UNIVERSITY OF
WATERLOO

Midterm Examination
Fall 2016

Computer Science 343
Concurrent and Parallel Programming
Sections 001, 002, 003, 004

Duration of Exam: 1 hour 50 minutes
Number of Exam Pages (including cover sheet): 5
Total number of questions: 6
Total marks available: 106

CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED

Instructors: Peter Buhr and Aaron Moss

November 2, 2016

1. (a) i. **1 mark** What is the problem with the following code:

```
f( x );
while ( Condition ) {
    ...
    f( x );
}
```

ii. **3 marks** Rewrite the code to eliminate the problem.

(b) i. **1 mark** What is the problem with the following code:

```
for ( ;; ) {
    ...
    if ( i >= 10 ) break;
    ...
    if ( j >= 10 ) break;
}
if ( i < 10 ) E1;
else E2;
```

ii. **2 marks** Rewrite the code to eliminate the problem.

(c) **1 mark** The **goto** statement should only be used to simulate *put answer in booklet*.

(d) **2 marks** The labelled **break** restricts the **goto** in what two ways?

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

(f) **2 marks** Both the termination and resumption *raise* call a handler. How is a *raise call* different from a *routine call*?

2. (a) **2 marks** Coroutines are described as being *input* or *output*. Explain both kinds of coroutine.

(b) **2 marks** Coroutine resume and suspend perform a *context switch* to inactivate and activate a coroutine. Explain what a context switch does.

(c) **2 marks** Each stackfull coroutine has its own stack. What programming issues arise when there are multiple stacks in a program.

(d) **2 marks** Why are non-local exceptions *disabled* when a coroutine is first resumed?

(e) **3 marks** What are the 3 phases of any full-coroutine program? (Name them; do not explain them.)

3. (a) **4 marks** μC++ has two threading modes: uniprocessor and multi-processor. Draw the *threading model* diagrams for each mode.

(b) **4 marks** Complete the following code fragment by concurrently summing the rows of a matrix using the COFOR statement.

```
#include <uCobegin.h>
void uMain::main() {
    const int rows = 10, cols = 10;
    int matrix[rows][cols], subtotals[rows], total = 0;
    // read matrix

    // WRITE CODE TO SUM ROWS USING COFOR

    for ( int r = 0; r < rows; r += 1 ) {
        total += subtotals[r];  // total subtotals
    }
    cout << total << endl;
}
```

(c) **2 marks** Define the terms *critical section* and *mutual exclusion*.

(d) **2 marks** What assumption does Peterson's algorithm require from the underlying hardware? Why is this assumption *cheating* with respect to creating mutual exclusion?

(e) **2 marks** Peterson's algorithm has *bounded overtaking*. Explain what that means.

(f) **5 marks** Given the following atomic swap-instruction, use it to create an N-thread mutual-exclusion lock. Starvation is allowed.

```
void Swap( int &a, &b ) {
    int temp;
    // begin atomic
    temp = a;
    a = b;
    b = temp;
    // end atomic
}
```

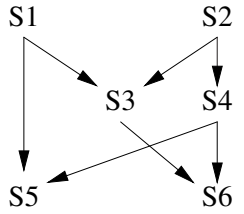
4. (a) **4 marks** Explain the difference between *spinning* and *blocking* locks.

(b) **2 marks** Do blocking locks eliminate spinning, and if not, why not.

(c) **4 marks** Show a *lock-release pattern* to ensure a mutual-exclusion lock is always released.

(d) **2 marks** Why is synchronization more complex for blocking locks than spinning locks?

(e) **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. **16 marks** Write a *semi-coroutine* with the following public interface (you may only add a public destructor and private members):

```
_Coroutine Compact {
    char ch;                    // character passed by cocaller
    void main();                // YOU WRITE THIS MEMBER
  public:
    void next( char c ) {
        ch = c;
        resume();
    }
};
```

which receives a sequence of characters, compacts whitespace (newline, blank and tab characters), and writes out the compacted sequence of characters to standard output (cout). Compacting means whitespace at the start and end of a line is removed, and whitespace within a line is reduced to a

single blank. Lines containing non-whitespace characters are terminated with a newline; nothing is printed for lines containing only whitespace characters. Terminate Compact::main when it receives the sentinel character `'\377'`; do not print the sentinel character. For example, given "input" as a character sequence passed to the coroutine one character at a time, where the symbol ⋆ represents a blank or tab, the coroutine writes out the "output" character sequence:

input sequence	output sequence
⋆⋆⋆⋆⋆⋆start⋆⋆⋆of⋆⋆⋆⋆⋆text⋆⋆⋆⋆⋆⋆⋆ more⋆⋆⋆⋆⋆⋆text ⋆⋆last⋆text⋆⋆⋆⋆⋆ ⋆⋆⋆ \377	start⋆of⋆text more⋆text last⋆text

Write **ONLY** Compact::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. **27 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 **COMPLETE** μC++ program to *efficiently* check if all the rows of a matrix of size $N \times M$ are identical. For example, in:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

   all the rows have the same values. The matrix is checked concurrently along its rows. Create the minimal number of tasks to check the rows for equality that has maximum concurrency. Each task has the following interface (you may only add a public destructor and private members):

```
bool stop = false;                  // global variable: true => stop all work
_Task EqualRows {
    ...                             // YOU ADD HERE
    void main();                    // YOU WRITE THIS ROUTINE
  public:
    _Event Stop {};                 // concurrent exception
    EqualRows( const int row1[],    // one row of the matrix
               const int row2[],    // another row of the matrix
               const int cols       // number of columns in row
             );                     // YOU WRITE THIS ROUTINE
};
```

uMain::main reads from standard input the matrix dimensions ($N \times M$), declares any necessary matrix, arrays and variables, reads (from standard input) and prints (to standard output) the matrix, concurrently checks the matrix values in each row, and prints a message to standard output if the matrix has equal rows or does not have equal rows. **No documentation or error checking of any form is required.**

As an optimization, the global (flag) variable stop is set when a task finds an unequal row (may be set multiple times), and uMain::main performs a resumption raise of exception Stop at any non-deleted tasks once the stop flag indicates the matrix has unequal values. When the concurrent Stop exception is propagated, each checking task stops performing the equality check and terminates.

An example of the input for the program is:

```
3  5              matrix dimensions

1 2 3 4 5         matrix values
1 2 3 4 5
1 2 3 4 5
```

(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.

Example outputs are:

```
1, 2, 3, 4, 5,     original matrix
1, 2, 3, 4, 5,
1, 2, 3, 4, 5,

matrix does have equal rows
```

```
1, 2, -1, 4, 5,    original matrix
1, 2, 3, 4, 5,
1, 2, 3, 4, 5,

matrix does not have equal rows
```

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