Final Exam Answers – CS 343 Fall 2018

Instructor: Peter Buhr

December 17, 2018

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

1. (a) i. **5 marks**

```
class Lock {
1   unsigned int tickets, serving;
  public:
1   Lock() : tickets( 0 ), serving( 0 ) {}
    void acquire() {                          // entry protocol
1       int ticket = fetchInc( tickets );     // obtain a ticket
1       while ( ticket != serving ) {}        // busy wait
    }
    void release() {                          // exit protocol
1       fetchInc( serving );
    }
};
```

   ii. **1 mark** If 4+ billion (assume 4 byte integers) tasks arrive simultaneously, the tickets overflow and threads get the same ticket.

   (b) i. **3 marks**
   - there is exactly one baton
   - nobody moves in the entry/exit code unless they have it
   - once the baton is released, cannot read/write variables in entry/exit

   ii. **1 mark** 0 bytes, there is no actual baton

   (c) i. **2 marks** A time-slice between the V and P can result in a task *barging or staleness* so waiting is in non-temporal (non-FIFO) order.

   ii. **2 marks** Member Xwait.P(entry), which atomically blocks on Xwait and unlocks entry.

   (d) **2 marks** When the chair is empty, tasks at the front of the bench are unblocked until there is a writer that cannot enter. This writer waits in the chair, and the chair is always unblocked (priority) before the bench.

2. (a) **5 marks**
   i. There exists more than one shared resource requiring mutual exclusion.
   ii. A process holds a resource while waiting for access to a resource held by another process (hold and wait).
   iii. Once a process has gained access to a resource, the runtime system cannot get it back (no preemption).
   iv. There exists a circular wait of processes on resources.
   v. These conditions must occur simultaneously.

   (b) **2 marks** The angels are in a livelock because, after the humans leave, and a cardboard is used to cover one of the angels eyes, it can move and then so can the other angels.

   The angels are not holding any resource or waiting for a resource (no hold and wait cycle).

   (c) **2 marks** Order resources and allocate resources in that order to prevent a *hold-and-wait cycle*.

   (d) **2 marks** should release some resources, should not block or busy wait

3. (a) **2 marks** A monitor solution cannot allow *simultaneous insert/remove* to an appropriate buffer because of the *mutual exclusion property*.

   (b) **3 marks** SHARED declarations become private monitor declarations, REGION statements become public monitor members, AWAIT clauses become **_Accept** or signal/wait statements.

   (c) **1 mark** External scheduling is simpler because unblocking (signalling) is implicit.

   (d) **2 marks**

```
_Accept( M1, M2 ); // OR
_Accept( M1 ); _Accept( M2 ); // AND
```

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

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

   (f) **3 marks** A task calls into monitor M1 and monitor M2, and waits in M2, releasing M2's monitor lock but not M1's monitor lock. Because M1's monitor lock is not released, a signalling task may not be able to enter M2 to signal the waiting task, leading to a *deadlock*.

   (g) **2 marks** Too *confusing* because either the signalled or signaller task can *randomly continue* in the monitor.

   (h) **2 marks** only one condition variable, barging

   (i) **1 mark** No!!!

4. (a) **2 marks** Without mutual-exclusion, multiple thread can enter the type object, including the task thread, which means the type's data members must be protected by explicit locks.

   (b) **4 marks**

```
    void mem() {
1       ... _Throw E(); ... // cause RendezvousFailure
    }
    void main() {
1       try {
1           ... _Accept( mem ); ...
1       } catch( uMutexFailure::RendezvousFailure ) {} // deal with RendezvousFailure
    }
```

   (c) **2 marks** The rendezvous is postponed or subdivided, and the server must fulfill the rendezvous later and unblock the client.

   (d) i. **2 marks _Accept** should block, run the destructor, and then unblock, but the object is gone (deleted).

   ii. **2 marks _Accept** continues running and the destructor call is postponed on the A/S stack.

   (e) **2 marks** Accessing a cancelled future raises an exception.
   There is race condition between the canceller and the processing/accessing of the future.

5. (a) **2 marks** disk/memory, memory/registers

   (b) **2 marks** The heap memory-allocator may place variables x and y on the *same cache line* resulting in *false sharing*.

   (c) **2 marks**

```
data = Data;            // W
while ( ! Insert );     // R
Insert = false;
```

   Allows reading of uninserted data.

   (d) **2 marks** All data to be processes must be copied from the CPU to the GPU, and all results copied back.

   (e) **3 marks** requeue cancels the current call (request) to a task, reschedules the call on another (usually non-public) mutex member of the task, and accepts it later.

   (f) **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.

   (g) **1 mark** implicit concurrency system

6. (a) **8 marks** There is duplicate code, which is only counted once across the solutions.

```
P()
3   if ( cnt == 0 ) for (;;) _Accept( V ) break; or _Accept( tryP );
1   cnt -= 1;
tryP()
1   if ( cnt == 0 ) return false;
1   cnt -= 1;
1   return true;
P( Semaphore s )
1   s.V();
1   P(); // or duplicate P() code
V()
1   cnt += 1;
```

   (b) **7 marks**

```
1   uCondition bench;
P()
1   if ( cnt == 0 ) bench.wait();
-   cnt -= 1;
tryP()
1   if ( cnt == 0 ) return false; // or same as for external
-   cnt -= 1;
-   return true;
P( Semaphore s )
1   s.V();
-   P(); // or duplicate P() code
V()
-   cnt += 1;
1   bench.signal();
```

7. **25 marks**

```
void MapleLeafTaxi::main() {
1    Taxi * taxitasks[NoOfTaxi];

1    for ( int id = 0; id < NoOfTaxi; id += 1 ) {
1        taxitasks[id] = new Taxi( *this, id );              // allocate taxis
     }
1    for ( ;; ) {
1        _Accept( close ) {
1            break;
1        } or _Accept( getClient, getTaxi ) {
1            if ( taxis.size() > 0 && clients.size() > 0 ) {
1                LocnClient *n = clients.front();
-                clients.pop_front();
1                xclient = n->x; yclient = n->y;
1                list<LocnTaxi *>::iterator nearest = nearestTaxi( n, taxis ); // find closest taxi
1                n->ftaxi.delivery( (*nearest)->id );
1                delete n;                                   // allocated in getTaxi
1                (*nearest)->idle.signalBlock();
1                taxis.erase( nearest );
             }
         }
     }
1    osacquire( cout ) << "Closed for the day." << endl;
1    for ( int i = 0; clients.size() != 0; i += 1 ) {            // notify potentially waiting clients
1        LocnClient *client = clients.front();
-        clients.pop_front();
1        client->ftaxi.exception( new Closed );              // raise exception
1        delete client;                                      // allocated in getTaxi
     }
1    closed = true;                                          // tell taxi tasks to go home
1    for ( int i = 0; i < NoOfTaxi; i += 1 ) {
1        if ( taxis.empty() ) _Accept( getClient );          // wait for taxi
1        taxis.front()->idle.signalBlock();
-        taxis.pop_front(); // unblock with closed
     }
1    for ( int i = 0; i < NoOfTaxi; i += 1 ) delete taxitasks[i]; // delete taxis
```