UNIVERSITY OF WATERLOO

Midterm Examination
Winter 2025

Computer Science 343
Concurrent and Parallel Programming
Sections 001

Duration of Exam: 2 hours
Number of Exam Pages (including cover sheet): 5
Total number of questions: 5
Total marks available: 116

CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED

Instructor: Peter Buhr

March 5, 2025

1. (a) **1 mark** Many C++ programmers use the following coding pattern to eliminate duplicate code.

```
while ( cin >> i ) {
    . . .
}
```

Explain how it works.

(b) **5 marks** Rewrite the following code fragment using only **if**, labels, and **goto**s; no **else** or compound-statement "{}".

```
if ( C ) {
    S1;
} else {
    S2;
}
```

(c) **1 mark** Why is out-denting loop exits good *eye-candy*?

(d) **2 marks** Simplify the following code pattern:

```
for ( ;; ) {
    S1
  if ( C1 ) break;
    S2
  if ( C2 ) break;
    S3
}
if ( C1 ) E1;
else E2;
```

(e) **1 mark** Explain the term *flag variable*.

(f) **1 mark** Explain the primary code pattern that necessitates the use of the heap allocation versus the stack.

(g) **2 marks** Explain the term *multiple/alternate outcomes* with respect to routine call.

(h) **2 marks** Explain the purpose of a *fixup routine*.

(i) **2 marks** Explain how recursion affects setting the value of a *label variable*.

(j) **2 marks** Explain the term *stack unwinding*.

(k) **2 marks** When an exception handler returns, give two possible transfer points.

2. (a) **2 marks** With respect to calling a coroutine's interface member, when does the **this** and uThisCoroutine variable change?

(b) **1 mark** When a coroutine's main returns, which coroutine is resumed?

(c) **1 mark** If two coroutines are in a resume-resume cycle, what happens if both coroutines do suspends?

(d) **2 marks** If two coroutines are in a resume-resume cycle, do the coroutine stacks grow in relation to the number of cycles? If not, why not?

(e) **2 marks** How is a generator/iterator different from a coroutine?

(f) **2 marks** Explain what the **_Enable** statement does, and why it is necessary with respect to non-local exceptions among coroutines?

3. (a) **1 mark** When is it wrong for a C++ destructor to raise an exception?

(b) **3 marks** Show why the statement `i += 1` is not safe in a concurrent program.

(c) **2 marks** What is the difference between *implicit* and *explicit* concurrency?

(d) **2 marks** Give the definition of *speedup* in parallel execution.

(e) **1 mark** Explain the term *critical path* with respect to parallel speedup.

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

(g) **1 mark** Explain rule 4 (liveness) of the mutual exclusion game.

(h) **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
}
```

(i) **1 mark** How do blocking locks reduce busy waiting?

(j) **2 marks** Explain barging avoidance.

4. **14 marks** Given the following abstract filter class.

```
_Coroutine Filter {
  protected:
    _Exception Eof {};                      // no more characters
    Filter * next;                          // next filter in chain
    unsigned char ch;                       // communication variable
  public:
    Filter( Filter * next ) : next( next ) {}
    void put( unsigned char c ) {
        ch = c;
        resume();
    }
};
```

Write the following *semi-coroutine* filter with the given public interface (you may only add a public destructor and private members), which counts the number of strings "Fred" in the input stream and passes all the text to the next filter.

```
_Coroutine Find : public Filter {
    void main() {
        const char word[ ] = "Fred";
        unsigned int len = sizeof( word ) - 1;
        unsigned int cnt = 0;
        // YOU WRITE THIS CODE
    }
  public:
    Find( Filter * f ) : Filter( f ) {}
};
```

The string "Fred" can appear anywhere, e.g., in this input sequence:

Fred
FreFre**Fred**stutter
asd ad **Fred** adsd **Fred**
FFre**Fred**Fre

the filter generates the following output at end-of-file.

```
5 Fred
```

Write **ONLY** the designated portion of Find::main; do **NOT** write any other filters or main program that uses them!

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* find the minimum and maximum values in an $N \times M$ matrix, if and only if each row has a *unique* min/max value. A non-unique row has min == max.

   non-unique
   $$\begin{pmatrix} \mathbf{7} \end{pmatrix}$$
   $min == max, row\ 1$

   unique
   $$\begin{pmatrix} -1 & 4 \\ -1 & 4 \end{pmatrix}$$
   $min\ -1, max\ 4$

   non-unique
   $$\begin{pmatrix} \mathbf{1} & \mathbf{1} & \mathbf{1} \\ 1 & 2 & 3 \\ \mathbf{3} & \mathbf{3} & \mathbf{3} \end{pmatrix}$$
   $min == max, rows\ 1,3$

   unique
   $$\begin{pmatrix} 1 & 1 & 3 & 4 \\ 2 & 2 & 3 & 3 \\ 3 & 4 & 6 & 6 \end{pmatrix}$$
   $min\ 1, max\ 6$

   Assume the following declarations in the program main.

```
unsigned int rows, cols;
int M[rows][cols], rmin[rows], rmax[rows];
int min = INT_MAX, max = INT_MIN;
unsigned int nonunique = 0;
```

   For the COFOR and actor solutions, find the overall min/max from arrays rmin and rmax and put them into variables min/max. (For the actor solution, just put a comment where this code goes rather than copying it again.) As well, count the number of non-unique rows in the matrix from the non-local exceptions raised.

   (a) **5 marks** Write a sequential routine to find the unique min and max in the row of an array.

```
_Exception NonUnique {}; // row is non-unique
void minmax( const int row[ ], const unsigned int cols,
             int & min, int & max, uBaseTask & prgMain ) {
    // YOU WRITE THIS FUNCTION
}
```

   where row is the matrix row to test, cols is the number of columns in the row, min/max are output variables for the results, and pgmMain is the address of the program-main task. If the routine determines the tested row is non-unique, it raises the exception NonUnique at the program main and returns. Note, a concurrent non-local exception works between the COFOR and actor executor threads, and the program-main thread.

   (b) **9 marks** Write a COFOR statement to appear in the program main, where each iteration of the COFOR uses routine minmax to find the min/max for its matrix row. **Warning**, uThisTask() inside the COFOR returns the task id of a thread created by COFOR not the program-main task.

   (c) **8 marks** Write a message and actor with the following interface.

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

WorkMsg contains the information needed by MinMax to compute a row's min/max values.

(d) **8 marks** Write the statements necessary for the program main to start the actor system, create a MinMax actor per row on the stack, and send each actor an appropriately-initialized dynamically-allocated WorkMsg message to start it.

(e) **8 marks** Write a task with the following interface (you may only add a public destructor and private members):

```
_Task MinMax {                      // check row of matrix
  public:
    _Exception Stop {};             // concurrent exception
  private:
    // YOU ADD MEMBERS
    void main() {
        // YOU WRITE THIS MEMBER
    } // MinMax::main
  public:
    MinMax(                         // YOU WRITE THIS MEMBER
        const int row[ ],           // matrix row
        int cols,                   // row columns
        int & min, int & max        // output returns
        uBaseTask & pgmMain         // contact if non-unique min/max
    );
};
```

The program main raises the concurrent Stop exception at a MinMax task, once it determines a matrix row is non-unique. The exception means stops performing the row check and return immediately.

(f) **14 marks** Write the statements necessary for the program main to create the MinMax tasks on the heap, and then delete each task. When the *first* concurrent NonUnique exception is caught, a MinMax::Stop is raised at any non-deleted MinMax tasks. **Note**, the program main must create *all* tasks, even though a NonUnique exception can be raised during creation.

No documentation or error checking of any form is required.