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

Part A - Multiple Choice

1. 1 mark D
2. 1 mark C
3. 1 mark B
4. 1 mark A
5. 1 mark E
6. 1 mark C
7. 1 mark A
8. 1 mark A
9. 1 mark D
10. 1 mark D
11. 1 mark E

Part B - Short Answer and Code

1. (a) 2 marks Any two of:
   i. labelled break
   ii. labelled continue
   iii. sequel
   iv. goto

(b) 2 marks They rely upon labels, which only have routine scope. (Thus cannot be used to transfer control between routines.)

(c) Needs to talk about the control flow, not the mechanism. Can’t just re-state the meaning/definition of non-local transfer.
   i. 1 mark They both remove activation/block frames from the run time stack.
   ii. 1 mark In C++, the run time stack is “unwound” i.e. destructors for local objects are run. This will not happen in a C++ program if the C longjmp is used.

(d) i. 4 marks
   • 1) sequel
   • 2) routine call
   • 3) termination
   • 4) one of: resumption, routine pointer, virtual routine

ii. 2 marks 3) is used for a non-recoverable raise, while 4) is used for a recoverable raise.

iii. 1 mark Code is restricted to all being in one large/monolithic program.
(e) 2 marks

<table>
<thead>
<tr>
<th>Exception Model</th>
<th>xxx</th>
<th>yyy</th>
</tr>
</thead>
<tbody>
<tr>
<td>termination</td>
<td>throw or _Throw</td>
<td>catch</td>
</tr>
<tr>
<td>resumption</td>
<td>_Resume</td>
<td>_CatchResume</td>
</tr>
</tbody>
</table>

2. (a) i. 1 mark Allows handlers to be set up before receiving exceptions.
         ii. 1 mark Turn on by inserting `__Enable` block.
         iii. 1 mark Only 3 points: on entry to `__Enable`, suspend, resume
         iv. 2 marks The source execution has a proxy raise that stands in for the actual (hidden) raise in the faulting execution. The proxy raise does NOT raise an exception in the source execution; the actual raise occurs at one of poll points in the faulting execution. 

(μC++ manual: “The faulting execution polls at certain points to check for pending nonlocal-exceptions; when nonlocal exceptions are present, the oldest matching exception is propagated (FIFO service) as if it had been raised locally at the point of the poll.”)

This is important since if using a resumption handler, execution after the handler completes continues from the proxy raise point.

(b) 2 marks Yes. Upon not finding a matching resumption handler, `defaultResume` executes, which raises the exception as a termination exception. This would then match the existing handler.

(c) 2 marks An unhandled exception raised by a coroutine raises the nonlocal exception `UnhandledException` at the coroutine’s last resumer.

(d) i. 1 mark It uses the caller’s runtime stack to store the coroutine object’s state.
     ii. 1 mark Can only suspend/yield in its own main routine i.e. cannot call another routine and suspend/yield there.
     iii. 1 mark Any one of: Simula, CLU, C#, Ruby, Python, JavaScript, Lua, F#, Boost

3. (a) i. 1 mark They both have state information and a thread.
     ii. 1 mark A process has its own memory while tasks share memory (they may mention that “it is reduced along some particular dimension”, which comes from the course notes.

(b) i. 1 mark implicit: any one of actor (Erlang, Scala), COFOR, COBEGIN/COEND
     ii. 1 mark explicit: any one of START/WAIT (FORK/JOIN), μC++ (_Task), pthreads, Go

(c) i. 1 mark We used the term `flicker` to describe a variable cycling multiple times between the original and new values before settling on the new value.
     ii. 2 marks If some task (let’s say t0) was the last one through the critical section, `::Last = t0` and its attempt to set its intent to `DontWantIn` in line 9 flickers to `DontWantIn`, t1 will enter the critical section and set `::Last = t1`. When t0’s intent flickers back to `WantIn`, t1 will back down and wait since it has a lower priority as the last one in; however, if t0 then terminates after finally setting its intent to `DontWantIn`, t1 will never break out of the loop.
4. (a)  

i. 5 marks

\[
\begin{array}{ccc}
S1 & S2 \\
\downarrow & \downarrow \\
\downarrow & \downarrow \\
S4 & S3 \\
\downarrow & \\
S5
\end{array}
\]

ii. 6 marks There are many possible solutions. Will either have too many or too few constraints, depending upon the graph.

\[
\text{COBEGIN} \\
A = 1; & \quad \text{\texttt{// S1}} \\
B = 2; & \quad \text{\texttt{// S2}} \\
\text{COEND} \\
\text{COBEGIN} \\
C = A + B; & \quad \text{\texttt{// S3}} \\
D = 2 * A; & \quad \text{\texttt{// S4}} \\
\text{COEND} \\
E = C + D; & \quad \text{\texttt{// S5}}
\]

No, the maximum concurrency cannot be achieved, e.g., S4 must wait for S2.

