Midterm Answers – CS 343 Fall 2015

Instructors: Peter Buhr and Ashif Harji
November 4, 2015

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

1. (a) **1 mark** remove duplicate/priming code
   (b) **1 mark** Code in the **else** is logically part of the loop body not part of the loop exit.
   (c) **1 mark** flag variables
   (d) **2 marks** The Sequel's static return forces stack unwinding after the block on the stack containing the Sequal.
   (e) **2 marks** C longjmp is a direct stack transfer versus an unwinding because it does not have to execute destructors associated with objects allocated in intervening stack frames.
   (f) **2 marks** If a resumption exception is not handled during propagation, it is reraised as a termination exception (throw).

2. (a) **2 marks** A coroutine properly handles the class of problems that require state information to be retained between successive calls (e.g. finite-state machine).
   (b) **2 marks** For semi-coroutines, the starter is often the only resumer.
   For full-coroutines, starters always lead to uMain::main, which can deleted unterminated coroutines.
   (c) **2 marks** An unhandled exception raised by a coroutine raises the nonlocal exception UnhandledException at the coroutine's last resumer.
   (d) **2 marks** Starting the cycle requires each coroutine to know at least one other coroutine, and the problem is mutually recursive references:

```
fc x(y), y(x);
```

3. (a) **2 marks** 2, 3, 4
   (b) **2 marks** A user thread is scheduled by the language runtime and executed by a kernel thread, while a kernel thread is scheduled by the operating system and executed by a CPU.
   (c) **2 marks** Unbounded overtaking allows a low-priority thread to reenter the critical section any number of times until the high-priority thread declares its intent.
   (d) **3 marks**
      - T0 executes Line 1 ⇒ ::Last = T0
      - T1 executes Line 1 ⇒ ::Last = T1
      - T1 executes Line 2 ⇒ T1 = WantIn
      - T1 enters CS, because T0 == DontWantIn
      - T0 executes Line 2 ⇒ T0 = WantIn
      - T0 enters CS, because ::Last == T1
   (e) **2 marks** atomic read and write
   violate order and speed of execution

(f) **6 marks**

```
        class Lock {
1           int ticket, serving;
          public:
1           Lock() : ticket( 0 ), serving( 0 ) {}
            void entryProtocol() {
1               int myTicket = fetchDec( ticket );
1               while ( myTicket != serving );
            }
            void exitProtocol() {
1               fetchDec( serving );
            }
        };
```

The solution assumes the total number of tasks simultaneously using a ticket lock is less than the total number of values that can be represented by an integer (4 billion for most 4 byte integers).

4. (a) **1 mark** Barging is prevented by not reseting the InUse flag and/or not releasing the spinlock.

(b) **5 marks**

(i) spin lock : synchronization and mutual exclusion
(ii) owner lock : mutual exclusion
(iii) condition lock : synchronization
(iv) barrier : synchronization
(v) semaphore : synchronization and mutual exclusion

(c) **7 marks** One of:

```
L1 = L2 = L3 = L41 = L42 = 0;
COBEGIN
    BEGIN S1; V(L1); END;
    BEGIN S2; V(L2); END;
    BEGIN P(L1); S3; V(L3); END;
    BEGIN P(L3); P(L2); S4; V(L41); V(L42); END;
    BEGIN P(L41); S5; END;
    BEGIN P(L42); S6; END;
COEND;
```

```
L1 = L2 = L4 = 0;
COBEGIN
    BEGIN S1; S3; V(L1); END;
    BEGIN S2; V(L2); END;
    BEGIN P(L1); P(L2); S4; V(L4); END;
    BEGIN P(L4); V(L4); S5; END;
    BEGIN P(L4); V(L4); S6; END;
COEND;
```

```
L1 = L2 = L3 = L4 = 0;
COBEGIN
    BEGIN S1; V(L1); END;
    BEGIN S2; V(L2); END;
    BEGIN P(L1); S3; V(L3); END;
    BEGIN P(L3); P(L2); S4; V(L4); END;
    BEGIN P(L4); V(L4); S5; END;
    BEGIN P(L4); V(L4); S6; END;
COEND;
```

```
L1 = L2 = 0;
COBEGIN
    BEGIN S1; S3; P(L1); S4; V(L2); S5; END;
    BEGIN S2; V(L1); P(L2); S6; END;
COEND;
```

5. **22 marks**

```
        void main() {
1           char X, Y, Z, W;
            int cnt = 1;

1           X = ch;
1           for ( ;; cnt += 1 ) {
1               suspend();
1             if ( ch != X ) break;
            } // for

1           Y = ch;
1           suspend();
1           Z = ch;
1           for ( ;; cnt += 1 ) {
1               suspend();
1             if ( ch != Y ) break;
1               suspend();
1             if ( ch != Z ) {
1                   if ( ch != Y ) { _Resume Error() _At resumer(); return; }
1                   cnt -= 1;
1                   break;
                } // exit
            } // for

1           W = ch;
1           if ( Z == W ) { _Resume Error() _At resumer(); return; }
1           for ( ;; cnt -= 1 ) {
1             if ( cnt == 0 ) { _Resume Match() _At resumer(); return; }
1               suspend();
1             if ( ch != W ) { _Resume Error() _At resumer(); return; }
            } // for
        } // Grammar::main
```

Maximum 10 if not using coroutine state.

6. **31 marks**

```
1   #include <iostream>
    using namespace std;
    _Task DiagSymmetric {
        const int (*M)[10], row, cols;
        bool &result;
        void main() {
1           try {
1               _Enable {
1                   for ( int i = row + 1; i < cols; i += 1 ) {
1                       if ( M[row][i] != M[i][row] ) { result = false; return;          }
                    } // for
1                   if ( row != 0 && M[row - 1][row - 1] != M[row][row] ) { result = false; return; }
                } // _Enable
1           } catch( Stop ) {}
        } // DiagSymmetric::main
      public:
        _Event Stop {};                         // stop checking
        DiagSymmetric( const int M[][10], const int row, const int cols, bool &result ) :
1           M( M ), row( row ), cols( cols ), result( result ) {}
    }; // DiagSymmetric

    void uMain::main() {
1       int rows, cols;
        cin >> rows >> cols;
1       if ( rows != cols ) {
1           cerr << " Usage: matrix must be square." << endl;
            exit( EXIT_FAILURE );
        } // if
1       int M[rows][10], r, c;
1       for ( r = 0; r < rows; r += 1 ) {                   // read/print matrix
1           for ( c = 0; c < cols; c += 1 ) {
1               cin >> M[r][c];
1               cout << M[r][c] << ", ";
            } // for
1           cout << endl;
        } // for
1       cout << endl;
1       DiagSymmetric *workers[rows];
1       bool result = true;
1       for ( r = 0; r < rows - 1; r += 1 ) {               // create task to calculate rows
1           workers[r] = new DiagSymmetric( M, r, cols, result );
        } // for
1       for ( r = 0; r < rows - 1; r += 1 ) {               // wait for completion and delete tasks
1           if ( ! result ) {                               // if unsymmetric, try to stop other tasks
1               for ( int i = r; i < rows - 1; i += 1 ) {
1                   _Resume DiagSymmetric::Stop() _At *workers[i];
                } // for
1               for ( int i = r; i < rows - 1; r += 1 ) {
1                   delete workers[i];
                } // for
1               break;
            } // fi
1           delete workers[r];
        } // for
1       cout << "matrix is" << (result ? " " : " not ") << "diagonal-symmetric" << endl;
    } // uMain::main
```