Ngoc Nguyen NITK17K - 1700062

Final Project

Question
Write a program in Linux OS which, by using the means of synchronization, two processes write and read from the same resource (ex. a file). See the following link as an example.

https://thysmichels.com/2012/01/31/process-synchronization-in-linux/

Preparation Phase
Tools:

- Linux machine
- Ubuntu 20.04.1 LTS
- Virtual machine: vmware ESXi, visit this link https://www.vmware.com for more information.

Programming language:

- C

Github link
https://github.com/NgocNguyenGit/Operating-Systems

Background
Synchronization process means that multiple processes are to join up or handshake at a certain point, in order to reach an agreement or commit to a certain sequence of action. One of the classic problems of synchronization is the readers-writers problem, and it is used to test nearly every newly proposed synchronization scheme or primitive.

Synchronization in Linux provides semaphores, spinlock, barriers, mutex, readers-write locks, and ready-copy-update (RCU).

Requirement Analysis
Suppose there is a file (critical section) and it is possible that a reader R might have the lock and a writer W be waiting for the lock. There will be a scenario that there is a R2 requests access and able to jump in immediately, ahead of W, which results in W will starve. In order to help W start as soon as possible, and make sure that no readers nor writers starve, the constraint "no thread shall be allowed to starve" is added. It means that the operation of obtaining a lock on the file will always terminate in a bounded amount of time. [1]

There are two common methods to solve the readers-writers problem which are using semaphore and monitor. In order to choose the one method to apply in this project, the limitation of them should be taken under consideration.

Monitors provide a structured concurrent programming primitive. Only one call to a monitor procedure can be active at a time and this protects data inside it from simultaneous access

[1] https://en.wikipedia.org/wiki/Readers–writers_problem

by multiple users. Therefore, the absence of concurrency is the major weakness in monitors, and this leads to weakening of encapsulation. In this project, the readers-writers' file is separated from the monitor and there is a possibility that some malicious R or W can access the database directly without getting a permission from the monitor to do so. [2]

Although semaphore is a lower-level object, the value of it indicates the number of shared resources available in the system. The limitations of semaphores are priority inversion, its use is not enforced but is by convention only, and with improper use, a process may experience deadlock. However, semaphores permit more than one thread to access the critical section, here in this project is a file, unlike monitors. Hence, this project will manage to approach the readers-writers in C programming language using semaphore with a type of producer-consumer synchronization.[3]

Internal Logic and Code Explanation

There are two locks in the program: in the ***data lock***, readers and writers need to acquire and keep this lock while they are working directly with the file (data source); in the ***consumer lock***, it is initially closed and a reader waits for this lock, but it never posts to it. When it gets the reader lock, it immediately asks for the data lock and posts to the data locks when done. The producer posts to the reader lock after it releases the data lock. It means that the readers will wait unless a producer has left something to do.

```
Semaphore s;
sem_unlink("Test 1");
sem_unlink("Test 2");
//O_CREATE: create if not existant, must also give the rwx bits (mode)
    for the file
//S_IRUSR: read permission: owner
//S_IWUSR: write permission: owner
s.data = sem_open("Test 1", O_CREAT, S_IRUSR | S_IWUSR, 0);
s.reader = sem_open("Test 2", O_CREAT, S_IRUSR | S_IWUSR, 0);

//Initial opening: unlock a semaphore
sem_post(s.data);
```

Figure 1.1. Data and Reader lock

A struct named Semaphore was created, and the program must include the <semaphore.h> library in order to use it. There are three more operations to run the code which are gap, reader and writer.

[2] https://iq.opengenus.org/reader-writer-problem-cpp/
[3] https://www.geeksforgeeks.org/monitor-vs-semaphore/

```c
#ifndef ReadersWriters_h
#define ReadersWriters_h
#include <stdio.h>
#include <semaphore.h>
void gap(void);
typedef struct {
    sem_t *data;
    sem_t *reader;
} Semaphore;
void reader (Semaphore *locks, int ID);
void writer (Semaphore *locks, int ID);
#endif /* ReadersWriters_h */
```

Figure 1.2. A glimpse at the ReadersWriters header file

In the Figure 1.2, the code demonstrates a type of producer-consumer synchronization using semaphores, in which *data* is used as a mutex, and *reader* is used to count the number of items that have been produced. When the count gets to 0, there is nothing to read (consume), so consumers will block.

```c
void reader(Semaphore *locks, int ID){
    for (int i = 0; i < LIMIT; ++i){
        sem_wait(locks->reader);
        sem_wait(locks->data);
        fprintf(stderr, "***\tReader <%d> is reading %d\n",ID, i);
        sem_post(locks->data);
    }
    exit(0);
}

void writer(Semaphore *locks, int ID){
    for (int i = 0; i < LIMIT; ++i){
        sem_wait(locks->data);
        fprintf(stderr, "Writer <%d> is writing %d\n",ID, i);
        sem_post(locks->data);
        sem_post(locks->reader);
        gap();                                  //Simulate activity
    }
}
```

Figure 1.3. Reader and writer function implementation

The internal logic of the function *reader* and *writer* was basically created to solve the third problem which is called "the third readers-writers problem":

1. Give readers priority: when there is at least one reader currently accessing the resource, allow new readers to access it as well. This can cause starvation if there are writers waiting to modify the resource and new readers arrive all the time, as the writers will never be granted access as long as there is at least one active reader.
2. Give writers priority: here, readers may starve.
3. Give neither priority: all readers and writers will be granted access to the resource in their order of arrival. If a writer arrives while readers are accessing the resource, it will

wait until those readers free the resource, and then modify it. New readers arriving in the meantime will have to wait.

