Lab 5 Synchronization

April 2021

1 Problem

2 Synchronization

3 Synchronization with Thread

4 Exercise (Required)

Problem 1 (5 points): Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: **deposit (amount)** and **withdraw (amount)**. These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the withdraw() function and the wife calls deposit(). Write a short essay listing possible outcomes we could get and pointing out in which situations those outcomes are produced. Also, propose methods that the bank could apply to avoid unexpected results.

Solution:
We could get the following possible outcomes:
When the Balance = 5000 and husband calls deposit(1000) then balance is 6000. And, concurrently, balance = 5000 and wife calls withdraw(1000) then balance is 4000. We can clearly see an inconsistency, which is not acceptable.
S0: husband execute register1 = balance (register1 = 5000)
S1: husband execute register1 = register1 + 1000 (register1 = 6000)
S2: wife execute register2 = balance (register2 = 5000)
S3: wife execute register2 = register2 – 1000 (register2 = 4000)
S4: husband execute balance = register1 (balance = 6000)
S5: wife execute balance = register2 (balance = 4000)
- Data inconsistency
This **race condition** happened when both husband and wife use 2 functions concurrently.

To solve this problem we apply the **Peterson's Algorithm** (to prevent race condition) Flags can be used for husband and wife accessing the bank account

For ex, the Husband section can be written below
int turn = 0;
bool flag[2] = {false,false}
/*The variable turn indicates whose turn it is to enter the critical section (husband or wife)*/
/*The flags make sure the events of 2 users accessing the software are mutually exclusive. */
while (true){
flag[i] = true;
turn = j;
while(flag [(i+1)%2] && turn==(i+1)%2);
/* do nothing */
deposit(1000); /* critical section */
flag[i] = false;
/* remainder section */
}

- This section can be repeated with user 2(Wife) with i will be replaced by j. i=0 or 1 and j=1 or 0.

Problem 2 (2 points) Write a new program nosynch.c by copying the program cond_usg.c (Section 4) and then removing all entry and exit sections. Write your remarks about the displayed outputs when executing these two programs nosynch.c and cond_usg.c.

Solution:
Outputs of cond_usg.c:

```
main : begin
Starting watch_count ( ) : thread 1
inc_count () : thread 2 , count = 11, uncloking mutex
watch_count () : thread 1 , count = 11, waiting . . .
inc_count () : thread 3 , count = 12, uncloking mutex
inc_count () : thread 2 , count = 13, uncloking mutex
inc_count () : thread 3 , count = 14, uncloking mutex
inc_count () : thread 3 , count = 15, uncloking mutex
inc_count () : thread 2 , count = 16, uncloking mutex
inc_count () : thread 2 , count = 17, uncloking mutex
inc_count () : thread 3 , count = 18, uncloking mutex
inc_count () : thread 3 , count = 19, uncloking mutex
inc_count () : thread 2 , count = 20, threshold reached .
Just sent signal
inc_count () : thread 2 , count = 20, uncloking mutex
watch_count ( ) : thread 1 . Condition signal received .Count = 20
watch_count ( ) : thread 1 Updating the count value . . .
watch_count ( ) : thread 1 count now = 100
watch_count ( ) : thread 1 . Unlocking mutex.
inc_count () : thread 3 , count = 101, uncloking mutex
inc_count () : thread 2 , count = 102, uncloking mutex
inc_count () : thread 3 , count = 103, uncloking mutex
inc_count () : thread 2 , count = 104, uncloking mutex
inc_count () : thread 3 , count = 105, uncloking mutex
inc_count () : thread 2 , count = 106, uncloking mutex
inc_count () : thread 3 , count = 107, uncloking mutex
inc_count () : thread 2 , count = 108, uncloking mutex
inc_count () : thread 2 , count = 109, uncloking mutex
inc_count () : thread 3 , count = 110, uncloking mutex
inc_count () : thread 2 , count = 111, uncloking mutex
inc_count () : thread 3 , count = 112, uncloking mutex
inc_count () : thread 2 , count = 113, uncloking mutex
inc_count () : thread 3 , count = 114, uncloking mutex
inc_count () : thread 2 , count = 115, uncloking mutex
inc_count () : thread 3 , count = 116, uncloking mutex
```

