Final Exam Answers – CS 343 Winter 2019

Instructor: Peter Buhr

April 15, 2019

These are not the only answers that are acceptable, but these answers come from the notes or lectures.

1. (a) **2 marks** A buffer smoothes out small, temporary differences in speed between adding and removing threads so them seldom block.

   The speed of the adding and removing threads must be virtually equal or the buffer is always full or empty.

   (b) **2 marks** Synchronization is required to prevent adding/removing when the buffer is full/empty. Mutual exclusion is required to atomically add/remove elements at each end of the buffer.

   (c) **1 mark** Baton passing conceptually passes the lock when unblocking to achieve barging prevention.

   (d) **2 marks**

   - both readers enter ⇒ 2:00 reader reads data that is **stale**; should read 1:30 write
   - writer enters and overwrites 12:30 data (never seen) ⇒ 1:00 reader reads data that is too **fresh** (i.e., missed reading 12:30 data)

   (e) **2 marks** No, FCFS (FIFO)

   (f) **2 marks** The V on the read bench (semaphore) is remembered by the semaphore counter, so when the reader finally restarts and P's on the read bench, it does not block.

2. (a) **2 marks** A race condition occurs when there is missing synchronization or mutual exclusion. It is hard to locate because the program runs but problems do not occur immediately because of non-determinism.

(b) **10 marks** 2 each

i. 4, the quadrants over which the cars drive

ii. 4, right hand turns

iii. 2, driving through the intersection

iv. 3, making a left-hand turn.

v. 4, the cars move into their quadrant simultaneously

(c) i. **1 mark** *mutual exclusion* deadlock

ii. **3 marks**

```
L1.P()              1   L1.P()
    R1
    L2.P()          1       L2.P()
        R1 & R2     1           R2      // access resource
                                R2 & R1 // access resource
```

iii. **2 marks** Resource utilization is reduced because task2 holds R1 longer than necessary.

(d) i. **4 marks**

	P1	P2	P3	P4
Maximum Needed	2	7	5	8
Current Acquired	2	2	0	7
Needed to Max.	0	5	5	1

11	Total Resources
-11	Used
0	Available for allocation
P1	0
	2
P4	1
	9

P2	4	P3	6
	11		11
P3	4	P2	4
	9		11

The state is safe as there are 2 sequences of execution that are safe.

ii. **3 marks**

	P1	P2	P3	P4
Maximum Needed	2	7	5	8
Current Acquired	2	4	1	4
Needed to Max.	0	3	4	4

11	Total Resources
-11	Used
0	Available for allocation
P1	0
	2

The state is NOT safe as there are insufficient resources for any process to execute so no sequence of execution is possible after this point.

3. (a) **11 marks** 3(a)v is 2 marks

   i. e

   ii. a and b called X, c called Y

   iii. d on A; g, h on B

   iv. mutex queues are auxiliary queues to allow O(1) access for the accept statement

   v. tasks f accepts a mutex queue with e at the front, and e does an accept
   tasks e and f were on a condition variable and both were signalled

   vi. signalling task

   vii. signalled task or a calling task

   viii. d (or signalled task)

   ix. signalling task, and tasks g h are moved to the A/S stack

   x. a enters, and the accepting task, e and f are on A/S stack

   (b) **2 marks** Yes, when the accepted call raises an exception, the acceptor receives the implicit RendezvousFailure from the caller.

   (c) **2 marks** μC++ monitors prevent barging by giving the acceptor/signaller (A/S) stack highest priority before looking at the calling queue (C < W < S).

   (d) **3 marks** advantage: no heap allocation to create a node and no data copy to the node
   disadvantage: link fields occupy space in the data and may be used infrequently or not at all

4. (a) **1 mark** Without a stack, a thread has no where to start execution.

   (b) **2 marks**

```
_Accept( m1, m2 ) S1 ⇒ _Accept( m1 ) S1; or _Accept( m2 ) S1;
```

   (c) **3 marks**

```
  _Task T {
    public:
1     void start() {}
1     void main() {
1         _Accept( start );      // 1st line
          . . .
```

   (d) **1 mark** monitor

   (e) **1 mark** The task's thread needs to do work or why create it, and the concurrency is inhibited for the caller.

   (f) **1 mark** The administrator never makes a blocking call (calls out).

   (g) **2 marks** An asynchronous call *returning a result* needs a mechanism (future) to match a completed result with the calling client.