The gap is randomly generated to help simulate something that may take a variable amount of time, while the program is supposed to do as much as possible outside the locked sections. Hence, the processes are not simply waiting for each other to finish but have something to do.

```
#define DEFAULT 10000000
#define LIMIT 5
void gap() {
    struct timespec interval;
    interval.tv_sec = rand()%3;
    interval.tv_nsec = (rand()%100)*DEFAULT;
    nanosleep(&interval, NULL);
}
```

Figure 1.4. The random gap function implementation

The rest of the code in main() is as following:

```
s.data = sem_open("Test 1", O_CREAT, S_IRUSR | S_IWUSR, 0);
s.reader = sem_open("Test 2", O_CREAT, S_IRUSR | S_IWUSR, 0);

//Initial opening: unlock a semaphore
sem_post(s.data);

srand(time(NULL)); ⚠ Implicit conversion loses integer precision: 'time_t' (

pid_t pid = fork();
if (pid == 0){
    reader(&s, 1);
} else {
    pid = fork();
    if (pid == 0){
        writer(&s, 1);
    } else {
        pid = fork();
        if (pid == 0){
            reader(&s, 2);
        } else {
            writer(&s, 2);
        }
        sleep (5);
        sem_close(s.reader);
        sem_close(s.data);
    }
}
return 0;
```

Figure 1.5. Implementation according to the reference
https://thysmichels.com/2012/01/31/process-synchronization-in-linux/

The below figure will show how the code output with the test limit is 5.

```
Writer <2> is writing 0
Writer <1> is writing 0
*** Reader <1> is reading 0
*** Reader <1> is reading 1
Writer <1> is writing 1
*** Reader <1> is reading 2
Writer <2> is writing 1
*** Reader <2> is reading 0
Writer <1> is writing 2
*** Reader <1> is reading 3
Writer <2> is writing 2
*** Reader <2> is reading 1
Writer <1> is writing 3
*** Reader <1> is reading 4
Writer <2> is writing 3
*** Reader <2> is reading 2
Writer <1> is writing 4
*** Reader <2> is reading 3
Writer <2> is writing 4
*** Reader <2> is reading 4
Program ended with exit code: 0
```

Figure 1.6. An example of output

Pthread Implementation Management

The author tried to implement pthread to solve this problem, however, there were some obstacles when compiling the code. The internal logic was based on the pseudo code taken from Wikipedia and Samuel Tardieu's website.

```
1  semaphore accessMutex;      // Initialized to 1
2  semaphore readersMutex;     // Initialized to 1
3  semaphore orderMutex;       // Initialized to 1
4
5  unsigned int readers = 0;   // Number of readers accessing the resource
6
7  void reader()
8  {
9    P(orderMutex);            // Remember our order of arrival
10
11   P(readersMutex);          // We will manipulate the readers counter
12   if (readers == 0)         // If there are currently no readers (we came first)...
13     P(accessMutex);         // ...requests exclusive access to the resource for readers
14   readers++;                // Note that there is now one more reader
15   V(orderMutex);            // Release order of arrival semaphore (we have been served)
16   V(readersMutex);          // We are done accessing the number of readers for now
17
18   ReadResource();           // Here the reader can read the resource at will
19
20   P(readersMutex);          // We will manipulate the readers counter
21   readers--;                // We are leaving, there is one less reader
22   if (readers == 0)         // If there are no more readers currently reading...
23     V(accessMutex);         // ...release exclusive access to the resource
24   V(readersMutex);          // We are done accessing the number of readers for now
25 }
26
27 void writer()
28 {
29   P(orderMutex);            // Remember our order of arrival
30   P(accessMutex);           // Request exclusive access to the resource
31   V(orderMutex);            // Release order of arrival semaphore (we have been served)
32
33   WriteResource();          // Here the writer can modify the resource at will
34
35   V(accessMutex);           // Release exclusive access to the resource
36 }
```

Figure 2.1. Pseudo code for third readers-writers problem

The result of the following code was only allowing writing on the file, and there were no readers being accessed. Despite many researches and tries, the author had to omit this method and changed to use semaphore.

```
sem_t mutex;
sem_t *m;
#define mutex *m
sem_t db;
sem_t *d;
#define db *d
int readercount=0;
pthread_t reader1,reader2,writer1,writer2;
void *Reader(void *p);
void *Writer(void *p);

int main(int argc, const char * argv[]) {
    m = sem_open("Mutex", O_CREAT, 0644, 1);
    d = sem_open("DB",O_CREAT, 0644, 1);
    while(1)
    {
        pthread_create(&reader1,NULL,Reader,"1");
        pthread_create(&reader2,NULL,Reader,"2");
        pthread_create(&writer1,NULL,Writer,"1");
        pthread_create(&writer2,NULL,Writer,"2");
    }
```

```
void *Reader(void *p)
{
    //printf("previous value %dn",mutex);
    sem_wait(&mutex);
    printf("Mutex acquired by reader %d\n",mutex);
    readercount++;
    if(readercount==1) sem_wait(&db);
    sem_post(&mutex);
    printf("Mutex returned by reader %d\n",mutex);
    printf("Reader %s is Reading\n",p);
    sleep(3);
    sem_wait(&mutex);
    printf("Reader %s Completed Reading\n",p);
    readercount--;
    if(readercount==0) sem_post(&db);
    sem_post(&mutex);
    return NULL;
}

void *Writer(void *p)
{
    printf("Writer is Waiting\n");
    sem_wait(&db);
    printf("Writer %s is writing\n ",p);
    sem_post(&db);
    sleep(2);
    return NULL;
```

Figure 2.2. Pthread implementation

Disclaimer

The code was done in Linux OS, however, all of the figures were taken by using Xcode IDE because the update of Mac OS has problems with Virtual Box so that the author could not start the Linux OS at the time of taking the figures.