Programming Assignment 1 - Producer-Consumer Problem

Minh Nhat Nguyen

CS 33211: OPERATING SYSTEMS

Dr. Qiang Guan

Kent State University

April 05, 2024

Introduction

This is the documentation for the programming assignment the aims to solve the producer-consumer problem. The main code for the programs is C++, and it uses semaphores, threads, and shared memory. In this document, we will discuss what the problem is, what the solution is, and how it is implemented.

1. The Producer-Consumer Problem.

The producer-consumer problem is a classic problem in concurrent programming. It involves a buffer that holds items, a producer, and a consumer. The producer places items into the buffer, while the consumer takes that item and uses it. The challenge lies within the fact that the producer and the consumer need to be coordinated properly. The producer cannot produce more items than the buffer's size limit, and the consumer must wait for the producer to make at least one item before executing.

There have been various algorithms designed to tackle this problem. In our case, we will be using semaphores, signaling mechanisms designed by Dijkstra that can be used to synchronize multiple processes. We will explain what semaphores are, and how they are relevant to the task at hand.

2. Synchronization With Semaphores

Semaphores were created as a model for the operation of railroads. According to Oracle's documentation on multithreaded programming, a semaphore

“synchronizes travel on this track. A train must wait before entering the single track until the semaphore is in a state that permits travel. When the train enters the track, the semaphore changes state to prevent other trains from entering the track. A train that is leaving this section of track must again change the state of the semaphore to allow another train to enter.”

In programming, a semaphore S is an integer variable that is either binary (0 or 1) or is a range of non-negative numbers (counting semaphores). Semaphores can only be accessed through two atomic operations: wait() and signal(). Their definitions are as follows (taken from the book: Operating System Concepts):

```
wait(S) {
    while (S <= 0)
      ; // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

By putting these operations in the producer and consumer processes, we can synchronize them and make sure that no overflow or underflow happens in the buffer.

3. Implementation

With that in mind, we can utilize semaphores in our code by using the various libraries available to us. In this particular solution, we will be using C++ to solve the problem, but these practices can be applied to any language and still yield the same result. Please note that all of these are implemented using Unix and Linux systems.

The **code** folder consists of 3 files: **shm.h**, **producer.cpp** and **consumer.cpp**. We will go through each file and discuss each of their purposes, what is included and how the codes work.

shm.h

Here is the code for the header file:

```
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <fcntl.h>
#include <ctype.h>
#include <iostream>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>

// Size of table
#define BUF_SIZE 2

// Shared buffer
struct shmbuf {
    sem_t empty;
    sem_t full;
    sem_t mutex;
    int indexIn, indexOut;
    int table[BUF_SIZE];
};
```

Firstly, let's go through all the libraries and what they provide (note that we will explain the specific operations later):

- `<semaphore.h>`: Define the `sem_t` type as well as provide semaphore related functions, namely: `sem_init`, `sem_unlink`, `sem_wait` and `sem_post`.
- `<stdio.h>`: Provides general purpose variables, like `size_t` and `NULL`
- `<stdlib.h>`: General purpose standard library of C++. In this case it provides `rand()` to generate a random number
- `<pthread.h>`: Provides thread operations, namely `pthread_create` and `pthread_join` in the main processes.
- `<fcntl.h>`: Contains constructs that refer to file control. Used in opening the shared memory in the main processes.
- `<ctype.h>`: Declares several functions that are useful for testing and mapping characters.
- `<iostream>`: Used to print messages to verify the producing and consuming process.
- `<unistd.h>`: Defines miscellaneous symbolic constants and types and declares miscellaneous functions.
- `<sys/mman.h>`: Provides a definition of a memory mapping control block in Linux. Used to check for `MAP_FAILURE` in consumer.cpp.
- `<sys/stat.h>`: Define the structure of the data returned by the stat functions. Not relevant but can be used if needed.

Next, we define the size of the buffer in case we need to change it later. For this assignment we only need a size of 2.

Finally, we declare the POSIX shared memory object named `shmbuf`. It contains the following:

- The semaphore `empty`, used to check if the buffer is empty for the consumer to wait.
- The semaphore `full`, used to check if the buffer is full for the producer to wait.
- The semaphore `mutex`, acts as a lock to alternate execution between processes.
- `indexOut` and `indexIn`, indexes used to check which buffer entry the processes are currently working with.
- `table[BUF_SIZE]`: the buffer with size of 2 (Defined earlier). It holds integers for this particular implementation but can be set to any data type.

producer.cpp

Now we go on to the producer program. Here is the code for this file:

