Midterm Answers – CS 343 Winter 2023

Instructor: Caroline Kierstead

March 8, 2023

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

1. (a) **5 marks**

Mechanism	**Exit type**		
	static multi exit	*static multi-level exit*	*dynamic multi-level exit*
unlabelled **break**	✓	×	×
labelled **break**	✓	✓	×
goto	✓	✓	×
longjmp	✓	✓	✓
exception	✓	✓	✓

1 mark per line; -0.5 if only 1 error, -1 if 2 or more errors

(b) **2 marks** The problem is that labels have routine scope. Thus label B1 isn't known within function rtn. [2 marks = good answer, 1 mark = on right-track, 0 marks = incorrect answer]

(c) i. **1 mark** A *return union* combines the result with the return-code, requiring a return-code check upon accessing the result.

ii. **1 mark** Advantages (one of):

- lets you separate the error conditions from the "normal" return results
- lets you return multiple types (variadic template, similar effect to a tagged union)

iii. **1 mark** Disadvantages (one of):

- still a passive check i.e. programmer needs to explicitly check so can be omitted
- pay a performance penalty for a more complex data structure and more checks (more than exceptions, global return codes or a tagged union)

2. (a) **2 marks**

Object	*Line number*	*Cocaller*
b	12	::main / program main
a	29	b

(b) **4 marks**

Line number	*Becomes inactive*	*Becomes active*
5	b	::main
6	b	b
18	::main	b
29	b	a

(c) **2 marks** Yes. [1 mark] The order is: a, b, ::main [1 mark]

(d) i. **1 mark** One of:

```
_Resume Stop{} _At *partner; // star is optional
_Resume Stop() _At resumer(); // alternative approach
```

ii. **2 marks** Coroutine a must be activated to receive the exception [1 mark] by calling partner->next() or suspend() [1 mark].

iii. **2 marks** Coroutine a needs a guarded block (**try** block) containing **_Enable** [1 mark] and a handler for Stop [1 mark] (could be more specific and say either a resumption handler, or a termination handler i.e. either is fine).

iv. **3 marks** When no matching resumption handlers are found, the exception is thrown/raised as a termination exception. [1 mark] When no matching termination handlers are found, the exception UnhandledException (more fully, it's a uBaseCoroutine::UnhandledException though I don't expect them to remember that) is raised. [1 mark] The exception is propagated to the terminating coroutine's most recent resumer. [1 mark]

(e) i. **1 mark** The yield statement can only be used in the coroutine's main method (i.e. cannot call other routines and suspend/yield).

ii. **1 mark** full coroutining

3. (a) i. **1 mark** *Concurrent execution* can occur when there is only 1 CPU/processor.

ii. **2 marks** Since concurrent execution simulates parallelism (by rapidly context-switching between threads), only 1 CPU/processor is necessary. [1 mark]
Parallel execution requires multiple CPUs/processors (there is no such thing as a parallel program since parallelism comes from the hardware) i.e. requires execution on multiple CPUs/processors. [1 mark]

(b) **2 marks** *Unbounded overtaking* allows entry into the critical section until the other task that retracted its intent redeclares it.
Bounded overtaking disallows entry into the critical section because the other task never retracted its intent.

(c) i. **1 mark** Yes.

ii. **2 marks** The algorithm uses static priorities based upon task's index position in the ticket array to break ticket value ties. [2 marks = good answer, 1 mark = on right-track, 0 marks = incorrect answer]

4. (a) **2 marks** In the *busy-waiting* approach, the producers/consumers would block for mutual exclusion, and depending upon the state of the buffer, the barging task and the signalled task may not have to block. If there is no busy-waiting, in addition to blocking for mutual exclusion, barging tasks have to block, and incoming tasks whose turn it is may have to block depending upon the state of the buffer. Therefore, there is more blocking/context-switching in the second approach, which slows down performance. [2 marks = good answer, 1 mark = on right-track, 0 marks = incorrect answer]

(b) i. **1 mark** It uses *barging avoidance*.

