Final Exam Answers – CS 343 Fall 2015

Instructors: Peter Buhr and Ashif Harji

December 18, 2015

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

1. (a) **2 marks** *Staleness* occurs when a task reads data that is temporally in the past with respect to a current data value.

   Freshness occurs when a task reads data that is temporally in the future with respect to a current data value.

   (b) **3 marks**

   - Not if signalled task must implicitly reacquire the mutex lock before continuing.
   - $\Rightarrow$ signaller must release the mutex lock.
   - There is now a race between signalled and calling tasks, resulting in *barging*.

   (c) **1 mark** No value. The baton is fictitious.

   (d) **1 mark** Multiple threads must acquire resources in the same order to prevent deadlock.

   (e) **1 mark** A cycle in a process graph, does not imply a deadlock if any resource has multiple instances.

   (f) **2 marks** The preemption victim must be restarted (but where) or killed.

2. (a) **6 marks**

```
1   shared list/queue<int> q; // shared queue of integers

    producer task                    consumer task

1   region q {                       region q {
1     await q.size() < 10;           1   await ! q.empty();
1       // add item to queue         1     // remove item from queue
    }                                }
```

   (b) **2 marks** For signal the signalling task continues execution until it waits or exits, and the signalled task is delayed (on the Acceptor/Signaller stack).

   For signalBlock the signalling task is delayed (on the A/S stack), and the signalled task continues execution until it waits or exits.

   (c) **4 marks** If a writer is using the resource (wcnt > 0), achieve cooperation by preventing all calls except endWrite, after which the writer has finished using the resource.

   If a reader is using the resource (rcnt > 0), achieve cooperation by preventing all calls except N calls to endRead, after which all readers have finished using the resource.

   (d) **2 marks** There is a fixed (bounded) number of restarted tasks that check their predicates before new tasks can enter.

   (e) **2 marks** Barging occurs when a calling task acquires the monitor *before* a signalled task.

   (f) **1 mark** Spurious wakeup means a task blocked on a lock can unblock without being released or signalled.

3. (a) **2 marks** The long form of the **_Accept** statement is necessary to add **_When** clauses and action statements onto individual **_Accept** clauses OR

```
_When( C1 ) Accept( m1 ) S1; or _When( C2 ) Accept( m2 ) S2;
```

   (b) **2 marks** It is important to have as little code as possible in the mutex members of a server task to increase concurrency of the client because the client spends as little time as possible performing server work.

   The mutex-member code goes into the task main of the server, so the server performs the administration work rather than the client.

   (c) **2 marks** The client has to be unblocked so it can raise the exception on its stack (rather than server's stack).

   (d) **2 marks** A task accepts its destructor to know when to terminate.

   Accepting the destructor is unusual because it behaves as a signal rather than a signalBlock, i.e., the acceptor runs first and then the acceptee (destructor) runs.

   (e) **1 mark** The general approach to increase client-side is turning synchronous calls into asynchronous calls.

   (f) **2 marks** The advantage of futures is that they do not have an explicit protocol because the server pretends to instantaneously return a computed a result.

   (g) **2 marks** The **_Select** statement allows a task to block waiting for multiple futures to become available depending on a complex relationships among the futures.

```
_Select( f1 || f2 && f3 );
```

4. (a) **3 marks** False sharing is when threads are accessing disjoint (non-shared) variables but the variables are actually shared on the same cache line.

   The sharing causes *cache bouncing* if each thread is writing to the variables.

   Aligning or separating (padding) the variables so they are on separate cache lines.

   (b) **2 marks** Lazy cache-consistency occurs because caches cannot be update instantly, allowing readers to see own write before acknowledgement so other threads continue to read stale data.

   (c) **1 mark** Declaration qualifier **volatile** prevents variables from being hidden in registers.

   (d) **1 mark** No, if a programming language does not have a memory model, concurrency cannot be implemented with a library approach.

   (e) **2 marks** advantage: lock-free has no deadlock OR hold resource on preemption
   disadvantage: not general to arbitrary critical section OR performance questionable.

   (f) **2 marks** Ada requeue mechanism is not as powerful as internal scheduling, because internal scheduling preserves data and execution state while blocked but requeue does not.

   (g) **2 marks** Go uses channels to support direct communication. Go uses a select statement to choose among a number of channels for data or block until data arrives.

5. (a) **2 marks** An address space is an area of memory addressed at 0, so addresses cannot be used among address spaces, which it the precursor to separate memories in a distributed system.

   (b) **1 mark** There is a problem passing *integer* values between different computers because the byte ordering of the integers OR big versus little endian.

   (c) **2 marks** One thread is designated the *root* and its sends and receives occur as written but all other threads do the opposite action (receive or send).

   (d) **2 marks** Message passing is used to carry the call arguments through the network to the server and carry the results back through the network to the client.

6. (a) i. **3 marks**

```
V()
1   counter += 1;
P()
1   if ( counter == 0 ) _Accept( V );
1   counter -= 1;
```

ii. **3 marks**

```
1   uCondition bench;
V()
-   counter += 1;
1   bench.signal();
P()
1   if ( counter == 0 ) bench.wait();
-   counter -= 1;
```

iii. **6 marks**

```
1   unsigned int tickets = 0, serving = counter; // prevent bargers
V()
1   serving += 1;
1   signalAll();
P()
1   unsigned int myticket = tickets;          // select ticket
1   tickets += 1;                             // advance ticket for next barger
1   while ( myticket >= serving ) wait();     // my turn to proceed ?
```

iv. **3 marks**

```
1   AUTOMATIC_SIGNAL;
V()
-   counter += 1;
1   RETURN();
P()
1   WAITUNTIL( counter > 0, , );
-   counter -= 1;
```

(b) **2 marks** Yes, it works. The P-waiter leaves first, and then the V-signaller, but that order is unimportant.

7. **23 marks**

```
    void fillShuttle( unsigned int noOfClients ) {
1       shuttles.signalBlock();                                  // get shuttle id
1       for ( unsigned int i = 0; i < noOfClients; i += 1 ) {
1           clients.front().delivery( shuttleId );
1           clients.pop_front();
        } // for
1       numClientsWaiting -= noOfClients;
    } // Coordinator::fillShuttle

    void main() {
1       for ( ;; ) {
1           _Accept( ~Coordinator ) {
1               break;
1           } or _Accept( timeUp ) {
1               if ( ! shuttles.empty() && numClientsWaiting > 0 ) fillShuttle( numClientsWaiting );
1           } or _Accept( checkIn ) {
1               if ( numClientsWaiting >= ShuttleSize ) fillShuttle( ShuttleSize );
1           } or _Accept( getRide ) {
1               if ( ! shuttles.empty() && numClientsWaiting >= ShuttleSize ) fillShuttle( ShuttleSize );
            } // _Accept
        } // for

1       shuttingDown = true;
        // can shutdown in any order
1       for ( unsigned int i = 0; i < NumClients; i += 1 ) {
1           if ( clients.empty() ) _Accept( getRide );
1           clients.front().exception( new Closed );
1           clients.pop_front();
        } // for
1       for ( unsigned int i = 0; i < NumShuttles; i += 1 ) {
1           if ( shuttles.empty() ) _Accept( checkIn );
1           shuttles.signalBlock();
        } // for
1       _Accept( timeUp );
    } // Coordinator::main
```