Midterm Answers – CS 343 Fall 2019

Instructor: Peter Buhr

November 3, 2022

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

1. (a) **4 marks** 1 duplicate code

```
1   for ( ;; ) {
1       cin >> d;
1     if ( cin.fail() ) break;
        ... // LOOP BODY
    }
```

   Cannot be done with **do**-**while**.

   (b) **2 marks** A flag variable is used solely to affect control flow, i.e., variable does not contain data associated with computation.

   (c) **1 mark** multi-level exit

   (d) **1 mark** To retain state from one inner lexical (static) scope to another.

   (e) **2 marks** One data structure cannot change its size after creation (fixed size) and the other can shrink/grow after creation (variable size). Fixed sized can appear on the stack.

   (f) **1 mark** The return code must be checked at each level in the unwinding.

   (g) **2 marks** Routine activation on the stack and transfer point within the routine.

   (h) **2 marks** Termination unwinds the stack during propagation and resumption does not unwind.

2. (a) **2 marks** Not terminate routine call on return and remember data/execution state for resuming.

   (b) **3 marks** 1 uThisCoroutine, 2 **this**, 3 last resumer

   (c) **2 marks** *cycle* is call graph, *definition-before-use* issues building the cycle

   (d) i. **2 marks** stackless: use the caller's stack and a fixed-sized local-state
   stackful: separate stack and a fixed-sized (class) local-state

   ii. **1 mark** stackful

   (e) **2 marks** A non-local exception is raised between entities with separate stacks. Coroutines must be explicitly restarted to receive the nonlocal exception but tasks have their own thread to receive the nonlocal exception.

3. (a) **2 marks** If interrupts affect scheduling (execution order), it is preemptive, otherwise the scheduling is non-preemptive.

   (b) **2 marks** Implicit concurrency indirectly accesses concurrency via specialized mechanisms (e.g., pragmas or parallel for) and *threads are implicitly managed*. Explicit concurrency directly accesses concurrency and *threads explicitly managed*.

   (c) **3 marks** Time to perform program work is user time.
   Time program runs is real time (wall-clock) .
   Real time

   (d) **1 mark** Critical path is the longest concurrent path bounding speedup.

   (e) **2 marks** COBEGIN/COEND can only create tree (lattice) thread graph, while START/WAIT can generate an arbitrary thread graph.

   (f) **1 mark** An actor does not have a thread.

   (g) **2 marks** Preventing simultaneous execution of a critical section by multiple threads.

   (h) **2 marks** Unbounded overtaking means other threads can use the critical section until the next thread scheduled to use it has indicated its intent.
   Bounded overtaking means other threads cannot use the critical section until the next thread scheduled to use it has proceeded.

   (i) **2 marks** A spin lock is implemented using busy waiting if the lock is in use.
   A blocking lock makes one check and blocks if the lock is in use.

   (j) **2 marks** Barging avoidance allows bargers but immediately blocks the barger inside the lock.
   Barging prevention does not allow bargers to enter the lock.

4. **20 marks**

```
      void main() {
1         try {
1             _Enable {
1                 for ( ;; ) {
1                     for ( ;; ) {                          // scan for letter
1                       if ( isalpha( ch ) ) break;
1                         cout << ch;
1                         suspend();
                      } // for
1                     ch = toupper( ch );                   // capitalize
1                     for ( ;; ) {
1                         if ( isspace( ch ) ) {            // space => could be sentence
1                             for ( ;; ) {                  // scan over space
1                                 cout << ch;
1                                 suspend();
1                               if ( ! isspace( ch ) ) break;
                              } // for
1                             ch = toupper( ch );           // capitalize ?
                          } // if
1                         if ( ! ispunc( ch ) ) {           // not punctuation ?
-                             for ( ;; ) {                  // scan over sentence
-                                 cout << ch;
-                                 suspend();
1                              if ( ispunc( ch ) ) break; // punctuation ?
                              } // for
                          } else {
1                             cout << ch;
1                             suspend();
                          } // if
                      } // for
                  } // _Enable
              } // _Enable
1         } catch( Eof ) {
          } // try
      } // Capitalize::main
```

-5 if not using coroutine state.

