ECE 434: Intro to Computer Systems -- Spring 2019 Instructor: Maria Striki

Homework 3 (74 points, 20+8+5+15+8+6+14)

Issue Date: Sun April 14th 2019 Due Date: Fri April 26th 2019

Note: Handled by groups of 4 OR 5 students, submit one document per group.

Your Names and Contribution per student:

Problem 1: (20 points)

In your textbook the Dining Philosophers Problem is solved with the use of Monitors but the solution in your textbook is not complete. Please:

1) **(7 pts)** Write the complete code for the Dining Philosophers Problem using Monitors and run it.
   Please note: Monitor mechanisms are currently (April 2019) defined only in Java and C++, and possibly a few more OO programming languages. So, if you are not using C++ with Linux you will not be able to define monitors. So, we will be doing the following:
   a) Those of you that can use a monitor directly are free to implement the deadlock-free solution proposed in your textbook (page 228), compile it, and run it.
   b) Those that cannot use monitors directly, you could implement the monitor using SEMAPHORES as suggested in your textbook (pages 229-230).
   c) If you select not to use monitors at, then implement a solution based entirely on condition variables.

2) **(7 pts)** Provide a solution to dining philosophers using only an alternative synchro mechanism:
   a) If you used an actual monitor before or solved the problem with only condition variables, now use POSIX locks and semaphores to provide a solution (preferably deadlock-free!).
   b) If use simulated a monitor using semaphores before, now use POSIX locks and condition variables to provide a solution (preferably deadlock-free).

3) **(6 pts):** Replace the while loop if you are using any with iterations on the number of attempts philosophers do in order to eat. You can try three samples: 1) 100 iterations, 2) 1,000 iterations, 3) 1,000,000 iterations. Time the execution of your programs and see which performs better. You may associate performance with the number of iterations. Do you see any pattern? Which method is better than the other, if there is a clear winner? If there is, why yes, if not, why not?

Solution:

Problem 2: (8 pts)

1. **(4 pts)** Show (pseudo-code) how to implement the `wait() and signal()` semaphore operations in multiprocessor environments using `test and set()` instruction. The solution should exhibit minimal busy waiting.

2. **(4 pts)** Describe (pseudo-code) how the `compare and swap()` instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.

Solution:

Problem 3: (5 points)

A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: The sum of all unique numbers associated with all the processes currently accessing the file must be less than *n*. Write a monitor (pseudo-code) to coordinate access to the file.

Solution:

Problem 4: (15 points)

In a corporate environment manageres work separately than office suppliers. Managers work in their office for a window of time (`work_actual()`) and take breaks periodically outside (`time_off()`). The suppliers' personnel never enter the office while there are still managers there and vice versa. The managers are represented by threads that execute function `manager()` and the office supplier personnel are represented by threads which execute function `supplier()`.

```
void manager () {
      while (;) {
      ….
      work_actual ();
      ….
      time_off ();
      }
}
```

```
void supplier () {
      while (;) {
      ….
      supply ();
      ….
      time_off ();
      }
}
```

1. **(6 pts)** You are requested to implement synchronization schemes that respect the working mode described above, in the dotted spaces. You may use: `signal()` and `wait()` calls to properly initialized semaphores and any shared variables among the required threads. Please justify if there is any possibility of starvation in the solution you provided.
2. **(5 pts)** Modify the synchronization scheme you built for above so that at most N managers can exist in the office at the same time, hence at most N threads execute function `work_actual()` simultaneously.
3. **(4 pts)** Assume that more suppliers' personnel is hired, who can all work together at the same time in the office, as long as no employees are present. Modify the synchro scheme issued in 2) so that at the office there can be either: at most N managers or at most M supplier personnel.

Solution:

Problem 5: (8 points)

Implement a synchronization scheme which simulates the process of the travelers boarding in planes through a waiting room of capacity N. New travelers arrive continuously, each of which executes function `traveler()`. Every traveler that enters the waiting room must do checking (ticket and passport check): execute function `cross-check()`. When there are N travelers to have completed `cross-check()`, then all together leave the waiting room and take the bus for the plane, i.e., execute: `boarding_bus()`, leaving the waiting room empty, available for the next batch of travelers. You are requested to implement synchronization schemes that respect the working mode described above, in the dotted spaces:

```
void traveler(){
       . . .
       cross_check ();
       . . .
       boarding_bus ();
}
```

Solution:

Problem 6: (6 points)

Below is a version of the basic/original readers-writers code, as discussed in class. Does the code below work? If not, provide an interleaving across at least 2 threads, using only the line number of the program, of multiple threads, that proves that the code does not work. If yes, justify your answer.

```
1. int readcount = 0;
2. Semaphore mutex = 1;
3. Semaphore w_or_r = 1;
4.
5. writer {
6. P(w_or_r);
7. Write;
8. V(w_or_r);
9. }
10.
11. reader {
12. P(mutex);
13. readcount += 1;
14. if (readcount == 1) {
15. V(mutex);
16. P(w_or_r);
17. } else {
18. V(mutex);
19. }
20. Read;
21. P(mutex);
22. readcount -= 1;
23. if (readcount == 0)
14. V(w_or_r);
15. V(mutex); }
16. }
```

Solution:

Problem 7: (14 points)

Synchronization strategy of taxis and customers at the airport. Taxis arrive at the airport in order. If there are customers waiting, a taxi driver immediately picks up the one waiting customer who arrived **the earliest**. Otherwise, the driver waits in a taxi-queue. Similarly, an arriving customer checks if there are taxis waiting. If so, she gets into the taxi that arrived **the earliest**. Otherwise she waits in line. Each taxi only picks up one customer.

Each driver and customer will be a separate thread executing functions `taxi()` and `customer()` respectively. Function `taxi()` calls `done()` when a customer is picked up, while `customer()` calls `done()` when a taxi has been boarded.

You are requested to write code for the above two functions.

Assume that the number of drivers (say N) is infinitely recycled and that new customers keep arriving. Hence, you may assume that the variable associated with the taxi drivers is circular or you may assume that the counter associated with the customers is also circular.

```
lock c_lock, d_lock; // protect the shared variables of customer and driver
int driver_ticket = customer_ticket = -1;
```

Part 1: (4 pts) How will you implement the above problem: with condition variables, with semaphores, or you may use any of the above? Please FULLY justify your selection.
Part 2: (10 pts) Write code or pseudo-code using the synchronization primitive(s) selected above.

Solution: