FIT2100 Laboratory #5
Concurrency, Race Conditions: Mutual Exclusion and Synchronisation, Week 10 Semester 2 2024

August 16, 2024

Revision Status:

Updated by Dr. Charith Jayasekara, Aug 2024.

Acknowledgement

Some content in this Laboratory adapted from: William Stallings (2017). *Operating Systems: Internals and Design Principles (9th Edition)*, Pearson.

Contents

1 Background

In this Laboratory, you will have the opportunity to explore further on the various concepts of concurrency as discussed in the lectures.

Before attending the Laboratory, you should:

- Complete your pre-class readings (Section 2)
- Prepare a list of questions on concepts that we have covered so far, which you would like to ask your tutor during the Laboratory
- Attempt the Laboratory tasks and come up with answers to the "Something to Think About" questions (Sections 3 and 4)
- Attempt Assessed laboratory tasks in section 5

Important: The review questions and problem-solving tasks in this module are not assessed, but there is a pre-class quiz for this module.

2 Pre-class Reading

You should complete the following two sets of reading:

- Lecture Notes: Week 7-9.
- Stallings' textbook: Chapter 5 and Chapter 6

3 Synchronisation and Mutual Exclusion

3.1 Non-Assessed Tasks

3.1.1 Task 1

Consider the pseudocode of the following programs[1]:

```
P1:
{
 shared int x;
 x = 10;
 while(1) {
     x = x - 1;
     x = x + 1;
     if (x != 10) {
         printf("x is %d", x);
     }
 }
}
```

```
P2:
{
 shared int x;
 x = 10;
 while(1) {
     x = x - 1;
     x = x + 1;
     if (x != 10) {
         printf("x is %d", x);
     }
 }
}
```

(a) Show a sequence (i.e. trace the sequence of inter-leavings of statements) such that the statement "`x is 10`" is printed.

3.1.2 Task 2

Consider the pseudocode of the following program:

```
const int n = 50;
int tally;

void total() {
    int count;
    for (count = 1; count <= n; count++)
        tally++;
}

void main() {
    tally = 0;
    parbegin (total(), total());
    printf ("%d\n", tally);
}
```

[1] Note that the scheduler in a uniprocessor system would implement pseudo-parallel execution of these two concurrent processes by *interleaving their instructions*, without restriction on the order of the interleaving.

(a) Determine the proper *lower bound* and *upper bound* of the final value for the shared variable `tally` outputted by the program that concurrently runs two instances of the `total()` function. Assume that the process can execute in any relative speed and that a value can be incremented after it has been loaded into a register by a separate machine instruction.

(b) Suppose that an arbitrary number of the above total processes are permitted to execute in parallel (say `N` such processes) under the same assumption of part (a). What effect this modification may have on the range of final values of `tally`?

3.1.3 Task 3

Consider the following definitions of *semaphores*:

```
1   void semWait(s)
2   {
3       if (s.count > 0) {
4               s.count--;
5       }
6       else {
7               /* place this process in s.queue */
8               /* block this process */
9       }
10  }
11
12  void semSignal(s)
13  {
14      if (there is at least one process blocked on semaphore s) {
15              /* remove a process P from s.queue */
16              /* place process P on the ready list */
17      }
18      else {
19              s.count++;
20      }
21  }
```

Compare this set of definitions with the other set of definitions presented in the lecture notes (Lecture 9, Slide 11). Note one difference: with the preceding definition, a semaphore can never take on a negative value.

(a) Is there any difference in the effect of the two sets of definitions when used in programs? Why, or why not?

(b) Could you substitute one set for the other without altering the meaning of the program? Why, or why not?

3.2 Something to Think About

Question 1

What is a *race condition*?

Question 2

What is mutual exclusion? Is mutual exclusion important for the execution of execution of concurrent processes?

Question 3

What is the difference between *binary* semaphores and *general* semaphores?

4 Deadlock and Starvation

4.1 Non-Assessed Tasks

4.1.1 Task 4

Consider the pseudocode of the following program, where three processes are competing for six resources labelled as **A** to **F**. By using a *resource allocation graph*, demonstrate the possibility of a deadlock in this implementation.

