Midterm Answers – CS 343 Fall 2019

Instructor: Peter Buhr

November 2, 2023

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

1. (a) **5 marks**

```
1   bool flag = false;
1   while ( ! flag ) {
        S1;
1       if ( C ) flag = true;
1       else S2;
    }
1   if ( flag ) E;
```

   (b) **1 mark** Code in the **else** clause is logically part of the loop body not part of the loop exit.

   (c) **1 mark** Because flag variables can be set/reset/tested at arbitrary locations in a program.

   (d) **1 mark** New nested control structures can be added without having to change existing code.

   (e) **2 marks**

      i. Cannot create a loop.

      ii. Cannot branch into a control structure.

   (f) **1 mark** To perform static multi-level exit when labelled break and continue are unavailable.

   (g) **3 marks** uNoCtor allocates a dynamically-sized array on the stack without running the default constructor. Subsequently, any constructor with arguments can be run on the each array element. unique_ptr allocates the array in the heap.

   (h) **2 marks** The $O(N)$ search for termination eliminates testing return codes in unwound routines. The $O(N)$ search for resumption eliminates passing fixup routines.

   (i) **2 marks** Destructors are called during exception propagation, and C++ does not allow another exception to be raised during propagation.

   A destruction can conditionally raise an exception only if it is not invoked during propagation (uncaught_exceptions).

   (j) **1 mark** The exit from the resumption handle changes the dynamic return to a static return.

   (k) **1 mark** A coroutine/task needs to execute to a safe point (e.g., **try**) so it can allow propagation of non-local exceptions.

2. (a) i. **1 mark** coroutine's starter

      ii. **2 marks** provides guaranteed path for control to return to the program main

   (b) **1 mark** To proceed into an initial state before the first member is called.

   (c) **2 marks** last resumer, joiner

3. (a) **2 marks** A *concurrent bottleneck* is a hardware or software restriction that slows down or stops concurrent speedup.

   (b) **1 mark** The *scheduler* manages the ready queue and placement of tasks/processes on CPUs.

   (c) **2 marks** Speedup starts linear as more CPUs are added but then hits a bottleneck and begins to level out and/or drop.

   (d) **2 marks** implicit, explicit, implicit, explicit

   (e) **4 marks** Two threads arrive simultaneously. Both declare their intent to enter at lines 1. Then both wait at line 2 for the other thread to retract its intent, which cannot happen (empty loop bodies). livelock or indefinite postponement

   (f) **2 marks** RW-safe means an algorithm works even if simultaneous writes to the same location scramble the bits or simultaneous read during a write to the same location sees the bits flicker.

   (g) **1 mark** A lock is needed for each *independent* critical section.

   (h) **1 mark** The ability to *read* and *write* atomically.

   (i) **1 mark** *Barging* occurs when waiting threads are scheduled (overtaken) by arriving threads.

   (j) **2 marks** Task tells scheduler to release current timeslice and do not be put on the ready queue, and atomically release the lock.

4. **16 marks**

```
        _Coroutine Bead {
            unsigned int id;
            Bead * part;

            void main() {
1               bool zombied = false;

1               suspend();
1               try {
1                   _Enable {
1                       for ( ;; ) {
1                           clasp.lock( prng() );
1                           part->next();
                        } // for
                    } // _Enable
1               } _CatchResume( Zombie & ) {
1                   zombied = true;
1               } catch( Won & ) {
1                   cout << (zombied ? "Zombie cannot win " : "Won") << id << endl;
                } // try
            } // main
          public:
            Bead( unsigned int id, Bead * partner ) : id{ id }, part{ partner } { resume(); };
            void next() { resume(); }
        }; // Beads

        int main() {
            enum { N = 5 };
1           uNoCtor<Bead> beads[N];

1           for ( unsigned int b = 0; b < N - 1; b += 1 ) {
1               beads[b].ctor( b, &beads[b + 1] );
            } // for
1           beads[N - 1].ctor( N - 1, &beads[0] );

// 1        for ( int b = 0; b < N; b += 1 ) {
// 2            beads[b].ctor( b, &beads[(b + 1) % N] );
//           }

1           beads[0]->next();
        } // main
```

-5 if not using coroutine state.