5. (a) **1 mark** Caching transparently hides the latency of accessing main memory.

   (b) **1 mark** Cache coherence ensures a shared value is uniformly updated across the cache giving a consistent view of values.

   (c) **2 marks**

   ```
   Insert = true; // W
   Data = i; // W
   ```

   Allows reading of stale data.

   (d) **2 marks**

   i. Declaration qualifier **volatile** prevents variables from being hidden in registers.

   ii. A shared flag is loaded into a register and checked there, hence it is impossible to see the flag change.

   (e) **2 marks** A counter is added to count pushes, which the CASD saves atomically. The counter ensures the second push of A in ABA has a different count value from the first push.

   (f) **2 marks** Threads are declared but a join member is used rather than deallocation for termination synchronization.

6. (a) **8 marks**

```
L2:
1    if ( voters < group ) _Throw Quorum();

L3:
1    try {
1        for ( ;; ) {
1            _Accept( done ) {
1              if ( voters < group ) break;
1            } or _Accept( vote ) break;
         } // for
1    } catch(uMutexFailure::RendezvousFailure &) {}
1    if ( voters < group ) _Throw Quorum();
```

```
L2:
1    if ( voters < group ) _Throw Quorum();

L3:



// same minus last line





L5:
1    if ( voters < group ) _Throw Quorum();
```

(b) **6 marks**

```
L1:
1    uCondition bench;

L2:
1    if ( voters < group ) _Throw Quorum();

L3:
1    bench.wait();



L5:
1    bench.signal();
1    if ( voters < group ) _Throw Quorum();

L6:
1    if ( voters < group ) bench.signal();
```

```
L1:
1    uCondition bench;

L2:
1    if ( voters < group ) _Throw Quorum();

L3:
1    bench.wait();
1    bench.signal();
1    if ( voters < group ) _Throw Quorum();

L4:
–    bench.signal();

L6:
1    if ( voters < group ) bench.signal();
```

(c) **7 marks**

```
L1:
1    AUTOMATIC_SIGNAL;

L2:
1    if ( voters < group ) _Throw Quorum();

L3:
1    WAITUNTIL( numVotes == 0, , );
1    if ( voters < group ) _Throw Quorum();

L5:
1    RETURN(talliedResult);

L6:
1    if ( voters < group ) numVotes = 0;
1    RETURN();
```

```
L1:
1    AUTOMATIC_SIGNAL;

L2:
1    if ( voters < group ) _Throw Quorum();

L3:
2    WAITUNTIL( voters < group || numVotes == 0, , );
1    if ( voters < group ) _Throw Quorum();

L5:
1    RETURN(talliedResult);

L6:

1    RETURN();
```

7. **29 marks**

```
        void flush( bool kind ) {
1           for ( int i = 0; i < votes.size(); i += 1 ) {          // flush
2               votes[i]->ftour.exception( kind ? new Quorum : new Closed );
1               delete votes[i];
            } // for
1           votes.clear();
        }
        void main() {
1           for ( ;; ) {
1               _Accept( ~TallyVotes ) {                            // shutdown ?
1                   break;
1               } or _Accept( done ) {                              // voter leaves
1                   voters -= 1;
1                   if ( voters < group && votes.size() ) {         // failure ?
1                       flush( true );                              // Quorum failure
                    }
1               } or _Accept( vote ) {                              // voter
1               } or _Accept( tour ) {                              // guide
                }

1               if ( ! wguides.empty() && votes.size() >= group ) { // guide and group ?
1                   for ( int i = 0; i < group; i += 1 ) {          // compute rank
1                       add( votes[i]->ballot );
                    }
1                   gtour = tally();                                // compute vote
1                   for ( int i = 0; i < group; i += 1 ) {          // put vote in futures
1                       votes[i]->ftour.delivery( gtour );
1                       delete votes[i];
                    }
1                   votes.erase( votes.begin(), votes.begin() + group ); // shorten
1                   wguides.signalBlock();                          // unblock guide
1                   reset();                                        // reset vote counters
                }
            }
            // Shut down and tell the tourists/guides to go home
1           closed = true;
1           for ( int i = 0; i < guides; i += 1 ) {
1               if ( wguides.empty() ) _Accept( tour );
1               wguides.signalBlock();
            }
1           flush( false );                                         // Closed failure
        }
```