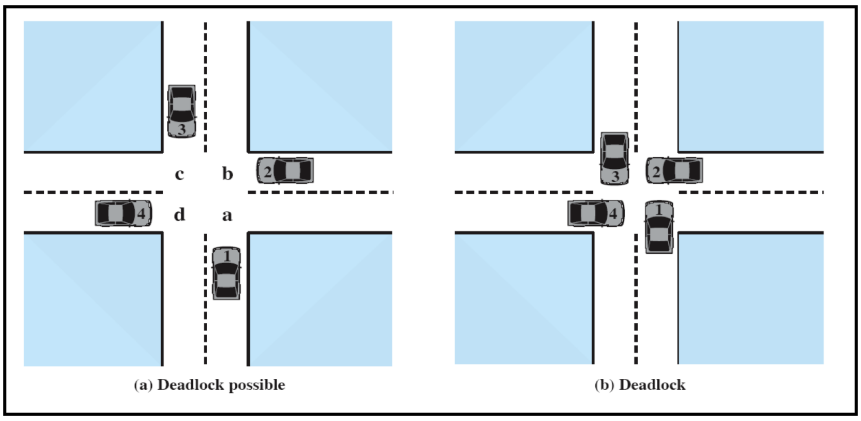
```
void P0()
{
   while (true) {
      get(A);
      get(B);
      get(C);
   // critical region:
   // use A, B, C
      release(A);
      release(B);
      release(C);
   }
}
```

```
void P1()
{
   while (true) {
      get(D);
      get(E);
      get(B);
   // critical region:
   // use D, E, B
      release(D);
      release(E);
      release(B);
   }
}
```

```
void P2()
{
   while (true) {
      get(C);
      get(F);
      get(D);
   // critical region:
   // use C, F, D
      release(C);
      release(F);
      release(D);
   }
}
```

4.1.2 Task 5

Demonstrate that the four conditions of deadlock apply to the following figure.



4.1.3 Task 6

Discuss how each of the approaches of *prevention*, *avoidance*, and *detection* can be applied to the deadlock situation presented in Task 4.

4.2 Something to Think About

Question 1

What are the four conditions that create *deadlocks*?

Question 2

Why should *mutual exclusion* not be disallowed in order to prevent deadlocks?

Question 3

How can the *hold-and-wait* condition be prevented?

Question 4

Suggest two ways in which the *no-preemption* condition can be prevented.

Question 5

Describe the three general approaches used for handling the problem of deadlocks.

5 Assessed Laboratory Tasks

5.1 Task 1: Analyzing Concurrency Issues (30%)

Description:
In this task, you will analyze a C program that has a race condition because it does not use any mutexes or other synchronization mechanisms. The program creates two threads that increments a shared variable *counter* a million times.

Instructions:

1. Download the 'RaceCondition.c' file from Moodle.
2. Compile the program using the following command:

   ```
   gcc -o RaceCondition RaceCondition.c -lpthread
   ```

3. Run the program several times (10 times) using the following command:

   ```
   ./RaceCondition
   ```

4. Record the output of each run.
5. Analyze and explain the reason for different outputs in each run.
6. Based on your understanding and the results you observed, can you identify a possible range for the final value of the 'counter' variable? Explain your reasoning.

5.2 Task 2 : Using Mutex to Resolve Race Condition Issues (30%)

1. Make a copy of `RaceCondition.c` and save it as `Task2.c`.
2. Open `Task2.c` in your favorite text editor or IDE.
3. At the top of the file, declare a global `pthread_mutex_t` variable named `mutex`.

```
pthread_mutex_t mutex;
```

4. In the `main` function, before creating the threads, initialize the mutex using `pthread_mutex_init`.

```
pthread_mutex_init(&mutex, NULL);
```

5. Modify the `updateCounter` function to lock the mutex before accessing the shared `counter` variable and unlock the mutex after modifying it. Use `pthread_mutex_lock` and `pthread_mutex_unlock` functions for this purpose.

```
void *updateCounter(void *args) {
    int id = *(int *)args;
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        pthread_mutex_lock(&mutex);
        int temp = counter;
        temp = temp + 1;
        counter = temp;
        pthread_mutex_unlock(&mutex);
    }
    printf("Thread %d finished. Counter = %d\n", id, counter);
    return NULL;
}
```

6. In the `main` function, after waiting for all threads to finish, destroy the mutex using `pthread_mutex_destroy`.

```
pthread_mutex_destroy(&mutex);
```

7. Compile your program using the `gcc` compiler.

```
gcc -o Task2 Task2.c -lpthread
```

8. Run your program and observe the output. The final value of the counter should now be correct.

```
./Task2
```

9. Compare the output with the original `RaceCondition.c` program and make sure the race condition is resolved.

Remember to free any dynamically allocated memory and destroy any pthread objects before exiting the program. This includes pthreads, mutexes, and condition variables.

