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

1. (a) 4 marks

```c
if ( i != 1 ) goto Default; // case 1 body
goto EndSwitch; // default body
Default: ;
EndSwitch: ;
```

(b) 2 marks Modularity is refactoring contiguous code into a routine. The problem is refactoring a label because it only has routine scope.

c) 1 mark False.

d) 1 mark One.

e) 2 marks The activation block is necessary because the label within an activation block is not unique due to recursion.

f) 2 marks setjmp is called explicitly to set the label and called implicitly when longjmp transfers to it.

(g) 1 mark Represent the nonlocal transfer using controlled exception handling or use try/catch.

(h) 2 marks If _Resume searches the stack and does not find a resumption handler, then the defaultResume throws the exception so the stack is searched again for a termination handler.

(i) 1 mark The stack unwinding ensures all objects on the coroutine’s stack with destructors have their destructor executed.

(j) 1 mark C++ does not support initialization of array elements with external values.

(k) 2 marks If the other thread deletes the storage, the original thread heap is now shared and must be protected with a lock.

2. (a) 2 marks An output coroutine returns (generates) values on each call to an interface member. An input coroutine is passed a value on each call to an interface member and processes it.

(b) 1 mark stack

(c) 2 marks first resume, coroutine-main returns

d) 1 mark To force the coroutine into an initial state before the first resume from an interface member.

(e) 2 marks The _Enable terminates, turning off nonlocal exceptions, so no new nonlocal exceptions can arrive during handling of the raised one.

(f) 2 marks The coroutine context switches to the coroutine’s starter coroutine, because following the starters gets back to the program main.

(g) 1 mark cycle

(h) 1 mark It context switches to itself.
3.  (a) **1 mark** M user threads are scheduling across N kernel threads.
    (b) **1 mark** some bottleneck is reached
    (c) **2 marks** implicit, explicit
    (d) **1 mark** barging
    (e) **2 marks** Rule 5 (starvation) is broken. The task in the inner loop retracts its intent and it may never see the other task retract its intent.
    (f) **3 marks** INT_MAX ⇒ do not want in, 0 ⇒ selecting ticket.
        Entering tasks with a ticket must waiting for a selecting task without a ticket because the selector may have a higher priority ticket value.
    (g) **1 mark** The arbiter serves tasks in cyclic order.

4.  (a) **2 marks** There is an unbounded number of yielding, which can result in starvation.
    (b) **2 marks** It is possible to be lazy with spin locks because if the event is missed there will be another check. With blocking locks there is only one check, so the wake up is more complex to ensure not to miss the check.
    (c) **1 mark** Have the scheduler release the lock or turn off preemption
    (d) **2 marks** Lock the stream so it is a critical section. Create an anonymous lock-object (RAII) that persists for the duration of the expression.
    (e) **2 marks** Synchronization lock.
        Coordinate a group of tasks performing a concurrent operation surrounded by sequential operations or ensure a group of tasks start and end at the same time.
    (f) **6 marks**

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

        1 Semaphore L1(0), L2(0);
        1 COBEGIN
        2 BEGIN S1; V(L1); S4; P(L2); S5; END;
        2 BEGIN S2; P(L1); S3; V(L2); END;
        COEND
5. 17 marks

```c
void getNdigits( int N ) {
  for ( int i = 0; i < N; i += 1 ) {
    if ( ! isdigit( ch ) ) {
      _Resume Error() _At resumer(); return; } // must have a digit
    suspend();
  } // for
} // Phone::getNdigits

void main() {
  try {
    if ( ch == '(' ) {
      suspend(); // get '(' digit
      getNdigits( 3 );
      if ( ch != ')' ) {
        _Resume Error() _At resumer(); return; } // must have a ')'
    suspend(); // get digit
  } // if

  getNdigits( 3 );
  if ( ch != '-' ) {
    _Resume Error() _At resumer(); return; } // must have a '-'
  suspend(); // get digit
  getNdigits( 4 );
  if ( ch != '\n' ) {
    _Resume Error() _At resumer(); return; }
  _Resume Match() _At resumer();
} catch ( int ) {}
} // Phone::main
```

Maximum 8 if not using coroutine state.
6. (a) 4 marks
   1 min = max = row[0]; // min = INT_MAX; max = INT_MIN;
   1 for ( unsigned int r = 1; r < cols; r += 1 ) { // find min, max of row
       1 if( row[r] < min ) min = row[r];
       1 if( row[r] > max ) max = row[r];
   } // for

(b) 2 marks
   1 COFOR( r, 0, rows, // thread per row
   1   minmax( M[r], cols, rmin[r], rmax[r] );
   1 ); // COFOR

(c) 28 marks
   1 #include <iostream>
      using namespace std;
   1 _Task MinMax {
     1 const int * row, cols;
     1 int & min, & max;
     1 uBaseTask & prgMain;
     1 void main() {
       1 try {
         1 _Enable {
           1 minmax( row, cols, min, max );
           1 if ( min == max ) _Resume Equal() _At prgMain;
         } // _Enable
       1 } catch( Stop & ) {} } // main public:
      MinMax( const int row[], const int cols, int & min, int & max, uBaseTask & prgMain ) :
      1 row( row ), cols( cols ), min( min ), max( max ), prgMain( prgMain ) {};
   1 int main() {
     1 int rows, cols;
     1 cin >> rows >> cols;
     1 int M[rows][cols], rmin[rows], rmax[rows], r, c;
     1 for ( r = 0; r < rows; r += 1 ) { // read/print matrix
       1 for ( c = 0; c < cols; c += 1 ) {
         1 cin >> M[r][c];
         1 cout << M[r][c] << " ",
       } // for
       1 cout << endl;
     } // for
     1 MinMax *workers[rows];
     1 for ( r = 0; r < rows; r += 1 ) { // create task to calculate rows
       1 workers[r] = new MinMax( M[r], cols, rmin[r], rmax[r], uThisTask() );
     } // for
     1 bool equal = false;
     1 int min = INT_MAX, max = INT_MIN;
     1 try {
       1 r = 0; // initialize before Enable
       1 _Enable {
         1 for ( ; r < rows; r += 1 ) { // wait for completion and delete tasks
           1 delete workers[r];
           1 if( rmin[r] < min ) min = rmin[r];
           1 if( rmax[r] > max ) max = rmax[r];
         } // for
       } // _Enable
     1) _CatchResume( Equal ) { // try
       1 if ( ! equal ) {
         1 for ( int i = r + 1; i < rows; i += 1 ) {
           1 _Resume MinMax::Stop() _At *workers[i];
         } // for
         1 equal = true;
       } // if
     1) // try
     1 if ( equal ) cout << "equal min/max" << endl;
     1 else cout << "min:" << min << " max:" << max << endl;
   } // main