

Assignment 3: Process Synchronization

CSC 139 Operating System Principles - Spring 2019

Posted on March 17, due on April 2 (11:59 pm)

This assignment is based on the project in Chapter 7 of the textbook with slight modification. For the cited figures/sections, you need to refer to the corresponding part in the textbook.

1 Introduction

In this project, you will design a programming solution to the bounded-buffer problem using the producer and consumer processes shown in Figures 7.1 and 7.2. The solution presented in Section 7.1.1 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 project, you will use standard counting semaphores for `empty` and `full` and a mutex lock, rather than a binary semaphore, to represent `mutex`. The producer and consumer — running as separate threads — will move items to and from a buffer that is synchronized with the `empty`, `full`, and `mutex` structures. You are required to use the Pthread package to solve this problem in this project.

2 The Buffer

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 header file such as the following:

```
/* buffer.h */
typedef int buffer_item;
#define 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:

```
#include "buffer.h"

/* 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
    placing it in item
    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 Figures 7.1 and 7.2. The buffer will also require an initialization function that initializes the mutual-exclusion object `mutex` along with the `empty` and `full` semaphores.

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 follows:

```

#include "buffer.h"

int main(int argc, char *argv[]) {
    /* 1. Get command line arguments argv[1],argv[2],argv[3] */
    /* 2. Initialize buffer */
    /* 3. Create producer thread(s) */
    /* 4. Create consumer thread(s) */
    /* 5. Sleep */
    /* 6. Exit */
}

```

3 The Producer and Consumer Threads

The producer thread will alternate between sleeping for a random period of time and inserting a random integer into the buffer. Random numbers will be produced using the `rand_r(unsigned int *seed)` function, which produces random integers between 0 and `RAND_MAX` safely in multithreaded processes. The consumer will also sleep for a random period of time and, upon awakening, will attempt to remove an item from the buffer. An outline of the producer and consumer threads appears as:

```

#include <stdlib.h> /* required for rand_r(...) */
#include "buffer.h"

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

    while (true) {
        /* sleep for a random period of time */
        sleep(...);
        /* generate a random number */
    }
}

```

```

        item = rand_r(...);
        if (insert_item(item))
            fprintf("report error condition");
        else
            printf("producer produced %d\n", item);
    }

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 consumed %d\n", item);
    }
}

```

4 Thread Creation in the Pthread Package

Creating threads using the Pthreads API is discussed in Section 4.4.1. Coverage of mutex locks and semaphores using Pthreads is provided in Section 7.3. The following code sample demonstrates the Pthread APIs for creating a new thread:

```

#include <pthread.h>

void *thread_entry(void *param) { /* the entry point of a new thread */
    ...
}

int main(...) {
    pthread_t tid;
    pthread_attr_t attr;

    /* get the default attribute */
    pthread_attr_init(&attr);

    /* create a new thread */
    pthread_create(&tid, &attr, thread_entry, NULL);
    ...
}

```

The Pthread package provides `pthread_attr_init(...)` function to set the default attributes for the new thread. The function `pthread_create(...)` creates a new thread, which starts the execution from the entry point specified by the third argument.

5 Mutex Locks in the Pthread Package

The following code sample illustrates how mutex locks available in the Pthread API can be used to protect a critical section:

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create the mutex lock */
pthread_mutex_init(&mutex, NULL);

/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/** critical section **/

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

The Pthread package uses the `pthread_mutex_t` data type for mutex locks. A mutex is created with the `pthread_mutex_init(&mutex, NULL)` function, with the first parameter being a pointer to the mutex. By passing `NULL` as a second parameter, we initialize the mutex to its default attributes. The mutex is acquired and released with the `pthread_mutex_lock(...)` and `pthread_mutex_unlock(...)` functions. If the mutex lock is unavailable when `pthread_mutex_lock(...)` is invoked, the calling thread is blocked until the owner invokes `pthread_mutex_unlock(...)`. All mutex functions return a value of 0 with correct operation; if an error occurs, these functions return a nonzero value.

6 Semaphores in the Pthread Package

The pthread package provides two types of semaphores – named and unnamed. For this project, we use unnamed semaphores. The code below illustrates how a semaphore is created:

```
#include <semaphore.h>
sem_t sem;

/* create the semaphore and initialize it to 5 */
sem_init(&sem, 0, BUFFER_SIZE);
```

The `sem_init(...)` creates a semaphore and initialize it. This function is passed three parameters:

1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphore's initial value

In this example, by passing the flag 0, we are indicating that this semaphore can only be shared by threads belonging to the same process that created the semaphore. A nonzero value would allow other processes to access the semaphore as well. In this example, we initialize the semaphore to the value 5.

For the semaphore operations `wait()` and `signal()` discussed in class, the Pthread package names them `sem_wait()` and `sem_post()`, respectively. The code example below creates a binary semaphore

mutex with an initial value of 1 and illustrates its use in protecting a critical section: (Note: The code below is only for illustration purposes. Do not use this binary semaphore for protecting critical section. Instead, you are required to use the mutex locks provided by the Pthread package for protecting critical section.)

```
#include <semaphore.h>
sem_t sem_mutex;

/* create the semaphore */
sem_init(&sem_mutex, 0, 1);

/* acquire the semaphore */
sem_wait(&sem_mutex);

/** critical section **/

/* release the semaphore */
sem_post(&mutex);
```

7 Compilation

You need to link two special libraries to provide multithreaded and semaphore support using the command “gcc <files> -lpthread -lrt”.

8 Testing

You can start use one producer thread and one consumer thread for testing, and gradually use more producer and consumer threads. For each test case, you need to make sure that the random numbers generated by producer threads should exactly match the random numbers consumed by consumer threads (both their orders and their values). Make sure your code can be compiled and work on Athena server correctly.

9 Deliverables

Upload to Canvas the following:

- A document containing the printout of the code with some sample runs, SOLUTIONS.TXT.
- All the files necessary to compile, link, and run your program.
- A document describing how to run the the program you created, README.TXT.
- These files should be placed in a directory called <ECS username>-asgmt3.
- Use tar command to place all the files in a single file, <ECS username>-asgmt3.tar. Assuming you are in the directory <ECS username>-asgmt3 do the following:
 - Goto the parent directory: `cd ..`
 - tar the files:
`tar -cvf <ECS username>-asgmt3.tar ./<ECS username>-asgmt3`

- Verify the files have been placed in a tar file:
`tar -tvf <ECS username>-asgmt3.tar`
- Compress the files using gzip: `gzip <ECS username>-asgmt3.tar`
- Verify that the gzipped file exists: `ls <ECS username>-asgmt3.tar.gz`