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

1. (a) i. 1 mark recomputing the reason for loop termination
   ii. 2 marks

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

    1 if ( j >= 10 ) { E1; break; }

}
```

(b) A nonlocal transfer goes to a:
   i. 1 mark routine activation on the stack and
   ii. 1 mark transfer point within the routine.

(c) i. 1 mark To access the normal outcome explicitly requires a getter operation, which verifies
    if it is available in the union.
   ii. 1 mark The `return union` separates the normal and alternate outcomes into separate fields
    in the union.

(d) 8 marks
   i. 2 unguarded, 4 guarded
   ii. 2 throws
   iii. 7 examined catch clauses
   iv. C8
   v. B6 resumption, B2 retry, B1 termination

(e) 6 marks

```c
1 void f( void (*E1)(), void (*E2)() ) {
    ... E1(); ... E2(); ...
}
1 void g( void (*E1)(), void (*E2)() ) {
    ... f( E1, E2 ); ...
}
1 void E1() {}
- void E2() {}
int main() {
    ... g( E1, E2 ); ...
}
```
2. (a) i. **2 marks** The cocaller creates the coroutine’s stack and context switches from its stack to the coroutine’s stack.
   
   ii. **2 marks** In full-coroutines, the last resumer may have terminated so the cocaller/starter is used to get back to the program main.

(b) **2 marks** When a coroutine suspends back to its last resume.
When a coroutine does not handle an exception and raises UnhandledException at its last resume.

(c) i. **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.

   ii. **2 marks** _Enable_ exception handling inside a try block to receive non-local exceptions, and have a _CatchResume or catch_ handler for E.

(d) i. **1 mark** The break must unwind the stack.

   ii. **1 mark** Throw an exception and catch it in the same try statement.

3. (a) **2 marks** Concurrent execution is any situation in which execution of multiple processes or tasks appears to be performed in parallel but in fact are being executed by only one processor. Parallel execution is when 2 or more operations occur simultaneously. This can only occur when multiple processors (CPUs) are present.

(b) i. **4 marks**

   ii. **1 mark** Without the compound statement, the newline is printed before rather then after the value.

(c) **2 marks** COBEGIN/COEND are very succinct for simple concurrency problems.
COBEGIN/COEND are restricted to producing lattice thread/process graphs, which restricts the problems that can be solved. OR it is difficult to communicate results

(d) i. **2 marks** Rule 1 (Mutual exclusion) : both threads check the lock and find it open, both threads close the lock, and both enter the critical section.

   ii. **2 marks** Rule 5 (Starvation) : one thread opens the lock, and check the lock is open and reenters the critical section, in between the other thread checking the lock status.

(e) **2 marks** If the lock is open, the thread acquires the lock. If the lock is closed, the thread does not block waiting.

(f) i. **1 mark** atomic read-write action

   ii. **1 mark** One of: fetch/assign, test/set, fetch/increment, compare/swap

   iii. **1 mark** One of: faster to react to change in a multiprocessor system since does not have to context switch, cheaper than a blocking lock

   iv. **1 mark** One of: allows starvation, allows barging, in a uniprocessor system wastes CPU cycles by busy-waiting since state does not change
4. 15 marks

```c
void main() {
    int counter = 0;

    if ( ch != 'a' ) {
        _Resume Error() _At resumer(); return; }

    for ( ;; ) {
        suspend();
        if ( ch != 'b' ) break;
        counter += 1;
    } // for

    if ( counter < 1 ) {
        _Resume Error() _At resumer(); return; }

    for ( ;; ) {
        if ( ch != 'c' ) break;
        counter -= 1;
        if ( counter < 1 ) break;
        suspend();
    } // for

    if ( counter != 1 || ch != 'd' ) {
        _Resume Error() _At resumer(); return; }

    _Resume Match() _At resumer();
} // main
```

Maximum 8 if not using coroutine state.
5. (a) 5 marks
1 ifstream input( filename.c_str() );
   string line;
1 for ( ;; ) {
1 getline( input, line );
1 if ( input.fail() ) return false;
1 if ( line.find( key ) != string::npos ) return true;
} // for

(b) 5 marks
1 #if defined( CFOR )
1 #include <uCobegin.h>
1 #elif defined( ACTOR )
1 #include <uActor.h>
1 #endif
1 #include <string>
1 #include <iostream>
1 #include <fstream>
1 using namespace std;
1 string key;
1 unsigned int NumFiles;
1 cin >> key >> NumFiles;
1 cout << "key " << (searchFiles( key, NumFiles ) ? "found" : "not found") << endl;

(c) 6 marks
1 bool found = false; // should appear in each solution but only grade once
1 COFOR( i, 0, NumFiles,
1 string filename;
1 isacquire( cin ) >> filename;
1 if ( search( filename, key ) ) found = true;
1);
1 return found; // should appear in each solution but only grade once

Marks for first and last line given once for all solutions.

(d) 12 marks
1 struct WorkMsg : public uActor::Message {
1 1 const string & key;
1 1 bool & found;
1 1 WorkMsg( const string & key, bool & found ) :
1 1 Message(uActor::Delete), key(key), found(found) {};
1 1 _Actor Search {
1 1 1 Case( WorkMsg, msg ) {
1 1 1 string filename;
1 1 1 isacquire( cin ) >> filename;
1 1 1 if ( search( filename, msg_d->key ) ) msg_d->found = true;
1 1 1 return Finished; // since on stack
1 1 1 };
1 1 uActorStart(); // start actor system
1 1 Search searchers[NumFiles];
1 1 for ( unsigned int i = 0; i < NumFiles; i += 1 ) {
1 1 searchers[i] | *new WorkMsg( key, found );
1 1 } // for
1 1 uActorStop();
(e) 20 marks

```cpp
void main() {
    string filename;
    isacquire( cin ) >> filename;
    try {
        _Enable { // allow delivery of Stop
            if ( search( filename, key ) ) _Resume Found() _At pgmMain;
        } // _Enable
    } catch( Stop & ) {} // Search::main
}
```

```cpp
public:
    _Event Stop {}; // concurrent exception
    Search( const string & key, uBaseTask & pgmMain ) :
        key(key), pgmMain{pgmMain} {}
}; // Search
```

```cpp
Search * workers[NumFiles];
for ( unsigned int i = 0; i < NumFiles; i += 1 ) {
    workers[i] = new Search( key, uThisTask() );
} // for
unsigned int s = 0; // initialize before Enable
try {
    _Enable { // wait for completion and delete tasks
        for ( ; s < NumFiles; s += 1 ) {
            delete workers[s];
        } // for
    } // _Enable
    _CatchResume( Found & ) {
        if ( ! found ) {
            for ( unsigned int i = s + 1; i < NumFiles; i += 1 ) {
                _Resume Search::Stop() _At *workers[i];
            } // for
            found = true;
        } // if
    } // try
}
```

Solutions of the form:
```cpp
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.