Final Exam Answers – CS 343 Winter 2013

Instructor: Bernard Wong

April 22, 2013

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

1. (a) **2 marks** Static/dynamic call, and static/dynamic return.

   (b) **1 mark** In termination, when a handler returns, it performs a static return. In resumption, when a handler returns, it is a dynamic return (goes back to exception throw site).

   (c) **2 marks** Only one OS thread means it can only make use of one core. No parallelization.

   (d) **2 marks** (10 + 50 + 25 + 15) / (10 + 50/5 + 25 + 15/3) = 2

   (e) **1 mark** Ticket values cannot increase indefinitely. Only probabilistically correct.

   (f) **2 marks** No. Breaks rule 4 (indefinite postponement). If both threads enter the loop at the same time, they will both spin forever in the while loop waiting for the other to drop their intent declaration.

   (g) **1 mark** If intents are not dropped in reverse order in the exit protocol, intents of waiting tasks at the base of the tree are overwritten.

2. (a) **6 marks** Yes, it allows the request.

total available resources

R1	R2	R3	R4	
6	12	4	2	(TR)

current available resources

	2	2	2	2	$(CR = TR - \sum C_{cols})$
T3	0	0	2	1	$(CR = CR - N_{T3})$
	3	4	2	2	$(CR = CR + M_{T3})$
T1	1	0	2	1	$(CR = CR - N_{T1})$
	5	10	3	2	$(CR = CR + M_{T1})$
T2	4	3	3	0	$(CR = CR - N_{T3})$
	6	12	4	2	$(CR = CR + M_{T3})$

   (b) **2 marks** Preemption victim must be restarted (but where) or killed.

   (c) **2 marks** An algorithm that *prevents* deadlock removes one of the conditions necessary for deadlock, thus ensuring that deadlock cannot occur. An algorithm that *avoids* deadlock might move into a potentially unsafe state, but the system prevents deadlock from occurring by refusing requests that would (conservatively) lead to deadlock.

3. (a) **2 marks** Different address space. Long pointers that include the location of the server.

   (b) **3 marks**

   - The send transmits a message to a receiving task and blocks until a reply is sent from the receiving task.
   - The receive accepts a message from any sending task and blocks the receiving task if there is no message.
   - The reply unblocks a sending task and may allow a message to be passed back to the sender.

   (c) **6 marks** A task dereferences the second node in stack into a temporary variable and attempts to make it the stack top.
   The task is preempted, and the top two nodes are popped, and the top node is pushed again.
   The preempted task restarts, and CAA removes the top node as it is the same as before the time-slice.
   But top is incorrectly set to the temporary value, which is no longer the second node in the list.
   CAV2 can enable the algorithm to keep a version number to distinguish between different versions.

   (d) **2 marks** The watchpoint used for LL/SC watches the memory location - in contrast to CAA, which only compares values. If there is any access to the location, even if the value ends up being unchanged, the SC instruction fails.

   (e) **2 marks** Can swap line 1 and 2 such that you race befor knowing intents, which leads to a mutual exclusion violation.

4. (a) **2 marks** The future is returned immediately from the server so the client can continue execution concurrently while the server performs the client's request.

   (b) **1 mark** A future is immutable because multiple threads may need to read its value at unknown times and in unknown order

   (c) **1 mark** Use a single semaphore (bench) on which tasks block in order of arrival (temporal order).

   (d) **2 marks** An advantage of the automatic-signal monitor is fast prototyping (or ease of use) and a disadvantage is poor performance.

   (e) **2 marks** The clients calling the server should not perform the server's work; hence, clients should block on entry or exit quickly to/from a mutex member and let the server's thread perform all administrative actions.

   (f) **3 marks**

```
if (count != 20 && count != 0)
    _Accept(insert, remove);
else if (count != 20)
    _Accept(insert);
else
    _Accept(remove);
```

   (g) **2 marks** The statement cannot be rewritten because the terminating **_Else** makes the **_Accept** non-blocking, and there is no other non-blocking version of **_Accept**.

5. (a) **6 marks**

```
_Monitor ReadersWriter {
    int rcnt, wcnt;
  public:
    ReadersWriter() : rcnt(0), wcnt(0) {}
    void endRead() {
        rcnt -= 1;
    }
    void endWrite() {
        wcnt = 0;
    }
    void startRead() {
        if ( wcnt > 0) _Accept( endWrite );
        rcnt += 1;
    }
    void startWrite() {
        if ( wcnt > 0) { _Accept( endWrite ); }
        else while ( rcnt > 0 ) _Accept( endRead );
        wcnt = 1;
    }
};
```

