Assignment 3 : Synchronizing Threads with POSIX Semaphores

Now it is time to take a look at some code that does something a little unexpected. The program badcnt.c creates two new threads, both of which increment a global variable called cnt exactly NITER, with NITER = 1,000,000. But the program produces unexpected results.

Exercise 1. Compile the code badcnt.c using

gcc badcnt.c -o badcnt -lpthread

Run the executable badcnt and observe the ouput.

Quite unexpected! Since **cnt** starts at 0, and both threads increment it NITER times, we should see cnt equal to 2\*NITER at the end of the program. What happens?

Threads can greatly simplify writing elegant and efficient programs. However, there are problems when multiple threads share a common address space, like the variable cnt in our earlier example.

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:

|  |  |  |
| --- | --- | --- |
| **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.

**Posix semaphores**

All POSIX semaphore functions and types are prototyped or defined in semaphore.h. To define a semaphore object, use

sem\_t *sem\_name*;

To initialize a semaphore, use sem\_init():

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

* sem points to a semaphore object to initialize
* pshared is a flag indicating whether or not the semaphore should be shared with fork()ed processes. Linux Threads does not currently support shared semaphores
* value is an initial value to set the semaphore to

Example of use:

sem\_init(&sem\_name, 0, 10);

To wait on a semaphore, use sem\_wait:

int sem\_wait(sem\_t \*sem);

Example of use:

sem\_wait(&sem\_name);

* If the value of the semaphore is negative, the calling process blocks; one of the blocked processes wakes up when another process calls sem\_post.

To increment the value of a semaphore, use sem\_post:

int sem\_post(sem\_t \*sem);

Example of use:

sem\_post(&sem\_name);

* It increments the value of the semaphore and wakes up a blocked process waiting on the semaphore, if any.

To find out the value of a semaphore, use

int sem\_getvalue(sem\_t \*sem, int \*valp);

* gets the current value of sem and places it in the location pointed to by valp

Example of use:

int value;

sem\_getvalue(&sem\_name, &value);

printf("The value of the semaphors is %d\n", value);

To destroy a semaphore, use

int sem\_destroy(sem\_t \*sem);

* destroys the semaphore; no threads should be waiting on the semaphore if its destruction is to succeed.

Example of use:

sem\_destroy(&sem\_name);

**Using semaphores - a short example**

Consider the problem we had before and now let us use semaphores:

Declare the semaphore global (outside of any funcion):

sem\_t mutex;

Initialize the semaphore in the main function:

sem\_init(&mutex, 0, 1);

|  |  |  |
| --- | --- | --- |
| **Thread 1** | **Thread 2** | **data** |
| sem\_wait (&mutex); | --- | 0 |
| --- | sem\_wait (&mutex); | 0 |
| a = data; | /\* blocked \*/ | 0 |
| a = a+1; | /\* blocked \*/ | 0 |
| data = a; | /\* blocked \*/ | 1 |
| sem\_post (&mutex); | /\* blocked \*/ | 1 |
| /\* blocked \*/ | b = data; | 1 |
| /\* blocked \*/ | b = b + 1; | 1 |
| /\* blocked \*/ | data = b; | 2 |
| /\* blocked \*/ | sem\_post (&mutex); | 2 |
| **[data is fine. The data race is gone.]** | | |

Exercise 2.Use the example above as a guide to fix the program badcnt.c, so that the program always produces the expected output (the value 2\*NITER). Make a copy of badcnt.c into goodcnt.c before you modify the code.

To compile a program that uses pthreads *and* posix semaphores, use

gcc -o filename filename.c -lpthread -lrt

Exercise 3.  ***Complete the incomplete code*** producer-comsumer.c to implement a solution to the Producer-Consumer problem using Posix threads and semaphores.

Comment well your code. Compile and run your program and observe the output. Label each line in the output by the identifier for each producer and consumer (P1, P2, P3, C1, C2, C3). The output of your program should be similar to the following:

[P0] Producing 0 ...

[P1] Producing 0 ...

[P2] Producing 0 ...

[P2] Producing 1 ...

------> [C2] consumed 0

------> [C2] consumed 1

[P1] Producing 1 ...

------> [C1] consumed 0

------> [C1] consumed 1

[P0] Producing 1 ...

------> [C1] consumed 1

------> [C0] consumed 0

[P2] Producing 2 ...

[P2] Producing 3 ...

[P1] Producing 2 ...

[P1] Producing 3 ...

------> [C0] consumed 2

------> [C0] consumed 3

------> [C0] consumed 2

------> [C1] consumed 3

[P0] Producing 2 ...

[P0] Producing 3 ...

------> [C2] consumed 2

------> [C2] consumed 3

To compile a program that uses pthreads *and* posix semaphores, use

gcc -o filename filename.c -lpthread -lrt