UNIVERSITY OF
WATERLOO

Midterm Examination
Fall 2022

Computer Science 343
Concurrent and Parallel Programming
Sections 001, 002

Duration of Exam: 2 hours
Number of Exam Pages (including cover sheet): 5
Total number of questions: 5
Total marks available: 111

CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED

Instructor: Peter Buhr

November 3, 2022

1. (a) **4 marks** What is the software engineering problem in this code fragment and write the code to fix it?

```
cin >> d;
while ( ! cin.fail() ) {
    ... // LOOP BODY
    cin >> d;
}
```

   (b) **2 marks** Define the term *flag variable*.

   (c) **1 mark** To eliminate all flag variables, what control-flow capability is required?

   (d) **1 mark** When is a flag variable necessary?

   (e) **2 marks** Explain the two kinds of dynamically-sized objects. Which can appear on the stack?

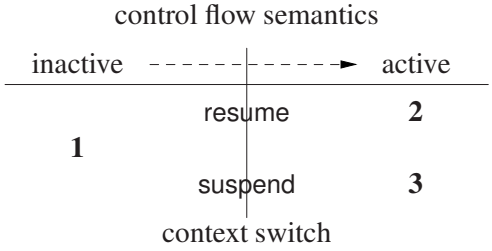
   (f) **1 mark** What is the control-flow issue for all return-code approaches, when unwinding multiple levels of routine calls?

   (g) **2 marks** Explain why a *label variable* for nonlocal transfer must be a tuple of two values.

   (h) **2 marks** Explain the fundamental difference between termination and resumption *propagation*.

2. (a) **2 marks** Explain what new control-flow property coroutines add to function-call semantics.

   (b) **3 marks** Name the coroutine that becomes inactive/active at locations **1**, **2**, and **3**, below.



   (c) **2 marks** What property is necessary for full coroutining and why is it difficult to create this property?

   (d) i. **2 marks** Explain the terms *stackless* and *stackful* coroutine.

   ii. **1 mark** Which is more powerful and why?

   (e) **2 marks** What is a non-local exception and why is it more complex for coroutines than tasks?

3. (a) **2 marks** Explain the difference between *preemptive* and *non-preemptive* concurrent scheduling.

   (b) **2 marks** Explain the difference between *implicit* and *explicit* concurrency.

   (c) **3 marks** Explain *user* and *real* time. Which time changes when concurrent speedup occurs?

   (d) **1 mark** Define *critical path* within concurrent speedup.

   (e) **2 marks** Explain why COBEGIN/COEND is not as expressive as START/WAIT.

   (f) **1 mark** Does an actor have a thread?

   (g) **2 marks** Briefly explain *mutual exclusion*.

   (h) **2 marks** Explain *unbounded* and *bounded* overtaking.

   (i) **2 marks** Explain the difference between a *spinning* and *blocking* lock.

   (j) **2 marks** Explain the difference between barging *avoidance* and *prevention*.

4. **20 marks** Write a *semi-coroutine* with the following public interface (you may only add a public destructor and private members):

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

which receives a sequence of characters forming sentences, capitalizes the first letter of each sentence, and writes out the capitalized sentences to standard output (cout). The exception Eof is raised at coroutine Caps by the program main when there are no more characters, indicating the coroutine must terminate.

A sentence starts with a letter preceded by whitespace characters (space, tab, newline) that follows a period, question mark, or exclamation point. If the starting letter is lower case, the filter transforms the letter to upper case. There is a special case for the first letter received if there is no preceding punctuation character. For example, given the input characters, the coroutine writes out the output characters, where the letters to be capitalized and the capitalizations are in **large bold** font.

input

```
the first letter in this line should be capitalized.  what about the next one?
that too!but not after the bang. 123 cannot be capitalized.
No space after this punctuation.so no capital.? !hi
and not another sentence. !hi!
and another sentence.
.. this is a sentence.
??? and this, too!
.
```

output

```
The first letter in this line should be capitalized.  What about the next one?
That too!but not after the bang. 123 cannot be capitalized.
No space after this punctuation.so no capital.? !hi
and not another sentence. !hi!
And another sentence.
.. This is a sentence.
??? And this, too!
.
```

Assume the existence of the following built-in routines.

ispunc(c) – returns true for a period, question mark, or exclamation point.
isspace(c) – returns true for a white-space characters (space, tab, newline).
toupper(c) – returns the upper case versions of a character, if possible, and otherwise returns c.