```
/*
    Minh Nhat Nguyen
    Operating Systems - CS33211, Project 1
    Producer file
*/

#include "shm.h"

using std::cout;

// Variables for the shared memory buffer
int fd;
char *shmpath;
struct shmbuf *proBuf;

// Prodcuer thread
void *producer(void *argc)
{
```

```
    // item for buffer
    int item1;

    // run production 5 times
    int i = 0;
    while (i < 5)
    {
        item1 = rand() % 50;

        // Wait
        sem_wait(&proBuf->empty);
        sem_wait(&proBuf->mutex);

        // Critical section
        cout << "Produced: " << item1 << std::endl;
        proBuf->table[proBuf->indexIn] = item1;
        proBuf->indexIn = (proBuf->indexIn + 1) % BUF_SIZE;

        // Signal
        sem_post(&proBuf->mutex);
        sem_post(&proBuf->full);
        ++i;

        // Waits 1 second
        sleep(1);
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    srand(time(NULL));

    // Initialize shared memory
    shmpath = argv[1];
    fd = shm_open(shmpath, O_CREAT | O_EXCL | O_RDWR, 0600);
    ftruncate(fd, sizeof(*proBuf));
    proBuf = static_cast<shmbuf*>(
         mmap(
```

```
            NULL, sizeof(*proBuf),
            PROT_READ | PROT_WRITE, MAP_SHARED,
            fd, 0
        )
    );

    // Initialize semaphores and indexes
    sem_init(&proBuf->mutex, 1, 1);
    sem_init(&proBuf->empty, 1, BUF_SIZE);
    sem_init(&proBuf->full, 1, 0);
    proBuf->indexIn = 0;
    proBuf->indexOut = 0;

    // Runs producer thread
    cout << "Producer thread starting" << std::endl;
    pthread_t pro;
    pthread_create(&pro, NULL, &producer, NULL);
    pthread_join(pro, NULL);

    // Remove the name of the shared memory object
    shm_unlink(shmpath);

    // End program
    return 0;
}
```

Firstly, we include the header file for the shared memory, then we declare some variables that are needed to access the memory: `fd`, `shmpath` and `proBuf`. Their usage will be explained shortly.

Then, we define the thread for producing items. It works as follows:

- First, we declare the item to be produced (`item1`)

- Then, we run a loop that iterates a specific number of times. In this case it is 5.
- In each iteration, we generate a random number between 0 and 50, call the wait() operation on the semaphores `empty` and `mutex` with `sem_wait()`, then place the item in the buffer, increase the index (but not over the maximum size), and call signal() on `mutex` and `full` via `sem_post()`.

In the main process, it executes the following:

- It takes the name that specifies the shared memory object to be created or opened. This name is passed in the `Makefile` and is stored in `shmpath`. Then, it opens a new POSIX shared memory, assign the result value to `fd`. It truncate (`ftruncate()`) the memory to the buffer holder (`proBuf`)'s length, and finally it maps the memory object to `proBuf` with `mmap()`.
- It then initializes all 3 semaphores as well as the indexes for the threads.
- Afterwards, it creates and runs the thread via `pthread_create()` and `pthread_join()`.
- Finally, it removes the name of the shared memory object with `shm_unlink()`.

consumer.cpp

The consumer program works very similarly to the producer program, which can be seen through the code:

```
/*
    Minh Nhat Nguyen
    Operating Systems - CS33211, Project 1
    Producer file
```

```
*/

#include "shm.h"

using std::cout;

// Variables for the shared memory buffer
int fd = -1;
char *shmpath;
struct shmbuf *conBuf;

// Consumer thread
void *consumer(void *argc) {

    // item for buffer
    int item2;

    // run consumption 5 times
    int i = 0;
    while (i < 5)
    {
        // Wait
        sem_wait(&conBuf->full);
        sem_wait(&conBuf->mutex);

        // Critical section
        item2 = conBuf->table[conBuf->indexOut];
        cout << "Consumed: " << item2 << std::endl;
        conBuf->indexOut = (conBuf->indexOut + 1) %
                           BUF_SIZE;

        // Signal
        sem_post(&conBuf->mutex);
        sem_post(&conBuf->empty);
        ++i;

        // Waits 1 second
        sleep(1);
    }
```

```
    return NULL;