ii. **2 marks** *Barging prevention* would require the release code to not release the lock upon determining that there is a waiting task. This implementation releases the lock, so it's *barging avoidance*. [2 marks = good answer, 1 mark = on right-track, 0 marks = incorrect answer]

5. **18 marks**

```
    void HexDump::main() {
        static const char Blank = ' ';
1       int i = 0, j = 0, k = 0;
1       try {
1           _Enable {
1               for ( ;; ) {
1                   for ( i = 0;; i += 1 ) {            // 4 main groups of characters
1                       for ( j = 0;; j += 1 ) {        // 2 sub-groups within each main group
1                           for ( k = 0; k < PAIRS; k += 1 ) { // 2 characters within each sub-sub-group
1                               cout << s[ch >> 4] << s[ch & 0x0f];
1                               suspend();
                            } // for
1                       if ( j == SUBGROUP - 1 ) break;
1                           cout << Blank;
                        } // for PAIR
1                   if ( i == GROUP - 1 ) break;
1                       cout << Blank << Blank << Blank;
                    } // for SUBGROUP
1                   cout << endl;
                } // for GROUP
            } // infinite loop
2       } catch( Eof & e ) {
1           if ( i < GROUP-1 || k < PAIRS || j < SUBGROUP-1 )
1               cout << endl;
        } // catch
    } // HexDump::main
```

-5 if not using coroutine state.
if statement in handler not needed due to placement of suspend but may be needed if code structured differently
Placement of handler should ensure coroutine main terminates so cannot be resumed (-1 if not done).

6. (a) **2 marks**

```
2   _Resume Found{} _At pgmMain;
```

(b) **2 marks**

```
1   COFOR( i, 0, NumFiles,
1       search( fileNames[i], key, pgmMain );
    );
```

(c) **6 marks**

```
    struct WorkMsg : public uActor::Message {
1       const string & key, & filename;
1       uBaseTask & pgmMain;

1       WorkMsg( const string & key, const string & filename, uBaseTask & pgmMain )
            : Message{uActor::Delete}, key{key}, filename{filename}, pgmMain{pgmMain} {}
    };

    _Actor Search {
        Allocation receive( Message & msg ) {
1           Case( WorkMsg, msg ) {
–               WorkMsg & w = *msg_d; // not required but nice; else use msg_d->...
1               search( w.filename, w.key, w.pgmMain );
            } else assert( false ); // bad message
1           return Finished; // one-shot
        }
    };
```

(d) **7 marks**

```
    _Task Search {
      public:
        _Event Stop {};                         // concurrent exception
      private:
1       const string & key, & filename;
1       uBaseTask & pgmMain;

        void main() {
1           try {
1               _Enable {                       // allow delivery of Stop
1                   search( filename, key, pgmMain );
                } // _Enable
1           } catch( Search::Stop & ) {}
        } // Search::main
      public:
1       Search( const string & key, const string & filename, uBaseTask & pgmMain ) :
            key{key}, filename{filename}, pgmMain{pgmMain} {}
    }; // Search
```

(e) i. **9 marks**

```
1   try {
1       _Enable {
1           uActor::start();          // start actor system
1           Search searchers[numFiles];
1           for ( unsigned int i = 0; i < numFiles; i += 1 ) {
1               searchers[i] | *new WorkMsg( i, key, fileNames[i], pgmMain );
            } // for
1           uActor::stop();
        }
1   } catch( Found & ) {
1       found = true;
    }
```

ii. **12 marks**

```
1   Search * workers[numFiles];

1   for ( unsigned int i = 0; i < numFiles; i += 1 ) {
1       workers[i] = new Search( i, key, fileNames[i], pgmMain );
    } // for
    unsigned int s = 0;                          // initialize before Enable

1   try {
1       for ( ; s < numFiles; s += 1 ) {         // wait for completion and delete tasks
1           _Enable {
1               delete workers[s];
            } // _Enable
        } // for
1   } _CatchResume( Found & ) {
1       if ( ! found ) {
1           found = true;
1           for ( unsigned int i = s + 1; i < numFiles; i += 1 ) {
1               _Resume Search::Stop() _At *workers[i];
            } // for
        } // 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
```

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