Final Examination
CSc 422, Fall 2020

December 15, 2020

Instructions: Please do not put your name anywhere on this exam. The exam consists of 4 problems on 8 pages; please be sure you have the entire exam before starting. The exam is open notes and book, but closed neighbor. The point values of the problems vary; please take this into account when allocating your time. The exam is worth 100 points.

Be concise and indicate clearly what your answer is. Presentation and simplicity of your answers may affect your grade. Please answer each question on a different page and submit to Gradescope when you are done.

Problem 1: Warmup (20 points [2 each])

Do **not** explain your answers.

A. Which mechanism requires thread creation for each invocation on the server side: RPC or rendezvous?

B. Which message passing scheme—synchronous or asynchronous—allows the sender to continue after sending a message, even if the corresponding receive has not yet been invoked?

C. True/False: Monitors and semaphores have equal power in that they can solve exactly the same set of problems.

D. You want to execute functions `f` and `g` in parallel. Your friend observes that the same global variable is present in each function and concludes that the functions cannot be parallelized. Why is your friend's conclusion not necessarily correct?

E. A program takes 100 seconds when run on one core. When a second core is added, it takes 80 seconds. There are many potential reasons why the ideal time of 50 seconds was not achieved. Name one of those reasons.

F. In part two of your concurrent malloc assigment, what is synchronization (i.e., the lock) protecting from race conditions?

G. If a semaphore is used as a lock, to what should its initial value be set?

H. True/False: There is a duality between monitors and message passing.

I. True/False: With distributed hash tables, removing a node that stores many key/value pairs results in all of those key/value pairs moving to another node.

J. Suppose a program has a race condition, and the first time it is run the output is incorrect, but the second time it is run the output is correct. If the program is executed 1000 additional times, how many times are you **guaranteed** to get correct output?

Problem 2: Semaphores (25 points) [20/5]

Consider the following code:

```
int a, b, c, d;

readFromKeyboard(a);
d = 1;
b = a * d;
c = a + a;
print(b,c);
```

A. Using semaphores, rewrite this program as a collection of five threads, one for *each line of code*. Each thread should executes its line of code, along with performing any necessary synchronization. You must use semaphores to retain the *original sequential semantics* of the code (i.e., the output must be identical to the same program where all statements are executed sequentially). However you must also exploit the *maximum* amount of concurrency; when two statements can safely execute concurrently, you must let them do so. Do not worry about actually creating the threads; just label the threads T_1, T_2, T_3, T_4, and T_5, and assume that they are created in the main function. You need not minimize the number of semaphores that you use.

B. Explain in one sentence why locks cannot solve this problem. (If you find yourself writing more than one sentence, you are probably writing down an incorrect answer.)

Problem 3: Monitors (25 points) [5 each]

The following is the solution to the readers/writers problem using monitors given in class. Note that this solution assumes Signal and Continue semantics. Each of parts (A-E) is independent.

```
monitor ReadersWriters
    int nr := 0, nw := 0
    cond okToRead, okToWrite;

    readEnter() {
        while (nw > 0) {
            Wait(okToRead);
        }
        nr++;
    }

    readExit() {
        nr−−;
        Signal(okToWrite);
    }

    writeEnter() {
        while (nw > 0 or nr > 0) {
            Wait(okToWrite);
        }
        nw++;
    }

    writeExit() {
        nw−−;
        Signal(okToWrite);
        Broadcast(okToRead);
    }

end monitor
```

A. Suppose that a reader thread, T_r, is executing in the middle of the `while` loop in `readEnter`, but there is then a context switch to a writer thread, T_w, whose first action is to invoke `writeEnter`. Explain precisely why T_w cannot start executing in `writeEnter`.

B. Suppose that a writer thread, T_w, is executing in the middle of the `while` loop in `writeEnter`, but there is then a context switch to a reader thread, T_r, whose first action is to invoke `readEnter`. Explain precisely why T_r cannot start executing in `readEnter`.

