Midterm Answers – CS 343 Winter 2019

Instructor: Peter Buhr

March 6, 2019

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

1. (a) **2 marks**

```
    for ( i = 0; ; i += 1 ) {          // linear search for key in list
1       if ( i == size ) { C1; break; }
1       if ( key == list[i] ) { C2; break; }
    }
```

   (b) **1 mark** Retain state from one inner lexical (static) scope to another.

   (c) **4 marks**

   - static call
   - dynamic call
   - static return
   - dynamic return

   (d) **2 marks** The **throw** raises a B, which is not caught by a D.
   μC++ raises a D.

   (e) **2 marks** When the raise site *cannot* continue, termination searches for a catch/handler that can *recover* and continue lower on the stack.
   When the raise site *can* continue, resumption searches for a catch/handler that can *fix up* and continue after the raise.

   (f) **2 marks** vector can use dynamic allocation and the heap is a point of lock contention because it is shared/serial resource among all threads.

2. (a) **2 marks** A coroutine allows a routine to suspend its execution rather than terminating (returning) to its caller.
   The caller can then resume the suspended routine rather than call it again from the top.

   (b) **2 marks** The stack does not grow.
   Set the stack to its maximum depth when the coroutine is created.

   (c) **2 marks** The first resume context switches (cocalls) to start the coroutine.
   A terminated coroutine context switches to its starter coroutine.

   (d) **3 marks**

   - program main creates ping and pong
   - program main starts ping; ping starts pong
   - ping and pong are in a cycle

   (e) **2 marks** Cannot modularize/call-routines because generator/iterator is stackless coroutine.

   (f) **2 marks** There is only one thread executing, which continues after the **_Resume**.
   Pass the thread to the coroutine by calling a member routine that does a resume.

3. (a) **2 marks** The other thread is simultaneously reading and sees the bits flicker in i or writing i and the bits become scrambled.

   (b) **1 mark** User threading has better performance because context switching does not cross the application/kernel (OS) boundary.

   (c) **3 marks** amount of concurrency, critical path among concurrency, scheduler efficiency

   (d) **1 mark** Yes

   (e) i. **1 mark** A thread may not enter the critical section successive times when the other thread does not want in.

   ii. **2 marks** Trick question converting alternation into a spinlock.

```
1   while( TestSet( ::Last ) == 0 ); // entry protocol
    CriticalSection();               // critical section
1   ::Last = 1;                      // exit protocol
```

   (f) **1 mark** Intents must be retracted in *reverse order*.

4. (a) **1 mark** Do not block waiting if the lock is already acquired.

   (b) **2 marks** *State* (spinlock) to facilitate lock semantics and *list* of blocked acquirers.

   (c) **1 mark** Any order guaranteeing eventual progress to all waiting threads. (Not FIFO)

   (d) **1 mark** They have no state.

   (e) **1 mark** The constructor allows the lock state to be initialized closed or open (0/1).

   (f) **6 marks** Can be done with one semaphore by reusing it.

```
1   Semaphore L1(0), L2(0);
1   COBEGIN
2       BEGIN S1; S3; P(L1); S4; V(l2); S5 END;
2       BEGIN S2; V(L1); P(L2); S6; END;
    COEND
```

```
1   Semaphore L1(0), L2(0);
1   COBEGIN
2       BEGIN S1; S3; V(L1); P(l2); S5 END;
2       BEGIN S2; P(L1); S4; V(L2); S6; END;
    COEND
```

5. **20 marks**

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

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

1     Y = ch;
1     suspend();
1     Z = ch;

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

1     W = ch;
1     for ( cnt = 1;; cnt += 1 ) {
1       if ( cnt == xcnt + 2 ) { _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. (a) **4 marks**

```
1   for ( int i = 0; i < cols; i += 1 ) {
2       if ( row[i] != ( i == r ? 1 : 0 ) ) return false;
    } // for
1   return true;
```

(b) **3 marks**

```
1   COFOR( r, 0, rows,                                  // thread per row
2       if ( ! identityCheck( r, M[r], cols ) ) identity = false;
    ); // COFOR
```

(c) **11 marks**

```
    struct WorkMsg : public uActor::Message {           // derived message
1       const int r, * row, cols;
        bool & identity;
        WorkMsg( const int r, const int row[], const int cols, bool & identity ) :
1           Message( uActor::Delete ), r( r ), row( row ), cols( cols ), identity( identity ) {}
    };
    _Actor Identity {
1       Allocation receive( Message & w ) {
1           Case( WorkMsg, w ) {                        // discriminate derived message
                WorkMsg & w = *w_d;                     // eye candy
2               if ( ! identityCheck( w.r, w.row, w.cols ) ) w.identity = false;
            };
1           return Delete;                              // one-shot
        }
    };
1   uActorStart();                                      // start actor system
1   for ( unsigned int r = 0; r < rows; r += 1 ) {
1       *new Identity | *new WorkMsg( r, M[r], cols, identity );
    }
1   uActorStop();                                       // wait for all actors to terminate
```

(d) **7 marks**

```
_Task IdentityCheck {
1   const int r, cols, *row;
    uBaseTask & prgMain;

    void main() {
1       try {
1           _Enable {
2               if ( ! identityCheck( r, row, cols ) ) _Resume NotIdentity() _At prgMain;
            }
1       } catch( Stop & ) {}
    }
  public:
    IdentityCheck( const int r, const int row[], const int cols, uBaseTask & prgMain ) :
1       r(r), row(row), cols(cols), prgMain( prgMain ) {}
};
```

(e) **19 marks**

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

    int main() {
1       int rows, cols;
-       cin >> rows >> cols;                                    // read matrix size

        int M[rows][cols], r, c;

1       for ( r = 0; r < rows; r += 1 ) {                       // read matrix
-           for ( c = 0; c < cols; c += 1 ) {
1               cin >> M[r][c];
1               cout << M[r][c] << ' ';
            } // for
-           cout << endl;
        } // for

        bool identity = true;

1       IdentityCheck *workers[rows];
1       for ( r = 0; r < rows; r += 1 ) {                       // create tasks to process rows
1           workers[r] = new IdentityCheck( r, M[r], cols, uThisTask() );
        } // for
1       try {
1           r = 0;                                              // initialize before Enable
1           _Enable {
1               for ( ; r < rows; r += 1 ) {
1                   delete workers[r];                          // wait for completion and delete tasks
                } // for
            } // _Enable
1       } _CatchResume( NotIdentity ) {
1           if ( identity ) {                                   // first identity-check failure ?
1               identity = false;
1               for ( int i = r + 1; i < rows; i += 1 ) {       // immediately stop any more checking
1                   _Resume IdentityCheck::Stop() _At *workers[i];
                }
            } // if
        } // try
1       cout << (identity ? "" : "not ") << "identity!" << endl;
    } // main
```