CS2106 Operating Systems

Lab 5 – Semaphores

INTRODUCTION

In this studio we will look at semaphores and how to use them. In particular we will revisit our “busy buffer” problem from Lab 4, which we solved by using mutexes. Here we will solve the problem using semaphores. Even better we will create a more advanced version of the buffer that can store multiple strings to be transmitted, not just a single string.

You will then use our new buffers to manage logging in our web server. We will also be implementing barriers using semaphores.

There are three activities in today’s lab:

Activity 1. Building a Blocking Buffer.
Activity 2. Enhancing Our Web Server with Blocking Buffers.
Activity 3. Building Barriers with Semaphores.

TEAM FORMATION

Please work in teams of two or three. Single person submissions are not allowed.

SUBMISSION INSTRUCTIONS

Please submit your answer book to your group’s IVLE folder by 2359 on Sunday 8 April 2018. Your submission should be called ExxxxxxY.docx where ExxxxxxY is the Student Number of the Team Leader. Please ensure that every member’s name and student number is in the report.

ACTIVITY 1

You are provided with three files:

buffer.h, buffer.cpp: Implementation of a simple FIFO circular buffer.
lab5p1.cpp: A program simulating the sending of data from a buffer over a 115200 bit-per-second (bps) serial link.

Step 1.

Similar to what we did in Lab 4, the lab5p1.cpp program simulates sending data over a 115200 bps serial link, with one byte sent every 70 microseconds. There are 32 sender threads sending the string "Thread X entry Y", where X is the thread number, and Y is the iteration number, once every millisecond.

Compile lab5p1.cpp with:

```
gcc lab5p1.cpp buffer.cpp –pthread –o lab5p1
```

Run lab5p1 with:

```
./lab5p1
```

Question 1. (1 mark)

Does the program produce the correct output? I.e. for a case with 3 threads, the output should show something like:

```
Thread 2 entry 0
Thread 2 entry 1
Thread 2 entry 2
Thread 0 entry 0
Thread 0 entry 1
Thread 1 entry 0
Thread 1 entry 1
Thread 2 entry 3
Etc.
```

I.e. All the threads and entry numbers should be present, though the threads may not run in order.

Question 2. (3 marks)

Examine the code in buffer.cpp. Explain why this code could be executed incorrectly in a multithreaded environment.

Step 2.

One issue alluded in Question 2 could be that the buffer variables are suffering from race conditions.

Question 3. (4 marks)

Using what you have learnt in the lectures and what we have covered on mutexes in the previous lab, modify buffer.cpp so that the shared variables are protected by one or more mutexes. (Note: to ensure data encapsulation, declare your mutex inside the TBuffer structure in buffer.h)

Cut, paste and explain your code.

Question 4. (1 mark)

Compile your program and run it. Is the output correct? (A yes or no will be adequate. You don't need to explain why or why not here: We will do that after step 3).

Step 3.

We will now use counting semaphores to stop writes to the buffer when it is full, and to top reads when it is empty.

a. Add in the following line to the top of buffer.h:

    #include <semaphore.h>

b. Modify TBuffer (declared in buffer.h) to include the following semaphore variables.

    sem_t empty, full.

   Your TBuffer structure will look like this (note: This includes the mutex from Question 3):

```
typedef struct
{
    sem_t empty, full;
    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
    char data[QLEN][ENTRY_SIZE];
    int len[QLEN];
    int front, back;
    int count;
} TBuffer;
```

c. Now in enq:

    Add in a sem_wait(&buffer->empty) near the start of enq.
    Add in a sem_post(&buffer->full) near the end of enq. Be sure to do it before you release your mutex (why?)

d. Similar in deq:

Add in a sem_wait(&buffer->full) near the start of deq.
Add in a sem_post(&buffer->empty) near the end of deq, making sure you do it before the releasing your mutex.

Compile your program using:

```
gcc lab5p1.cpp buffer.cpp –pthread –lrt –o lab5p1
```

(Note the extra –lrt option, required for using semaphores.)

Run it as usual with:

```
./lab5p1
```

Question 5. (3 marks)

You will see that your program now runs correctly. Explain what you think the problem was in Question 4, and how you think the sem_wait and sem_post in parts c and d above fixed these problems

ACTIVITY 2. MODIFYING YOUR WEB SERVER

In the previous lab you used a rather unsatisfactory way to control logging in your web server. Now in Activity 1 you created a buffer that is secured using mutexes, and regulated using semaphores. The mutexes ensure that the buffer variables do not have race conditions and are updated correctly, while the semaphores ensure that you do not overwrite entries in the buffer.

Question 6. (4 marks)

Modify your web server from Question 9 in Lab 4 to use the buffer routines in buffers.cpp. Cut and paste your code into the answer book explaining your changes.

Hint: writeLog should enq new messages to the log, while main (or any other routine actually writing to the log file) should deq the new messages and write to the log file.

ACTIVITY 3. BUILDING BARRIERS WITH SEMAPHORES

In the lecture we learnt about barriers, and in this activity we will see how to implement barriers using semphores.

You are provided with a barrier.h and barrier.cpp, which implement the barrier class. You are also provided with testbarrier.cpp, which creates several processes and delay for a random number of seconds before calling reachBarrier to indicate that the process has reached the barrier.

There are three functions in barrier.h that were created for you.

Function	Parameters	Description
initBarrier	TBarrier *barrier Int numProcessess	Initializes a new barrier. Barrier = Barrier to initialize numProcesses = # of processes this barrier is expecting. When this many processes have called reachBarrier, everyone is unblocked. Otherwise a process calling reachBarrier will remain blocked.
reachBarrier	TBarrier *barrier int procNum)	Lets a process/thread tell the barrier it has arrived. barrier = Barrier you are using. procNum = Your own process number. See testbarrier.cpp how to derive this. If fewer than numProcesses processes have called reachBarrier, anyone calling reachBarrier is blocked. Once numProcesses have called reachBarrier, all processes are simultaneously unblocked.
resetBarrier	TBarrier *barrier	Resets number of processes at the barrier to 0.

Compile your testBarrier program with:

```
gcc barrier.cpp testbarrier.cpp –pthread –lrt –fpermissive –o testbarrier
```

Run it with

```
./testbarrier
```

Question 7. (1 mark)

Describe what is happening when you run ./testbarrier.

Question 8. (3 marks)

Describe how the reachBarrier function in barrier.cpp works; i.e. how semaphores are used to implement barrier behaviour.