(b) 2 marks All lock implementations rely upon a spin lock, which busy-waits, at their core and thus cannot guarantee there is no starvation; however, in practice, a task rarely starves. At a higher level, we can ensure fairness.

(c) i. 2 marks Original solution did not release the lock before calling \texttt{yieldNoSchedule}, so the task blocked while holding the mutual exclusion lock, deadlocking the program.

ii. 2 marks If we add the lock release before the call to \texttt{yieldNoSchedule}, the task could be context-switched after adding itself to the queue of blocked tasks and dropping the lock but before the \texttt{yieldNoSchedule}, which would put it on the scheduler’s ready queue.

The task releasing the lock would see the blocked task in the queue, and also put it in the scheduler’s ready queue. Being in the scheduler’s ready queue twice is a problem.

iii. A. 1 mark The solution allowed \textit{barging}.

B. 1 mark \textit{Barging avoidance} and \textit{barging prevention}. 
5. **18 marks** Instead of throwing a local exception, could call a helper method, or use a goto. Needs to stop as soon as detects an error, though.

```cpp
// –1 if cannot remember how to convert a char to an int and has to resort to fake “int ctoi(char c);”
void IBAN::main() {
    try {
        int check, digits, sum = 0; // check BBAN
        for (int i = 0; i < 2; i+= 1) { // country code
            if (!isupper( ch )) { throw Error{}; }
            suspend();
        } // for
        if ( !isdigit( ch ) ) { throw Error{}; } // check digit
        check = ch - '0';
        if ( check < 2 ) { throw Error{}; }

        for ( digits = 0;; digits += 1 ) { // check digit
            suspend();
            if ( ch == '!') break;
            if ( digits == MAX_LENGTH | | !isdigit( ch ) ) { throw Error{}; }
            sum += ch - '0';
        } // for
        if ( 2 <= digits && sum % check == 1 ) {
            _Resume Match{} _At resumer();
        } else {
            _Resume Error{} _At resumer();
        } // if
    } catch( Error ) {
        _Resume Error{} _At resumer();
    } // IBAN::main()
}
```

Maximum of 9 marks if not using coroutine state to fullest effect.
6. (a) 3 marks
   1 COFOR(i, 0, rows,
   2     rotate( rotateVector[i], M[i], cols );
   3 );

(b) 10 marks
   struct WorkMsg : public uActor::Message {
     1 const unsigned int amount, cols;
     2 int * const row;
     1 WorkMsg( const unsigned int amount, int * const row, const unsigned int cols ) :
     1     Message(uActor::Delete), amount(amount), cols(cols), row(row) {}  
   1 }

   _Actor Rotate {
     1 Allocation receive( Message & msg ) {
     1     Case( WorkMsg, msg ) {
     1       rotate( msg_d->amount, msg_d->row, msg_d->cols );
     1     }
     1       return Finished;
     1     }
   1 }

   - uActorStart(); // start actor system
   1 Rotate workers[rows];
   1 for ( unsigned int i = 0; i < rows; i += 1 ) {
   1     workers[i] | new WorkMsg( rotateVector[i], M[i], cols );
   1     } // for
   - uActorStop();
i. 3 marks

```cpp
void main() {
    rotate(rotation, row, cols);
}
```

```cpp
public:
rotate( rotation, row, cols ) :
    row(row), cols(cols), rotation(rotation) {}
};
```

ii. 5 marks

```cpp
int main() {
    unsigned int rows, cols, r, c;
    // read in rows and cols
    int M[rows][cols], rotateVector[rows];
    // read in M and rotateVector
    Rotate *workers[rows];
    for ( r = 0; r < rows; r += 1 ) {
        workers[r] = new Rotate( matrix[r], M, rotateVector[r] );
    }
    for ( r = 0; r < rows; r += 1 ) {
        delete workers[r];
    }
    // wait for completion and delete tasks
    for ( r = 0; r < rows; r += 1 ) {
        for ( c = 0; c < cols; c += 1 ) {
            cout << M[r][c] << ' ';
        }
        cout << endl;
    }
    // main
```