UNIVERSITY OF WATERLOO

Midterm Examination
Fall 2015

Computer Science 343
Concurrent and Parallel Programming
Sections 001, 002, 003

Duration of Exam: 1 hour 50 minutes
Number of Exam Pages (including cover sheet): 6
Total number of questions: 6
Total marks available: 100

CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED

Instructors: Peter Buhr and Ashif Harji

November 4, 2015

1. (a) **1 mark** What is the primary benefit of using an exit in the middle of a loop?

   (b) **1 mark** Why should a loop exit NOT have an **else** clause?

   (c) **1 mark** Multi-level exits allow the elimination of all ________ *put answer in booklet* ________.

   (d) **2 marks** How does a *Sequel* provide simple exception handling?

   (e) **2 marks** Why does C longjmp execute faster than C++ **throw**?

   (f) **2 marks** Explain how a *non-matching handler* is executed for the following resumption raise.

```
_Event E {};
try {
    _Resume E();      // match with _CatchResume
} catch( E ) {};      // handles the resumption
```

2. (a) **2 marks** What class of problems does a coroutine handle better than the basic control-flow constructs?

   (b) **2 marks** When a coroutine's main terminates (returns), it resumes its starter coroutine rather than its last resumer coroutine.

   Why does this special semantics work for 80% of semi- and full-coroutines?

   (c) **2 marks** What actions occur in μC++ if a coroutine does not handle an exception?

   (d) **2 marks** Why is starting a full-coroutine cycle difficult?

3. (a) **2 marks** Two threads executing the statement i += 1 **twice** per thread, and i is initialized to 0. State all possible values of i after the increments by the threads.

   (b) **2 marks** What is the relationship between a user and kernel thread?

   (c) **2 marks** Dekker's algorithm has *unbounded overtaking*. Explain what that means.

   (d) **3 marks** Peterson's 2-thread software-solution for mutual exclusion uses a race in the entry protocol:

```
1    me = WantIn;    // entry protocol, order matters
2    ::Last = &me;   // RACE!
```

   Explain the failure situation if these two lines are interchanged?

   (e) **2 marks**

      i. State the special property of a hardware atomic instruction that allows it to solve mutual exclusion.

      ii. To achieve this special property what rule of the critical-section game is violated. (State the rule in words not by a number.)

   (f) **6 marks** The fetch-and-decrement instruction atomically reads and decrements a memory location. The instruction has one parameter: a reference to a value. The action of this single atomic instruction is the same as the following C routine if it could be performed as a single atomic operation without interruption with respect to the given parameter:

```
int fetchDec( int &val ) {      // atomic execution
    int temp = val;             // read
    val -= 1;                   // increment and write
    return temp;                // return previous value
}
```

   Show how this atomic instruction can be used to build mutual exclusion by completing a class with the following public interface (you may only add a public destructor and private members):

```
class Lock {
    . . .                       // YOU ADD HERE
  public:
    Lock();                     // YOU WRITE THESE ROUTINES
    void entryProtocol();
    void exitProtocol();
};
```

which is used as follows:

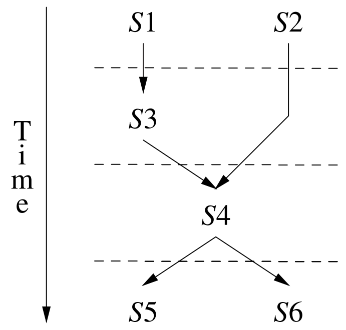
```
Lock lock;

lock.entryProtocol();
// critical section
lock.exitProtocol();
```

Your solution may use busy waiting, must deal with starvation, and **you cannot assume assignment is atomic**. State any assumptions or limitations of your solution. (Hint: think of "tickets" from the Bakery algorithm.)

4. (a) **1 mark** How is barging prevented when implementing a mutex lock?

   (b) **5 marks** Categorize the following types of locks as either being used solely for synchronization, solely for mutual exclusion, or could be used for either synchronization or mutual exclusion:
   (i) spin lock
   (ii) owner lock
   (iii) condition lock
   (iv) barrier
   (v) semaphore

   (c) **7 marks** The following precedence graph shows the optimal concurrency possible for a series of statements $S1..S6$:



   Code the statements using only *one* COBEGIN and COEND in conjunction with *binary* semaphores using P and V to achieve the concurrency of the precedence graph. Use pseudo-code for this problem, not μC++. Use BEGIN and END to make several statements into a single statement and show the initial value (0/1) for all semaphores. Name your semaphores Ln, e.g., $L1, L2, ...$, to simplify marking.