5.3 Task 3: Concurrency Control: Using Mutex Locks to Synchronize Deposits from Multiple ATMs (40%)

In this task, you will work with a simple C program that emulates multiple ATMs depositing money into a shared account concurrently. This is a simple model for a real-world situation where multiple threads (in this case, ATMs) need to update a shared resource (the account balance) at the same time. You will use a mutex to ensure that only one thread updates the account balance at any given time, preventing race conditions.

5.3.1 Code Functionality

The code consists of a shared `account_balance` variable, which is accessed by multiple threads. Each thread represents an ATM that deposits a certain amount of money into the account balance multiple times.

The `deposit_money` function is executed by each thread. It takes a `ThreadArgs` struct as an argument, which contains the deposit amount and the number of deposits for that ATM. The function then deposits the specified amount into the account balance the specified number of times. A mutex, `balance_mutex`, is used to ensure that only one thread updates the account balance at any given time.

In the `main` function, the number of ATMs (threads) is specified as a command-line argument. The `main` function then prompts the user to enter the deposit amount and the number of deposits for each ATM on a single line. Each ATM thread is created and receives a `ThreadArgs` struct as an argument to the `deposit_money` function. This allows each ATM to deposit a different amount of money a different number of times into the account balance.

After all the ATMs have finished depositing money, the `main` function prints the final account balance and destroys the mutex.

5.3.2 Instructions

This task will help you understand how to use a mutex to synchronize access to a shared resource by multiple threads. Although in this example we emulate multiple users by taking user inputs for the sake of understanding, in a real-world application, the ATMs would be operating concurrently and continuously, making the use of a mutex essential to prevent race conditions and ensure the correct operation of the program.

5.3.3 Requirements

Download the `Task3.c` template file from Moodle.

Complete the following requirements in the `Task3.c` file:

1. **Lock the Mutex:** Before depositing money into the account, you need to lock the mutex to ensure that only one thread is accessing the account balance at a time. Use the `pthread_mutex_lock` function to lock the `balance_mutex`.
2. **Deposit Money:** Deposit the specified amount into the `account_balance`. You need to perform this operation multiple times, as specified by the `num_deposits` variable.
3. **Unlock the Mutex:** After depositing the money, you need to unlock the mutex to allow other threads to access the `account_balance`. Use the `pthread_mutex_unlock` function to unlock the `balance_mutex`.
4. **Initialize the Mutex:** Before creating the threads, you need to initialize the `balance_mutex`. Use the `pthread_mutex_init` function to initialize the `balance_mutex`.
5. **Create Threads:** Create a thread for each ATM. Use the `pthread_create` function to create each thread. Pass the `deposit_money` function and the appropriate arguments to each thread.
6. **Wait for Threads:** Wait for all the threads to finish. Use the `pthread_join` function to wait for each thread to finish its execution.
7. **Destroy the Mutex:** After all the threads have finished, you need to destroy the `balance_mutex`. Use the `pthread_mutex_destroy` function to destroy the `balance_mutex`.

Compile and run the code. You will need to specify the number of ATMs (threads) as a command-line argument. For example:

```
gcc -o Task3 Task3.c -lpthread
./Task3 3
```

The program will prompt you to enter the deposit amount and the number of deposits for each ATM. Enter the values on a single line, separated by a space. For example:

```
Enter the deposit amount and the number of deposits for ATM 0 (e.g. 10 1000): 10 1000
Enter the deposit amount and the number of deposits for ATM 1 (e.g. 10 1000): 20 500
Enter the deposit amount and the number of deposits for ATM 2 (e.g. 10 1000): 50 200
```

The program will then run and print the final account balance.

Verify that the final account balance is correct and that there are no race conditions or other concurrency issues.

Submit the completed `Task3.c` file along with other tasks for submission.

5.4 Wrapping Up

Please upload all your work to the Moodle submission link, which should include your .c source files and your responses to any non-coding tasks. Please note that non-code answers should be provided in plain text files. There is no need to include the compiled executable programs. Ensure to upload all the necessary files before the conclusion of your lab class. Any delay in submitting these files to Moodle will result in a late penalty.