(b) **6 marks**

```
_Monitor ReadersWriter {
    int rcnt, wcnt;
    uCondition RWers, PriorityWriters;
    enum RW { READER, WRITER };
  public:
    ReadersWriter() : rcnt(0), wcnt(0) {}
    void startRead() {
        if ( wcnt !=0 || ! RWers.empty() || ! PriorityWriters.empty() ) RWers.wait( READER );
        rcnt += 1;
        if ( ! RWers.empty() && RWers.front() == READER ) RWers.signal();
    }
    void endRead() {
        rcnt -= 1;
        if ( rcnt == 0 ) {
            if ( ! PriorityWriters.empty() ) PriorityWriters.signal();
            else RWers.signal();
        }
    }
    void startWrite() {
        if ( wcnt != 0 || rcnt != 0 || ! PriorityWriters.empty() ) RWers.wait( WRITER );
        wcnt = 1;
    }
    void endWrite() {
        wcnt = 0;
        if ( ! PriorityWriters.empty() ) PriorityWriters.signal();
        else RWers.signal();
    }
    void startPriorityWrite() {
        if ( wcnt != 0 || rcnt != 0 ) PriorityWriters.wait();
        wcnt = 1;
    }
    void endPriorityWrite() {
        wcnt = 0;
        if ( ! PriorityWriters.empty() ) PriorityWriters.signal();
        else RWers.signal();
    }
};
```

6. (a) **6 marks**

```
TallyVotes( unsigned int group ) : group( group ), numVotes( 0 ), sum ( 0 ), tickets( 0 ) {}

bool vote( unsigned int id, bool ballot ) {
    int myticket = tickets / group;              // divide tickets into group-sized groups
    tickets += 1;
    // Bargers wait until their group is selected.
    while ( myticket != server ) bench.wait();

    numVotes += 1;
    sum += ballot ? 1 : 0;

    if ( numVotes < group ) {                    // all but last voter
        bench.wait();
        if ( numVotes == 1 ) {                   // last member of group ?
            server += 1;
            bench.broadcast();
        }
    } else {                                     // last voter
        talliedResult = sum > group / 2;         // compute result of vote
        sum = 0;                                 // reset for next vote
        if ( group == 1 ) { server += 1; }       // special case for single groups
            bench.broadcast();
    } // if
    numVotes -= 1;                               // uncount each leaving task
    return talliedResult;
}
```

(b) **6 marks**

```
bool vote( unsigned int id, bool ballot ) {
    numVotes += 1;
    sum += ballot;
    if ( numVotes < group ) {                    // all but last voter
        bench.wait();
    } else {                                     // last voter
        talliedResult = sum > group / 2;         // compute result of vote
        sum = 0;                                 // reset for next vote
    }
    bool saveTalliedResult = talliedResult;
    numVotes -= 1;
    bench.signal();                              // blocking
    return saveTalliedResult;
}
```

7. **25 marks**

```
void fillGame(PlayerSkill skill) {
    servers.signalBlock();
    for (unsigned int i = 0; i < NumPlayersPerGame; i += 1 ) {
        players[skill].front().delivery(serverId);
        players.pop_front();
    } // for
    numPlayersWaiting -= NumPlayersPerGame;
}

void main() {
    for ( ;; ) {
        _Accept( ~Coordinator ) {
            break;
        } or _Accept(timeUp) {
            if (!servers.empty() && numPlayersWaiting > NumPlayersPerGame) {
                servers.signalBlock();
                for (unsigned int i = 0; i < NumPlayersPerGame; ++i) {
                    for (unsigned int j = 0; j < 3; ++j) {
                        if (!players[j].empty()) {
                            players[j].front().delivery(serverId);
                            players[j].pop_front();
                            break;
                        }
                    }
                }
                numPlayersWaiting -= NumPlayersPerGame;
            }
        } or _Accept(checkIn) {
            for (int i = 0; i < 3; ++i) {
                if (players[i].size() > NumPlayersPerGame) {
                    fillGame(i);
                    break;
                }
            }
        } or _Accept(joinGame) {
            if (!servers.empty() && players[lastPlayerSkill].size() > NumPlayersPerGame) {
                fillGame(lastPlayerSkill);
            }
        } // _Accept
    } // for
```

```
    shuttingDown = true;
    // can shutdown in any order
    for ( unsigned int i = 0; i < NumClients; i += 1 ) {
        if (numClientsWaiting == 0) {
            _Accept( joinGame );
            players[lastClientSkill].front().exception(new Closed);
            players[lastClientSkill].pop_front();
            numClientsWaiting -= 1;
        } else {
            for (unsigned int i = 0; i < 3; ++i) {
                if (!players[i].empty()) {
                    players[i].front().exception(new Closed);
                    players[i].pop_front();
                    numClientsWaiting -= 1;
                    break;
                }
            }
        }
    }
    for ( unsigned int i = 0; i < NumServers; i += 1 ) {
        if (servers.empty()) _Accept( checkIn );
        servers.signalBlock();
    } // for
    _Accept( timeUp );
} // Coordinator::main
```