. . . . .

```
inc_count () : thread 3 , count = 284, uncloking mutex
inc_count () : thread 2 , count = 285, uncloking mutex
inc_count () : thread 3 , count = 286, uncloking mutex
inc_count () : thread 3 , count = 287, uncloking mutex
inc_count () : thread 2 , count = 288, uncloking mutex
inc_count () : thread 2 , count = 289, uncloking mutex
inc_count () : thread 3 , count = 290, uncloking mutex
main : finish , final count = 290
```

Remark:
(thread 1(watch_count),thread 2,3 (int_count)
- First the program executes thread 1 (watch_count()), concurrently, thread 2 (int_count()) also runs. Both threads use MUTEX LOCK for avoid race conditions.
- Then, thread 1 met wait condition, thus kept waiting for thread 2 and 3 running. When count = 20, it met signal condition, which wake up thread 1 to run, but had to wait for mutex unlock...
- Thread 1 continued running, finished, and unlocked mutex. Then, thread 2 and 3 continued to run till the end.
- We got final result count = 290 (as expected), 10(initial) + 100(thread 2) + 100 (thread 3) + 80(thread 1) = 290
(because we had avoided race condition by MUTEX LOCK)
Output of nosynch.c:

```
main : begin
inc_count () : thread 2 , count = 11, uncloking mutex
inc_count () : thread 3 , count = 12, uncloking mutex
Starting watch_count ( ) : thread 1
watch_count () : thread 1 , count = 12, waiting . . .
watch_count ( ) : thread 1 . Condition signal received .Count = 12
watch_count ( ) : thread 1 Updating the count value . . .
watch_count ( ) : thread 1 count now = 92
watch_count ( ) : thread 1 . Unlocking mutex.
inc_count () : thread 2 , count = 93, uncloking mutex
inc_count () : thread 3 , count = 94, uncloking mutex
inc_count () : thread 2 , count = 95, uncloking mutex
inc_count () : thread 3 , count = 96, uncloking mutex
inc_count () : thread 3 , count = 97, uncloking mutex
inc_count () : thread 2 , count = 98, uncloking mutex
inc_count () : thread 3 , count = 99, uncloking mutex
inc_count () : thread 2 , count = 100, uncloking mutex
inc_count () : thread 3 , count = 101, uncloking mutex
inc_count () : thread 2 , count = 102, uncloking mutex
inc_count () : thread 3 , count = 103, uncloking mutex
inc_count () : thread 2 , count = 104, uncloking mutex
```

. . . . .

```
inc_count () : thread 3 , count = 276, uncloking mutex
inc_count () : thread 2 , count = 277, uncloking mutex
inc_count () : thread 3 , count = 278, uncloking mutex
inc_count () : thread 2 , count = 279, uncloking mutex
inc_count () : thread 3 , count = 281, uncloking mutex
inc_count () : thread 2 , count = 280, uncloking mutex
main : finish , final count = 281
```

Remark:
- With no condition variables and MUTEX lock, 3 threads runned concurrently and gave wrong final result (281). Moreover, watch_count() didn't wait for thread 2 and 3 to run (due to lack of condition wait and cond signal)
- Therefore, condition wait and signal are used to sleep a thread, mutex lock are used to avoid race conditions by prevent other thread to enter critical section.

Problem 3:

The modified program for pi calculation gave the following result:

```
antslayer@DESKTOP-43H1UG3:/mnt/c/Users/Admin/Desktop/OS-LAB/1952737-CC04-lab3$ ./pi_multi-thread 50000000
3.141746
Time taken: 2 sec
```

Last output (race condition):

```
antslayer@DESKTOP-43H1UG3:/mnt/c/Users/Admin/Desktop/OS-LAB/1952737-CC04-lab3$ ./pi_multi-thread 50000000
0.000000
Time taken: 0 sec
```

This result is more precise than the one in lab 3 since we had avoided race condition using mutex lock.