INDIAN INSTITUTE OF TECHNOLOGY GOA

OPERATING SYSTEMS LAB (310)

Faculty: Dr. Sharad Sinha
Teaching Assistant: Prachi Kashikar

<u>LAB 05</u>

In this lab, you will be introduced to IPC mechanisms and synchronization using semaphores. This lab exercise will be split into two parts. At the end there are assignment questions provided which you could try after the lab exercise is done.

PART A:

Since all threads run in the same address space, they all have access to the same data and variables. If two threads simultaneously attempt to update a global counter variable, it is possible for their operations to interleave in such way that the global state is not correctly modified. Although such a case may only arise only one time out of thousands, a concurrent program needs to coordinate the activities of multiple threads using something more reliable that just depending on the fact that such interference is rare. The semaphore is designed for just this purpose.
Threads can greatly simplify writing elegant and efficient programs. However, there are problems when multiple threads share a common address space.
To understand what might happen, let us analyze this simple piece of code:

```
THREAD 1            THREAD 2
 a = data;          b = data;
 a++;               b--;
 data = a;          data = b;
```

Now if this code is executed serially (for instance, THREAD 1 first and then THREAD 2), there are no problems. However, threads execute in an arbitrary order, so consider the following situation assuming data = 0 initially:

Thread 1	**Thread 2**	**data**
a = data;	---	0
a = a+1;	---	0
---	b = data; // 0	0
---	b = b - 1;	0
data = a; // 1	---	1
---	data = b; // -1	-1

So data could end up +1, 0, -1, and there is **NO WAY** to know which value! It is completely non-deterministic!
The solution to this is to provide functions that will block a thread if another thread is accessing data that it is using.
Pthreads may use semaphores to achieve this.

Reference codes are provided along with this manual to understand this concept further.

PART B:

1. Compile and run lab5_1.c with gcc. Review lab5_1.c to be sure you understand how it works.
2. Use the idea of semaphores, as implemented in lab5_1.c , in place of the spinlocks in lab 4 to handle the synchronization of the reader and writer process from lab5_1.c .
3. Compile and run lab5_2.c with gcc. Review lab5_2.c to be sure you understand how it works.
4. This program contains a third semaphore, mutex. Explain the purpose of this semaphore. Specifically, if semaphore mutex were omitted, give a timing sequence involving the several readers and/or writers showing what might go wrong.
5. Program lab5_2.c prevents any reader from working at the same time as any writer. Assuming that the buffer contains several locations, however, writing to one buffer location should not interfere with reading from another. That is, the critical section for readers need not be considered exactly the same as the critical section for writers. Remove semaphore mutex and add additional semaphores, as needed, so that some reader could work concurrently with some writer (assuming the buffer contained some data but was not full – so both reading and writing made sense).

Assignment:

1. Consider the following problem: A program is to be written to print all numbers between 1 and 1000 (inclusive) that are not (evenly) divisible by either 2 or 3. This problem is to be solved using three processes (P0, P1, P2) and two one-integer buffers (B0 and B1) as follows:

   1. P0 is to generate the integers from 1 to 1000, and place them in B0 one at a time. After placing 1000 in the buffer, P0 places the sentinel 0 in the buffer, and terminates.
   2. P1 is to read successive integers from B0. If a value is not divisible by 2, the value is placed in B1. If the value is positive and divisible by 2, it is ignored. If the value is 0, 0 is placed in B1, and P1 terminates.
   3. P2 is to read successive integers from B1. If a value is not divisible by 3, it is printed. If the value is positive and divisible by 3, it is ignored. If the value is 0, P2 terminates.

   Write a program to implement P0, P1, and P2 as separate processes and B0 and B1 as separate pieces of shared memory – each the size of just one integer. Use semaphores to coordinate processing. Access to B0 should be independent of access to B1; for example, P0 could be writing into B0 while either P1 was writing into B1 or P2 was reading.

2. Write a C program to implement the following game. The parent program P first creates two pipes, and then spawns two child processes C and D. One of the two pipes is meant for communications between P and C, and the other for communications between P and D. Now, a loop runs as follows. In each iteration (also called round), P first randomly chooses one of the two flags: MIN and MAX (the choice randomly varies from one iteration to another). Each of the two child processes C and D generates a random positive integer and sends that to P via its pipe. P reads the two integers; let these be c and d. If P has chosen MIN, then the child who sent the smaller of c and d gets one point. If P has chosen MAX, then the sender of the larger of c and d gets one point. If c= d, then this round is ignored. The child process who first obtains ten points wins the game. When the game ends, P sends a user-defined signal to both C and D, and the child processes exit after handling the signal (in order to know who was the winner). After C and D exit, the parent process P exits. During each iteration of the game, P should print appropriate messages (like P's choice of the flag, the integers received from C and D, which child gets the point, the current scores of C and D) in order to let the user know how the game is going on. Name your program childs game.c .