5. (a) **1 mark** The Halfway and manager threads must communicate during their lifetime or manager must stop workers part way to completion.

(b) **24 marks**

```
      _Actor Manager {
          enum { N = 10 };
1         uNoCtor<Halfway> halfway[N];
1         unsigned int completers = 0;
1         bool completed[N] = {};                                  // set to false

1         Allocation receive( Message & msg ) {
1             Case ( Progress, msg ) {
1                 if ( prng( 5 ) == 0 ) msg_d->complWork -= 50;

                  // Told to stop but work still has more work to do,
                  // i.e, worker still processing start messages.
1                 if ( ! completed[msg_d->id] && msg_d->complWork < 1000 ) {
1                     *halfway[msg_d->id] | uActor::startMsg;
                  } else {
                      // If not told to stop already, i.e., receive spurious progress messages,
                      // then reply with complete message.
1                     if ( ! completed[msg_d->id] ) {
-                         *halfway[msg_d->id] | complete;
1                         completers += 1;
1                         completed[msg_d->id] = true;
                      } // if
1                     if ( completers >= N / 2 ) {
1                         for ( unsigned int w = 0; w < N; w += 1 ) {
1                             if ( ! completed[w] ) {
1                                 completed[w] = true;
1                                 *halfway[w] | uActor::stopMsg;
                              } // if
                          } // for
                      } // if
                      // Cannot finish until all actors have been told to complete or stop.
1                     if ( completers == N ) return Finished;
                  } // if
              } // case
1             return Nodelete;
          } // Manager::receive
        public:
          Manager() {
1             for ( unsigned int w = 0; w < N; w += 1 ) {
1                 halfway[w].ctor( w, *this );
1                 *halfway[w] | uActor::startMsg;
              } // for
          } // Manager::Manager
      }; // Manager

      int main() {
1         uActor::start();
1         Manager manager;
1         uActor::stop();
      } // main
```

(c) **25 marks**

```
      _Task Halfway {
          void main() {
1             try {
1                 _Enable {
1                     for ( unsigned int complWork = 100;; complWork += 100 ) {
1                         _Resume Progress( id, complWork, spin ) _At pgmMain;
1                         while ( spin ) yield();                    // busy wait
1                         spin = true;
                      }
                  }
1             } catch( Complete & ) {
1             } catch( Stop & ) {
              } // try
          } // main
        public:
          Halfway( unsigned int id, uBaseTask & pgmMain ) : id{ id }, pgmMain{ pgmMain } {};
      }; // Halfway

      int main() {
          enum { N = 10 };
1         uNoCtor<Halfway> workers[N];
          {
1             for ( unsigned int w = 0; w < N; w += 1 )              // create tasks
-                 workers[w].ctor( w, uThisTask() );

1             unsigned int completers = 0;
1             bool completed[N] = {};                                // set to false
1             try {
1                 _Enable {
1                     for ( ; completers < N / 2; ) {
1                         uThisTask().yield();                       // busy wait
                      }
                  }
1             } _CatchResume( Progress & p ) {
1                 if ( prng( 5 ) == 0 ) p.complWork -= 50;
1                 if ( p.complWork >= 1000 ) {
1                     completers += 1;
1                     completed[p.id] = true;
1                     _Resume Complete() _At (uBaseCoroutine &)workers[p.id];
                  }
1                 p.spin = false;                                    // restart worker
              }
1             for ( unsigned int w = 0; w < N; w += 1 )
1                 if ( ! completed[w] ) _Resume Stop() _At (uBaseCoroutine &)workers[w];
          }
      } // main
```