Write **ONLY** Caps::main, do **NOT** write a program main 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.

5. 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 the following μC++ code fragments to *efficiently* check if all the rows of a matrix of size $N \times M$ are identical. The following matrices have identical rows (including the empty matrix).

$$() \quad \begin{pmatrix} 7 \end{pmatrix} \quad \begin{pmatrix} -3 & 4 \\ -3 & 4 \end{pmatrix} \quad \begin{pmatrix} 1 & 2 & 3 & -4 & 5 \\ 1 & 2 & 3 & -4 & 5 \\ 1 & 2 & 3 & -4 & 5 \end{pmatrix}$$

   A matrix is checked concurrently along its rows. Perform this check using the minimal number of (logical) threads required to achieve maximum concurrency.

   Assume the following code in the program main.

```
int main() {
    int rows, cols;
    cin >> rows >> cols;
    int M[rows][100], r, c;
    // read/print matrix
    bool notEqual = false;                  // used for output
    uBaseTask & prgMain = uThisTask();      // program main' s task id for equalCheck

    // COFOR/ACTOR/TASK code to drive the concurrent solutions

    cout << "matrix is" << (notEqual ? " not " : " ") << "equal rows" << endl;
}
```

   (a) **4 marks** Write a sequential function with the following interface to check if two rows have equal values.

```
_Event NotEqual {};                     // not equal rows
void equalCheck(
    const int row1[],                   // row to check
    const int row2[],                   // row to check
    const int cols,                     // columns in row
    uBaseTask & pgmMain                 // contact if not equal rows
);
```

   where row1/row2 are the rows to check for equal values, cols is the number of columns in a row, and pgmMain is the program-main task. If the function determines the two rows are are *NOT* equal, it raises the exception NotEqual at the program main and returns. **Note:** a concurrent non-local exception works between a COFOR thread and the program main thread; similarly, it works between an actor executor thread and the program main thread.

   (b) **6 marks** Write a fragment of the program main using a COFOR statement that calls the equalCheck function to check the matrix rows concurrently.

(c) **7 marks** Write a message and actor type with the following interface that uses the equalCheck function to check the matrix rows concurrently.

```
struct WorkMsg : public uActor::Message {
    // YOU WRITE THIS TYPE
}; // WorkMsg
_Actor EqualRows {
    Allocation receive( Message & msg ) {
        // YOU WRITE THIS MEMBER
    } // EqualRows::receive
}; // EqualRows
```

The program main in question 5(e)i creates these messages/actors and passes each actor a message containing all the information needed to call function equalCheck.

(d) **7 marks** Write a task type with the following interface that uses the equalCheck function to check the matrix rows concurrently (you may only add a public destructor and private members).

```
_Task EqualRows {                       // check rows
  public:
    _Event Stop {};                     // concurrent exception
  private:
    // YOU ADD MEMBERS HERE
    void main() {
        // YOU WRITE THIS MEMBER
    } // EqualRows::main
  public:
    EqualRows( ... ) {
        // YOU WRITE THIS MEMBER
    }
};
```

The program main in question 5(e)ii creates these tasks and passes via its constructor all the information needed to call function equalCheck.

As an optimization, if the program main receives the concurrent NotEqual exception, it raises exception EqualRows::Stop at any non-deleted EqualRows tasks. When the concurrent Stop exception is propagated in a EqualRows task, it stops performing its row check, and returns.

(e) With respect to the body of the program main write:

i. **9 marks** For the actor implementation, create the actor system and handle the NotEqual exception. Create the actors on the stack and dynamically allocate the messages.

ii. **12 marks** For the task implementation, create the task system, handle the NotEqual exception, and raise exception EqualRows::Stop at any non-deleted EqualRows tasks. **Note:** *all* tasks must be created, even if a NotEqual exception is raised during creation.

No documentation or error checking of any form is required.