Lab-7: Classical Synchronization Problems

Lab-7 can be done in a group of 5 to 8 students. Carefully read the document and implement all the steps. Submit a PDF document containing your code, relevant screenshots and team members' names and roll numbers. Only one PDF should be submitted per team. Ensure that your PDF document is strictly in line with the material mentioned in the lab document. Part 1 of the document should be done during the lab hours. For part 2, you can take additional time. You only need to submit part 2. Part 1 is supposed to completed in the lab hours.

1 Bounded buffer problem

Refer to section 5.1 of the textbook (Link here) to understand the problem well. Our objective is to implement 2 threads, producer and consumer according to the given pseudo-code in the book. However, for the purpose of understanding semaphores, we will only focus on how the value of "count" changes. We won't care about the values in the buffer. So, you can ignore the array "buffer", and integers "in" and "out".

1.1 Setting up the code

Use the following code to set up two parallely running threads. The variable "count" (with initial value as 0) is being shared by two threads "producer" and "consumer". Each iteration of "producer" increments count by 1 (denoting an item produced in the buffer) and that of "consumer" decrements count by 1 (denoting the consumption of the item). The variable "times" indicates the number of times each thread runs. If both the threads were to run in perfect synchronization (without any race condition), then count should be 0 when both the threads have finished executing. Execute the following program for yourself. Try and have some fun with the code. Do you get the expected result all the time? Try changing the value of "times" to large values. Does the result change?

Ensure that you compile the code as gcc -pthread [filename.c]

```c
#include <stdio.h>
#include <stdlib.h>
#include<pthread.h>

int count = 0;
int times = 90;
void *producer(void *arg) {
    int i;
    for (i = 0; i < times; i++)
    {
  count++;
    }
    return NULL;
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < times; i++)
    {
  count--;
    }
    return NULL;
}

int main() {
    pthread_t p1, p2;
    printf("Initial count : %d\n", count);
    pthread_create(&p1, NULL, producer, NULL);
    pthread_create(&p2, NULL, consumer, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final count : %d\n", count);
    return 0;
}
```

You might have figured out by now that increasing the variable “times” results in race conditions. Hence, the output is not as expected. Let us now include semaphores in our code to ensure mutual exclusion. But before you proceed,

1. Identify the critical sections and remainder sections for both producer and consumer?
2. At what places in the code do we need to include the entry and exit section?

1.2 Achieving mutual exclusion with semaphores

Semaphore is an abstract data type with an integer value that we can manipulate with two routines; in the POSIX standard, these routines are sem_wait() and sem_post(). Because the initial value of the semaphore determines its behavior, before calling any other routine to interact with the semaphore, we must first initialize it to some value. You can use the following code to initialise a semaphore.

```c
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

Your first task is to correctly include the semaphore operations in the above code to avoid race condition. Try it on your own. If you find it challenging, refer to the following code but before proceeding further, make sure that you understand the code and the semaphore operations well.

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include<pthread.h>
4  #include<semaphore.h>
5  sem_t lock;
6  int count = 0;
7  int times = 90000;
8  void *producer(void *arg) {
9      int i;
10     for (i = 0; i < times; i++)
11     {
12         sem_wait(&lock);
13   count++;
14   sem_post(&lock);
15     }
16     return NULL;
17 }
18
19 void *consumer(void *arg) {
20     int i;
21     for (i = 0; i < times; i++)
22     {
23         sem_wait(&lock);
24   count--;
25   sem_post(&lock);
26     }
27     return NULL;
28 }
29
30 int main() {
31     sem_init(&lock,0,1);
32     pthread_t p1, p2;
33     printf("Initial count : %d\n", count);
34     pthread_create(&p1, NULL, producer, NULL);
35     pthread_create(&p2, NULL, consumer, NULL);
36     pthread_join(p1, NULL);
37     pthread_join(p2, NULL);
38     printf("Final count : %d\n", count);
39     return 0;
40 }
```

Closely study lines 4, 5, 12, 14, 23, 25 and 31. Before you proceed,

1. Identify the entry and exit sections for the critical section?
2. Do you think that all the 3 requirements of synchronization, i.e., mutual exclusion, progress and bounded wait are met?

1.3 Achieving ordering with semaphores

Print the value of count and note how producer and consumer are changing it. Eventually, the value reaches 0. However, you might notice count becoming negative at times. Since count represents the number of items in the buffer, it can't go below 0. Similarly, if the buffer size is 100, count should not exceed 100. Hence, let us make producer and consumer processes wait whenever they find count to be buffer size or 0 respectively. Note that you need to define a new variable for buffer size.

1. Refer to the pseudo-code in the textbook and add relevant busy wait (while loop) to resolve this.
2. Since we don't prefer busy waits, use two additional semaphores variables to resolve the problem. You will soon realise that initialising semaphores with 1 will no longer help you. You might need to initialise a semaphore variable as 10 or 20 or 300 depending on the buffer size. Finding it tricky?, refer to the section 5.7.1 of the textbook.

2 For brownie points

In the same way as above, model the problems mentioned in section 5.7.2 and 5.7.3 of the textbook. For the first problem, assume that there are 2 writers and 4 readers. For the second problem, assume 5 philosophers to be dining together. Note that you will get marks based on the clarity and precision of your document.

3 Submission Instructions

Upload a single PDF titled Lab7_[Roll numbers in capital letters separated with underscores].