}

int main(int agrc, char* argv[]) {
    srand(time(NULL));

    // Opens shared memory
    shmpath = argv[1];
    while (fd == -1) fd = shm_open(shmpath, O_RDWR, 0);
    conBuf = static_cast<shmbuf*>(
         mmap (
              NULL, sizeof(shr_mem),
              PROT_READ | PROT_WRITE, MAP_SHARED,
              fd, 0
         )
     );

    // Keep retrying until the shared buffer is accessed
    while (conBuf == MAP_FAILED)
        conBuf = static_cast<shmbuf*>(
              mmap(
                   NULL, sizeof(shr_mem),
                   PROT_READ | PROT_WRITE, MAP_SHARED,
                   fd, 0
              )
        );

    // Runs consumer thread
    cout << "Consumer thread starting" << std::endl;
    pthread_t con;
    pthread_create(&con, NULL, &consumer, NULL);
    pthread_join(con, NULL);

    // Remove the name of the shared memory object
    shm_unlink(shmpath);

    // End program
    return 0;
```

```
}
```

Here are the key differences between the producer and the consumer:

- In the thread's definition, the consumer waits for full and signals empty instead. It also works with `indexOut` instead of `indexIn`, and it takes out items instead of putting them in.
- Instead of opening a new shared memory object, it accesses the one already opened by the producer, therefore the syntax slightly differs.
- Because it can only access the memory object after it is opened by the producer, the code runs it in a while loop until the memory is found. This way it makes sure that it is working with the same memory as the producer.

Now that we know what the programs do, let us go on to how to run it properly.

Compilation.

Once you have all the code, you can simply run the `Makefile` to compile the processes. You can do this by entering:

```
make all
```

into the Linux or Unix terminal. Make sure to navigate to the path where the `Makefile` is. It will clean up all the previous object files, include the header, compile all the necessary processes, and give the name to the shared memory.

Here is an example:

```
vboxuser@LinuxVM:~/Desktop/Assignment 1$ make all
rm -f producer consumer
g++ ./code/shm.h ./code/producer.cpp -pthread -lrt -o producer
g++ ./code/shm.h ./code/consumer.cpp -pthread -lrt -o consumer
./producer shmPath3 & ./consumer shmPath3
Producer thread starting
Produced: 30
Consumer thread starting
Consumed: 30
Produced: 45
Consumed: 45
Produced: 30
Consumed: 30
Produced: 25
Consumed: 25
Produced: 25
Consumed: 25
vboxuser@LinuxVM:~/Desktop/Assignment 1$
```

Conclusion

The producer-consumer problem emphasizes the intricate practice of resource management in concurrent computing systems, from synchronization, communication, to coordination between processes or threads. Although there are many solutions, semaphores being one of them, the final goal remains the same: to ensure efficient utilization of resources while maintaining performance and system integrity. This assignment has helped me greatly in understanding that mindset as well as grasping the concepts of threads, locks (semaphores in particular), how to run multiple processes at the same time with shared memory and how to implement them in C++.

References

Ftruncate(3P) - linux manual page. (n.d.).

https://man7.org/linux/man-pages/man3/ftruncate.3p.html

GfG. (2020, December 11). *How to use POSIX semaphores in C language*. GeeksforGeeks.

https://www.geeksforgeeks.org/use-posix-semaphores-c/

Producer and consumer problem (multithreaded programming guide). (n.d.).

https://docs.oracle.com/cd/E1912001/open.solaris/8165137/6mba5vq4p/index.html#:~:text=The%20producer%20and%20consumer%20problem,of%20the%20buffer%20(consumers).

Pthread_create(3) - linux manual page. (n.d.).

https://man7.org/linux/man-pages/man3/pthread_create.3.html

Pthread_join(3) - linux manual page. (n.d.).

https://man7.org/linux/man-pages/man3/pthread_join.3.html

SHM_OPEN(3) - linux manual page. (n.d.).

https://man7.org/linux/man-pages/man3/shm_open.3.html

Shm_unlink(3P) - linux manual page. (n.d.).

https://man7.org/linux/man-pages/man3/shm_unlink.3p.html

SILBERSCHATZ, A. (2021). *Operating system concepts*. JOHN WILEY.

Synchronization with semaphores. Synchronization With Semaphores (Multithreaded aProgramming Guide). (n.d.).

https://docs.oracle.com/cd/E19120-01/open.solaris/816-5137/sync-11157/index.html