFFER_SIZE = 5
```

The buffer will be manipulated with two functions, `insert_item()` and `remove_item()`, which are called by the producer and consumer threads, respectively. A skeleton outlining these functions appears as:

Example snippet of buffer.c or buffer.java file

```
#include <buffer.h> //Java: import

/* the buffer */

buffer_item buffer[BUFFER_SIZE];

int insert_item(buffer_item item) {
      /* insert item into buffer
      return 0 if successful, otherwise
      return -1 indicating an error condition */
}

int remove_item(buffer_item *item) {

      /* remove an object from buffer
      return 0 if successful, otherwise
      return -1 indicating an error condition */
}
```

The `insert_item()` and `remove_item()` functions will synchronize the producer and consumer using the algorithms outlined in the lecture slides. The buffer will also require an initialization code section (which is part of the `main()` function) that initializes the `empty`, `full`, and `mutex` semaphores.

Task 2: The main() Function (or Main.java)

The `main()` function will initialize the buffer and create the separate producer and consumer threads. Once it has created the producer and consumer threads, the `main ()` function will sleep for a period of time and, upon awakening, will terminate the application. The `main ()` function will be passed three parameters on the command line:

1. How long to sleep before terminating
2. The number of producer threads
3. The number of consumer threads

A skeleton for this function appears as:

Example snippet of buffer.c or Main.java file

```
#include <buffer.h>
int main(int argc, char *argv[]) { //Java: public static void
                                   //     main(String[] args)
/* 1. Get command line arguments argv[1], argv[2], argv[3]*/
/* 2. Initialize buffer */
/* 3. Create producer threads */
/* 4. Create consumer threads */
/* 5. Sleep */
/* 6. Exit */
```

Task 3: Producer and Consumer Threads (or Producer.java and Consumer.java)

The producer thread will alternate between sleeping for a *random period of time (1ms to maximum 1000 ms)* and inserting a *random integer into the buffer*. Random numbers will be produced using the `rand()` function, which produces random integers between `0` and `RAND_MAX` (Java: `Math.random()`). The consumer will also sleep for a *random period of time (1ms to maximum 1000 ms)* and, upon awakening, will attempt to remove an item from the buffer.

Please print the following for both consumer and producer threads:

1) Thread id

2) The value of the produced/consumed item.

3) The current status of the buffer, i.e., the number of items currently in the buffer as output message

Hint: In Linux C, we can get thread id using: `pthread_self()`. In Java, inside the `run()` method, we can get thread id using:

```
long threadId = Thread.currentThread().getId();
```

An outline of the producer and consumer threads is given below:

Example snippet of buffer.c or Producer.java and Consumer.java file

```
#include <stdlib.h> /* required for rand() */
#include <buffer.h>
#include <unistd.h>

void *producer(void *param) {
      buffer_item item;

      while (TRUE) {
         /* sleep for a random period of time */
         sleep( ... );

         /* generate a random number */
         item = rand() ;

         if (insert_item(rand))
           fprintf("report error condition");

         else{

            printf("producer id: %lu produced item: %d\n",
                                        pthread_self(), rand);

         /* print the status of the buffer, i.e., how many
                              items are in the buffer now */
         printf("the buffer now contains %d items\n", count);
               …
             }
       }
}




void *consumer(void *param) {
      buffer_item item;

      while (TRUE) {
      /* sleep for a random period of time */
      sleep ( ... ) ;

      if (remove_item(&item))
            fprintf("report error condition");

      else{
       printf("consumer id: %lu consumed
                               item: %d\n",pthread_self(),item);
```

```
        /* print the status of the buffer, i.e., how many items
           are inside the buffer now */
        printf("the buffer now contains %d items\n", count);

            }
        }
    }
```

Sample Output (Producer thread):

```
producer id: 3 produced item: 536
the buffer now contains 8 items
```

Task 4: Test Run

Please create a section called "*Running result analysis*" in a text file `readme.txt` and include both the running output and your explanation to the result in this section. Please allow your main function to run (sleep) for 10 seconds for the test runs below.

4.1 Run your program using 1 producer, 1 consumer, and buffer size 5. Explain the output of your run: Has the buffer ever been full? Usually how many items are in the buffer? Why?

4.2 Run your program using 5 producer, 5 consumer, and buffer size 1. Explain the output of your run: Are the producer and consumer running alternatively? Why?

4.3 Run your program using 5 producer, 5 consumer, and buffer size 10. Explain the output of your run: Are there any repeated patterns/rules in the running sequence? Why?

Submission Requirements

1. For Java code, include five files in this directory: `Main.java, Buffer.java, Producer.java, Consumber.java, Constants.java`. For C code, include two files in this directory: `buffer.h`, and `buffer.c`.

2. In addition, a text file `readme.txt` that includes:
    a. Your name and student ID.
    b. A brief instruction on how to compile and run the program
    c. Your answer to “Task 4: Test Run”.

For each submitted program file, we require well-structured code with clear comments including student information, description of the file, and description of each function defined in this file.

Submissions that fail to follow "Submission Requirements" will NOT be assessed.

Marking guide: See next page.

Marking Guide

Tasks	Criteria	Allocated Marks
General	a. readme.txt file	1
	b. Code Structure and Comments	4
	Sub-Total	**5**
Task-1	**The buffer**	
	a. Variable Declaration	1
	b. The `insert_item()` function	10
	c. The `remove_item()` function	10
	Sub-Total	**21**
Task-2	**The main function**	
	a. Get command line arguments	3
	b. Initialize buffer	1
	c. Create producer thread(s)	5
	d. Create consumer thread(s)	5
	e. sleep and exit code	5
	Sub-Total	**19**
Task-3	**Producer and Consumer Threads**	
	a. The `producer()` function	10
	b. The `consumer()` function	10
	Sub-Total	**20**
Task-4	**Test Run**	
	a. Running output and explanation (4.1)	5
	b. Running output and explanation (4.2)	5
	c. Running output and explanation (4.3)	5
	Sub-Total	**15**
Task-5	**Demonstration**	**20**
	Total	**100**