5. **22 marks** Write a *semi-coroutine* with the following public interface (you may only add a public destructor and private members):

```
_Coroutine Grammar {
  public:
    _Event Match {};          // characters form a valid string in the language
    _Event Error {};          // last character results in string not in the language
  private:
    char ch;                  // character passed by cocaller
    void main();              // YOU WRITE THIS ROUTINE
  public:
    void next( char c ) {
        ch = c;
        resume();
    }
};
```

which verifies a string of characters matches the language $X_i^+(YZ)_j^+W_{i+j}^+$, i.e., one or more X characters totalling i characters, followed by one or more pairs of characters YZ totalling j pairs of characters, followed by one or more W characters totalling $i+j$ characters, where $X \neq Y$ and $Z \neq W$, e.g.:

valid strings	invalid strings
xyzww	xyzw
aabcbcdddd	abcbcdd
33330077777	33300007777
##$%$%$%$%@@@@@@	##$%$%$%$%@@@@@

After creation, the coroutine is resumed with a series of characters (one character at a time). The coroutine accepts characters until:

- the characters form a valid string in the language, and it then raises the exception Grammar::Match at the last resumer;
- the last character results in a string not in the language, it then raises the exception Grammar::Error at the last resumer.

After the coroutine raises a Match or Error exception, it must terminate; sending more characters to the coroutine after this point is undefined. (You may use multiple **return** statements in Grammar::main.)

Write **ONLY** Grammar::main, do **NOT** write a main program that uses it! **No documentation or error checking of any form is required.**

Note: Few marks will be given for a solution that does not take advantage of the capabilities of the coroutine, i.e., you must use the coroutine's ability to retain data and execution state.

6. **31 marks** Divide and conquer is a technique that can be applied to certain kinds of problems. These problems are characterized by the ability to subdivide the work across the data, such that the work can be performed independently on the data. In general, the work performed on each group of data is identical to the work that is performed on the data as a whole. What is important is that only termination synchronization is required to know the work is done; the partial results can then be processed further.

   Write a program to *efficiently* check if a matrix of size $N \times N$ is a diagonally-symmetric matrix. (Notice, the matrix *must* be square and assume $N \leq 10$.) A diagonally-symmetric matrix has identical values along the diagonal and is equal to its transpose, i.e., $M = M^T$. That is, given $A = a_{i,j}$, then

$a_{0,0} = a_{i,i}$ and $a_{i,j} = a_{j,i}$, for all indices i and j. The following are all diagonally-symmetric matrices:

$$\begin{pmatrix} \mathbf{1} \end{pmatrix} \quad \begin{pmatrix} \mathbf{3} & 2 \\ 2 & \mathbf{3} \end{pmatrix} \quad \begin{pmatrix} \mathbf{7} & 2 & 3 \\ 2 & \mathbf{7} & 4 \\ 3 & 4 & \mathbf{7} \end{pmatrix} \quad \begin{pmatrix} \mathbf{-1} & 2 & 3 & 4 & 5 \\ 2 & \mathbf{-1} & 4 & 5 & 6 \\ 3 & 4 & \mathbf{-1} & 6 & 7 \\ 4 & 5 & 6 & \mathbf{-1} & 8 \\ 5 & 6 & 7 & 8 & \mathbf{-1} \end{pmatrix}$$

Solve the problem using task objects not the COFOR statement. Create one task per row of the matrix to concurrently check the values of that particular row for the diagonal-symmetric property. Each task has the following interface (you may only add a public destructor and private members):

```
_Task DiagSymmetric {
    void main();
  public:
    _Event Stop {};                            // stop checking
    DiagSymmetric( const int M[ ][10],         // matrix to check for diagonal-symmetric
                   const int row,              // row to be checked
                   const int cols,             // number of columns in row
                   bool &result                // set true if diagonal-symmetric; false otherwise
               );
};
```

Marks will be deducted for any unnecessary checking to prove the matrix is diagonal-symmetric.

Member uMain::main does the following:

- reads from standard input the matrix dimensions ($N \times N$, where $1 \leq N \leq 10$)
- checks the matrix dimensions are square (print a message and stop if not square)
  No other error checking of any form is required.
- declares any necessary matrix, arrays and variables
- reads (from standard input) and prints (to standard output) the matrix
- concurrently checks the matrix for diagonal-symmetry
  During deletion of the checking tasks, if the matrix is marked as not diagonal-symmetric, raise the concurrent exception Stop at the undeleted tasks to stop checking.
- and prints a message to standard output if the matrix is or is not symmetric.

Your answer must be a complete, working μC++ program.

An example of input for the above is:

```
4 4             matrix dimensions

7 2 3 4         matrix values
2 7 4 5
3 4 7 6
4 5 6 7
```

(The phrases "*matrix dimensions*" and "*matrix values*" do not appear in the input.) In general, the input format is free form, meaning any amount of white space may separate the values.

An example of output for the above is:

```
7, 2, 3, 4,          original matrix
2, 7, 4, 5,
3, 4, 7, 6,
4, 5, 6, 7,

matrix is diagonal-symmetric
```

(The phrase "*original matrix*" does not appear in the output.)