5. (a) **4 marks**

```
1   for ( int r = 0; r < cols; r += 1 ) {
1       if ( row1[r] != row2[r] ) {
1           _Resume NotEqual() _At prgMain;
1           return;
        } // exit
    } // for
```

(b) **6 marks**

```
1   try {
1       _Enable {
1           COFOR( r, 0, rows − 1,
1               equalCheck( M[r], M[r + 1], cols, prgMain );
            );
        } // _Enable
1   } catch( NotEqual ) {
1       notEqual = true;
    } // try
```

-2 for **try**-**catch** inside COFOR because wrong thread gets exception

(c) **7 marks**

```
struct WorkMsg : public uActor::Message {
1   const int * row1, * row2, cols;
1   uBaseTask & prgMain;
1   WorkMsg( const int row1[], const int row2[], int cols, uBaseTask & prgMain ) :
        Message( uActor::Delete ), row1( row1 ), row2( row2 ), cols( cols ), prgMain( prgMain ) {}
}; // WorkMsg

_Actor EqualRows {
    Allocation receive( Message & msg ) {
1       Case( WorkMsg, msg ) {                    // discriminate derived message
1           WorkMsg & w = *msg_d;                 // eye candy
1           equalCheck( w.row1, w.row2, w.cols, w.prgMain );
        } else assert( false );                   // bad message
1       return Finished;                          // one−shot
    } // EqualRows::receive
}; // EqualRows
```

(d) **7 marks**

```
_Task EqualRows {                                 // check rows
  public:
    _Event Stop {};                               // concurrent exception
  private:
1   const int * row1, * row2, row, cols;
1   uBaseTask & prgMain;

    void main() {
1       try {
1           _Enable {
1               equalCheck( row1, row2, cols, prgMain );
            } // _Enable
1       } catch( Stop ) {
        } // try
    } // EqualRows::main
  public:
1   EqualRows( const int row1[], const int row2[], int row, int cols, uBaseTask & prgMain ) :
            row1( row1 ), row2( row2 ), row( row ), cols( cols ), prgMain( prgMain ) {}
}; // EqualRows
```

(e) i. **9 marks**

```
1   try {
1       _Enable {
1           uActor::start();                                    // start actor system
1           EqualRows equalRows[rows - 1];
1           for ( unsigned int r = 0; r < rows - 1; r += 1 ) {
1               equalRows[r] | *new WorkMsg( M[r], M[r + 1], cols, prgMain );
            } // for
1           uActor::stop();                                     // wait for all actors to terminate
        } // _Enable
1   } catch( NotEqual ) {
1       notEqual = true;
    } // try
```

ii. **12 marks**

```
1   EqualRows * workers[rows - 1];

1   for ( r = 0; r < rows - 1; r += 1 ) {                       // create task to calculate rows
1       workers[r] = new EqualRows( M[r], M[r + 1], r, cols, prgMain );
    } // for

1   try {
1       for ( r = rows - 2 ; r >= 0; r -= 1 ) {                 // wait for completion and delete tasks
1           _Enable {                                           // poll for concurrent exception
1               delete workers[r];
            } // _Enable
        } // for
1   } _CatchResume( NotEqual ) {
1       if ( ! notEqual ) {                                     // if not diagonal symmetric
1           for ( int i = r - 1; i >= 0; i -= 1 ) {             // try to stop other tasks
1               _Resume EqualRows::Stop() _At *workers[i];
            } // for
1           notEqual = true;                                    // do not do this again
        } // if
    } // try
```

Solutions of the form:

```
for ( unsigned int i = 0; i < NumFiles; i += 1 ) {
    _Resume Search::Stop() _At *workers[i];
    delete workers[i];
} // for
```

had a -3 deduction for prohibiting concurrency by waiting for each worker to end before notifying the next to stop.