Midterm Answers – CS 343 Fall 2016

Instructors: Peter Buhr and Aaron Moss

November 2, 2016

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

1. (a) i. **1 mark** duplicate code

   ii. **3 marks**

   ```
   1    for ( ;; ) {          // or while ( true )
   1        f( x );
   1      if ( ! Condition ) break;
            . . .
        }
   ```

   (b) i. **1 mark** recomputing the reason for loop termination

   ii. **2 marks**

   ```
        for ( ;; ) {
            . . .
   1        if ( i >= 10 ) { E2; break; }
            . . .
   1        if ( j >= 10 ) { E1; break; }
        }
   ```

   (c) **1 mark** multi-level exit or labelled break

   (d) **2 marks**

   i. Cannot loop (only forward branch) ⇒ only loop constructs branch back.

   ii. Cannot branch into a control structure.

   (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** A *routine call* is a direct transfer (routine address is known), while a *raise call* involves a dynamic search to locate the handler before it can be called.

2. (a) **2 marks** An *input coroutine* accepts a stream of values and consumes them (consumer).
   An *output coroutine* generates a stream of values for consumption (producer).

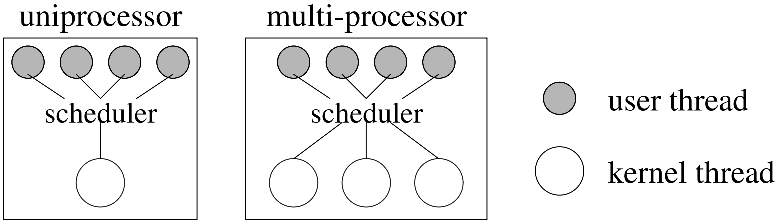
   (b) **2 marks** A *context switch* saves the execution state of a coroutine to make it inactive, and restores the execution state of another coroutine to make it active.

   (c) **2 marks** With multiple stacks, each stack cannot grow up to the program memory-size. Hence, the issue of bounded stack-size has to be addressed within each coroutine.

   (d) **2 marks** Non-local exceptions are initially disabled to allow a coroutine to complete initialization and install exception handlers before non-local exceptions are delivered.

   (e) **3 marks** cycle creation, executing around the cycle, returning to the root coroutine (cycle stopping)

3. (a) **4 marks**



(b) **4 marks** Syntax may vary as long as it makes sense.

```
1   COFOR( row, 0, rows,
1       subtotals[row] = 0;  // row is loop number
1       for ( int c = 0; c < cols; c += 1 ) {
1           subtotals[row] += matrix[row][c];
        }
    );
```

(c) **2 marks** A *critical section* is block of code that must be executed atomically. *Mutual exclusion* is code placed before/after a critical section to ensure only one thread is in the critical section.

(d) **2 marks** Peterson's algorithm assumes atomic write (assignment).
Peterson's algorithm does not create atomicity from expressions and control flow (out of thin air) but relies on pre-existing atomicity.

(e) **2 marks** *Bounded* overtaking prevents a thread from immediately reentering the critical section if another thread has declared its intent.

(f) **5 marks**

```
1   int Lock = OPEN; // shared
    void Task::main() { // each task does
1       int dummy = CLOSED;
        do {
1           Swap( Lock, dummy );
1       while( dummy == CLOSED );
        /* critical section */
1       Lock = OPEN;
    }
```

4. (a) **4 marks** *Spinning locks* have an unbounded loop that checks for completion of an event (synchronization) or acquiring access to a resource (mutual exclusion).
Blocking locks perform one check of an event or for access and block; unblocking occurs through cooperation from another thread setting the event or releasing the resource.

(b) **2 marks** Blocking locks do not eliminate spinning because they need mutual exclusion to protect state and queue operations, which can only be implemented using a spinlock.

(c) **4 marks**

```
1   lock.acquire();
1   try {
        . . .  // protected by lock
1   } _Finally {
1       lock.release();
    }
```

or

```
1   class RAII {
1       LockType &lock;
      public:
1       RAII( uOwnerLock &lock ) : lock( lock ) { lock.acquire(); }
1       ~RAII() { lock.release(); }
    };
```

(d) **2 marks** A blocking lock only gets one chance to test lock-state whereas spinning lock gets any number of chances. Therefore, must ensure the test is not missed for blocking lock.

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

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

or

```
L11 = L12 = L21 = L22 = L3 = L41 = L42 = 0;
COBEGIN
    BEGIN  S1;  V(L11); V(L12); END
    BEGIN  S2;  V(L21); V(L22); END
    BEGIN  P(L11); P(L21);  S3;  V(L3); END
    BEGIN  P(L22);          S4;  V(L41); V(L42); END
    BEGIN  P(L12); P(L41);  S5; END
    BEGIN  P(L3);  P(L42);  S6; END
COEND
```

5. **16 marks**

```
       void main() {
         line:
1          for ( ;; ) {
2              while ( ch == '\n' || ch == ' ' || ch == '\t' ) { // skip white space at start of line
1                  suspend();
               } // while
1              for ( ;; ) {                                // within a line...
1                  for ( ;; ) {                            // process block of text
1            if ( ch == '\377' ) break line;               // no more input, terminate ?
1                      cout << ch;                         // write non-blank characters
2            if ( ch == '\n' ) { suspend(); continue line; } // end of line, start new line ?
1                      suspend();
1                    if ( ch == ' ' || ch == '\t' ) break; // whitespace ending text ?
                   } // for
1                  for ( ;; ) {                            // compact intermediate whitespace
1                      suspend();
1                    if ( ! ( ch == ' ' || ch == '\t' ) ) break;
                   } // for
1                  if ( ch != '\n') cout << ' ';           // single blank between words
               } // for
           } // for
       } // main
```

Maximum 8 if not using coroutine state.

6. **27 marks**

```
1   #include <iostream>
    using namespace std;

    bool stop = false;                          // global variable: true => stop all work

    _Task EqualRows {
1       const int *row1, *row2, cols;
        void main() {
1           try {
1               _Enable {
1                   for ( int r = 0; r < cols; r += 1 ) {
1                       if ( row1[r] != row2[r] ) {
1                           stop = true;
1                           return;
                        } // if
                    } // for
                } // _Enable
1           } catch( Stop ) {
            } // try
        } // EqualRows::main
      public:
1       EqualRows( const int row1[ ], const int row2[ ], const int cols ) :
            row1(row1), row2(row2), cols(cols) {}
    }; // EqualRows

    void uMain::main() {
1       int rows, cols;
        cin >> rows >> cols;
1       int M[rows][cols], 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
            cout << endl;
        } // for
        cout << endl;

1       EqualRows *workers[rows - 1];           // N - 1 tasks
1       for ( r = 0; r < rows - 1; r += 1 ) {   // create task to calculate rows
1           workers[r] = new EqualRows( M[r], M[r + 1], cols );
        } // for
1       bool once = true;                       // only throw exceptions once
1       for ( r = 0; r < rows - 1; r += 1 ) {   // wait for completion and delete tasks
1           if ( once && stop ) {               // if unequal, try to stop other tasks
1               for ( int i = r + 1; i < rows - 1; i += 1 ) {
1                   _Resume EqualRows::Stop() _At *workers[i];
                } // for
1               once = false;                   // do not do this again
            } // if
1           delete workers[r];
        } // for
1       cout << "matrix does" << ( ! stop ? " " : " not ") << "have equal rows" << endl;
    } // uMain::main
```