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

1. (a) **2 marks**
   ```
   for ( i = 0; i += 1 ) { // linear search for key in list
     if ( i == size ) { C1; break; }
     if ( key == list[i] ) { C2; break; }
   }
   ```

(b) **1 mark** Retain state from one inner lexical (static) scope to another.

(c) **4 marks**
   - static call
   - dynamic call
   - static return
   - dynamic return

(d) **2 marks** The `throw` raises a B, which is not caught by a D.
   `µC++` raises a D.

(e) **2 marks** When the raise site cannot continue, termination searches for a catch/handler that can recover and continue lower on the stack.
   When the raise site can continue, resumption searches for a catch/handler that can fix up and continue after the raise.

(f) **2 marks** `vector` can use dynamic allocation and the heap is a point of lock contention because it is shared/serial resource among all threads.

2. (a) **2 marks** A coroutine allows a routine to suspend its execution rather than terminating (returning) to its caller.
   The caller can then resume the suspended routine rather than call it again from the top.

(b) **2 marks** The stack does not grow.
   Set the stack to its maximum depth when the coroutine is created.

(c) **2 marks** The first resume context switches (cocalls) to start the coroutine.
   A terminated coroutine context switches to its starter coroutine.

(d) **3 marks**
   - program main creates ping and pong
   - program main starts ping; ping starts pong
   - ping and pong are in a cycle

(e) **2 marks** Cannot modularize/call-routines because generator/iterator is stackless coroutine.

(f) **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`.
3. (a) **2 marks** The other thread is simultaneously reading and sees the bits flicker in \( i \) or writing \( i \) and the bits become scrambled.

(b) **1 mark** User threading has better performance because context switching does not cross the application/kernel (OS) boundary.

(c) **3 marks** Amount of concurrency, critical path among concurrency, scheduler efficiency

(d) **1 mark** Yes

(e) i. **1 mark** A thread may not enter the critical section successive times when the other thread does not want in.

ii. **2 marks** Trick question converting alternation into a spinlock.

\[
\begin{align*}
1 & \text{while} ( \text{TestSet}(::\text{Last}) \equiv 0); & \text{// entry protocol} \\
& \text{CriticalSection();} & \text{// critical section} \\
1 & ::\text{Last} = 1; & \text{// exit protocol}
\end{align*}
\]

(f) **1 mark** Intents must be retracted in reverse order.

4. (a) **1 mark** Do not block waiting if the lock is already acquired.

(b) **2 marks** State (spinlock) to facilitate lock semantics and list of blocked acquirers.

(c) **1 mark** Any order guaranteeing eventual progress to all waiting threads. (Not FIFO)

(d) **1 mark** They have no state.

(e) **1 mark** The constructor allows the lock state to be initialized closed or open (0/1).

(f) **6 marks** Can be done with one semaphore by reusing it.

\[
\begin{align*}
1 & \text{Semaphore } L1(0), L2(0); & 1 & \text{Semaphore } L1(0), L2(0); \\
1 & \text{COBEGIN} & 1 & \text{COBEGIN} \\
2 & \text{BEGIN } S1; S3; P(L1); S4; V(L2); S5 \text{ END;} & 2 & \text{BEGIN } S1; S3; V(L1); P(L2); S5 \text{ END;} \\
2 & \text{BEGIN } S2; V(L1); P(L2); S6; \text{ END;} & 2 & \text{BEGIN } S2; P(L1); S4; V(L2); S6; \text{ END;} \\
& \text{COEND} & & \text{COEND}
\end{align*}
\]
void main() {
  char X, Y, Z, W;
  int xcnt, cnt;

  X = ch;
  for ( xcnt = 1;; xcnt += 1 ) {
    suspend();
    if ( ch != X ) break;
  } // for

  Y = ch;
  suspend();
  Z = ch;

  for ( cnt = 1;; cnt += 1 ) {
    if ( cnt > xcnt + 1 ) { __Resume Error() __At resumer(); return; }
    suspend();
    if ( ch != Y ) break;
    suspend();
    if ( ch != Z ) { __Resume Error() __At resumer(); return; }
  } // for

  W = ch;
  for ( cnt = 1;; cnt += 1 ) {
    if ( cnt == xcnt + 2 ) { __Resume Match() __At resumer(); return; }
    suspend();
    if ( ch != W ) { __Resume Error() __At resumer(); return; }
  } // for
} // Grammar::main

Maximum 10 if not using coroutine state.
6. (a) 4 marks
1  for ( int i = 0; i < cols; i += 1 ) {
2       if ( row[i] != ( i == r ? 1 : 0 ) ) return false;
} // for
1  return true;

(b) 3 marks
1  COFOR( r, 0, rows, // thread per row
2       if ( ! identityCheck( r, M[r], cols ) ) identity = false;
2       ); // COFOR

(c) 11 marks
1  struct WorkMsg : public uActor::Message {
1     const int r, * row, cols;
1     bool & identity;
1     WorkMsg( const int r, const int row[], const int cols, bool & identity ) :
1         Message( uActor::Delete ), r( r ), row( row ), cols( cols ), identity( identity ) {} };
1  _Actor Identity {
1     Allocation receive( Message & w ) {
1       Case( WorkMsg, w ) {
1         WorkMsg & w = *w_d;
1         if ( ! identityCheck( w.r, w.row, w.cols ) ) w.identity = false;
1       }; // discriminate derived message
1       return Delete; // one-shot
1     };
1  }
1  uActorStart(); // start actor system
1  for ( unsigned int r = 0; r < rows; r += 1 ) {
1     *new Identity | *new WorkMsg( r, M[r], cols, identity );
1  }
1  uActorStop(); // wait for all actors to terminate
(d) 7 marks

    _Task IdentityCheck {
        const int r, cols, *row;
        uBaseTask & prgMain;

        void main() {
            try {
                _Enable {
                    if ( ! identityCheck( r, row, cols ) ) _Resume NotIdentity() _At prgMain;
                }
                } catch( Stop & ) {} }
            }
        }
    }

    public:
        IdentityCheck( const int r, const int row[], const int cols, uBaseTask & prgMain ) :
            r(r), row(row), cols(cols), prgMain( prgMain ) {}
    }

(e) 19 marks

    #include <iostream>
    using namespace std;

    int main() {
        int rows, cols;
        cin >> rows >> cols;  // read matrix size
        int M[rows][cols], r, c;

        for ( r = 0; r < rows; r += 1 ) {  // read matrix
            for ( c = 0; c < cols; c += 1 ) {
                cin >> M[r][c];
                cout << M[r][c] << ' ';  // for
            }
            cout << endl;  // for
        }

        bool identity = true;

        IdentityCheck *workers[rows];
        for ( r = 0; r < rows; r += 1 ) {  // create tasks to process rows
            workers[r] = new IdentityCheck( r, M[r], cols, uThisTask() );
        }  // for
        try {
            r = 0;  // initialize before Enable
            _Enable {
                for ( ; r < rows; r += 1 ) {
                    delete workers[r];  // wait for completion and delete tasks
                }  // for
            }  // _Enable
            _CatchResume( NotIdentity ) {
                if ( identity ) {  // first identity-check failure?
                    identity = false;
                    for ( int i = r + 1; i < rows; i += 1 ) {  // immediately stop any more checking
                        _Resume IdentityCheck::Stop() _At *workers[i];
                    }  // for
                }  // if
            }  // _CatchResume
            cout << (identity ? "" : "not ") << "identity!" << endl;
        }  // try
    }  // main