C. Explain precisely why signaling a writer **and** broadcasting to all readers in `writeExit` does not violate the constraint that there cannot be a reader and a writer in the database simultaneously.

D. If the `or` is changed to `and` in `writeEnter`, the solution is no longer correct. Explain why and do so precisely.

E. If nr is initialized to 1 instead of 0, the solution is no longer correct. Explain why and do so precisely.

Problem 4: Message Passing and Deadlock (30 points) [5 each]

Recall that with message passing programs, `Send` and `Receive` can be *asynchronous* (non-blocking) or *synchronous* (blocking). For this problem we assume that `Receive` is **always** synchronous (blocking).

For each of the following six parts (A-F), the integer variables x, y, and z, and w have initial values 1, 2, 3, and 4, respectively. In the message-passing pseudocode, the first parameter to `Send` is the destination, and the first parameter to `Receive` is the source. If the source of a `Receive` on process P is specified as `ANY_SOURCE`, then (1) this `Receive` is matched by a `Send` from any other process whose destination is process P, and (2) if multiple messages arrive at a destination that executed a `Receive` using `ANY_SOURCE`, whichever message arrives first is received first. (In other words, `ANY_SOURCE` works exactly as it did in your MPI programming assignment.) Finally, `Barrier` has the same semantics as discussed in class—every process must reach a barrier before any can proceed.

The second parameter is the value being sent (for `Send`) or the variable (i.e., l-val) into which the data is placed (for `Receive`). Another way to think of this is that the second parameter to `Receive` is being passed by reference.

A. In the code below, can deadlock occur if `Send` is **asynchronous (non-blocking)**? If so, explain how deadlock occurs. If not, explain why not and indicate the final values of x, y, and z.


```
P1:
  Receive(P2, x)   // the integer received from P2 is placed into variable x
  Send(P2, x)      // the value of x is what is sent to P2
  Send(P3, x)
  Barrier()
P2:
  Send(P1, y)
  Receive(P1, y)
  Barrier()
P3:
  Receive(P1, z)
  Barrier()
```


B. Consider the same code from part (A). Can deadlock occur if `Send` is **synchronous (blocking)**? If so, explain how deadlock occurs. If not, explain why not and indicate the final values of x, y, and z.

C. In the code below, can deadlock occur if `Send` is **asynchronous (non-blocking)**? If so, explain how deadlock occurs. If not, explain why not and indicate the final values of x, y, and z.

```
P1:
  Send(P2, x)
  Receive(P2, x)
  Send(P3, x)
  Barrier()
P2:
  Send(P1, y)
  Receive(P1, y)
  Barrier()
P3:
  Receive(P1, z)
  Barrier()
```


D. Consider the same code from part (C). Can deadlock occur if `Send` is **synchronous (blocking)**? If so, explain how deadlock occurs. If not, explain why not and indicate the final values of x, y, and z.

E. In the code below, can deadlock occur if `Send` is **asynchronous (non-blocking)**? If so, explain how deadlock occurs. If not, explain why not and indicate the final values of x, y, z, and w. (If there are multiple possibilities, list them all.)


```
P1:
  Send(P2, x)
  Send(P3, x)
  Barrier()
P2:
  Receive(ANY_SOURCE, y)
  Receive(P1, w)
  Barrier()
P3:
  Send(P2, z)
  Receive(P1, z)
  Barrier()
```

F. In the code below, can deadlock occur if `Send` is **asynchronous (non-blocking)**? If so, explain how deadlock occurs. If not, explain why not and indicate the final values of x, y, z, and w. (If there are multiple possibilities, list them all.)

```
P1:
  Send(P2, x)
  Send(P3, x)
  Barrier()
P2:
  Receive(ANY_SOURCE, y)
  Receive(ANY_SOURCE, w)
  Barrier()
P3:
  Send(P2, z)
  Receive(P1, z)
  Barrier()
```