UNIVERSITY OF
WATERLOO

Midterm Examination
Fall 2018

Computer Science 343
Concurrent and Parallel Programming
Sections 001, 002

Duration of Exam: 1 hour 50 minutes
Number of Exam Pages (including cover sheet): 6
Total number of questions: 6
Total marks available: 108

CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED

Instructor: Peter Buhr

October 31, 2018

1. (a) **4 marks** Rewrite the following **switch** statement using only **if**, **goto**s, and labels; no **else** or compound-statements "{}".

```
switch ( i ) {
  case 1:
    // case 1 body
    break;
  default:
    // default body
}
```

   (b) **2 marks** What is *modularity* in the context of static multi-level exit, and what problems does it present?

   (c) **1 mark** True or False: routines have one outcome.

   (d) **1 mark** How many levels of stack unwinding does a variant return-type support?

   (e) **2 marks** Why does a label variable have a pointer to an activation block (stack frame)?

   (f) **2 marks** Explain the two different behaviours of setjmp.

   (g) **1 mark** The **goto** to a label variable is a dangerous way to do *nonlocal transfer*. What do modern programming languages do to restrict nonlocal transfer to make it safe?

   (h) **2 marks** In μC++ exception handling, the propagation mechanism may search the stack twice. Explain how this happens.

   (i) **1 mark** When an unterminated coroutine is deleted, why is its stack unwound before deletion?

   (j) **1 mark** Why are these objects allocated on the heap rather than the stack?

```
Obj * objs[size];
for ( int id = 0; id < size; id += 1 )
    objs[id] = new Obj( id );
```

   (k) **2 marks** If each thread has a separate heap, what issue arises if one thread passes one of its heap pointers to another thread?

2. (a) **2 marks** What is an output coroutine? What is an input coroutine?

   (b) **1 mark** What property of a μC++ coroutine allows modularization within the coroutine main?

   (c) **2 marks** When does an instance of a coroutine type transition from an object to a coroutine? When does an instance of a coroutine type transition from a coroutine to an object?

   (d) **1 mark** What is the reason for resuming a coroutine in its constructor?

   (e) **2 marks** When a *nonlocal exception* is propagated, why is it safest to nest the μC++ **_Enable** within the **try** block?

   (f) **2 marks** When a coroutine terminates, which coroutine does it context switch to and why?

   (g) **1 mark** What property transitions a coroutine from semi to full?

   (h) **1 mark** What does resume do when executed in a coroutine main?

3. (a) **1 mark** Explain the *M*:*N* threading model, where *M* is user threads and *N* is kernel threads.

   (b) **1 mark** What causes *sub-linear* speed up to eventually change to *non-linear*?

   (c) **2 marks** Is COBEGIN/COEND an implicit or explicit concurrency system? Is START/WAIT an implicit or explicit concurrency system?

   (d) **1 mark** Rule 3 of the mutual-exclusion game states:

> If a thread is not in the entry or exit code controlling access to the critical section, it may not prevent other threads from entering the critical section.

   What lock issue can be eliminated by controlling order/speed of execution in the entry/exit code?

(e) **2 marks** The following is Dekker's software solution for mutual exclusion of two tasks:

```
enum Intent {WantIn, DontWantIn};
Intent *Last;

_Task Dekker {
    Intent &me, &you;

    void main() {
        for (int i = 1; i <= 1000; i += 1) {
            for ( ;; ) {                              // entry protocol
                me = WantIn;
              if ( you != WantIn) break;
                if ( ::Last == &me ) {                // low or high priority ?
                    me = DontWantIn;
                    while ( ::Last == &me ) {}        // <**************
                } // if
            } // for
            CriticalSection();                        // critical section
            ::Last = &me;                             // exit protocol
            me = DontWantIn;
        } // for
    } // main
 public:
    Dekker(Intent &me, Intent &you) : me(me), you(you) {}
}; // Dekker
```

If the condition marked with asterisks (`'*'`) `Last == &me` is changed to `you == WantIn`, explain what rule of the critical-section game is violated and why.

(f) **3 marks** In the *Hehner-Shyamasundar* version of Lamport's bakery algorithm for N-thread software mutual-exclusion, two ticket values cannot be used: 0 and INT_MAX. Explain when these two special ticket values are used, and explain why the 0 value is necessary for correctness.

(g) **1 mark** In the *Arbiter* algorithm for N-thread software mutual-exclusion, how is fairness (rule 5) guaranteed?

4. (a) **2 marks** Explain why a *yielding* spin-lock is still busy waiting.

(b) **2 marks** Why are spin locks easier to implement than blocking locks?

(c) **1 mark** When implementing locks you "Need magic to atomically yield without scheduling and release a spin lock.". Explain a way to supply the "magic" to perform this action.

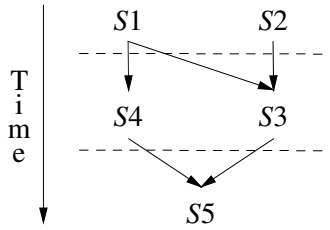
(d) **2 marks** Given the following μC++ code:

```
osacquire( cout ) << "abc " << "def " << endl;
```

explain the purpose of osacquire and how it works in this expression.

(e) **2 marks** What kind of lock is a barrier lock, and explain the kind of problem it handles.

(f) **6 marks** Given the following precedence graph:



construct an *optimal* solution, i.e., minimal threads and locks, using COBEGIN and COEND in conjunction with *binary* semaphores using P and V to achieve the precedence graph. 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. **17 marks** Write a *semi-coroutine* with the following public interface (you may only add a public destructor and private members):

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

which verifies a string of characters constitutes a valid North American telephone number. The string is described by the following grammar:

phoneno : $area_{opt}$ *trunk dash number* '\n'
area : '(' *3-digit-number* ')'
trunk : *3-digit-number*
dash : '–'
number : *4-digit-number*

where the quotation marks are metasymbols and not part of the described language, and $_{opt}$ means optional (0 or 1). The following are some valid and invalid phone numbers:

valid strings	invalid strings
(876)343-8760	789 6543
456-9807	(88)345-8790
786-5555	(888)45-8790
(800)555-1212	(888)345-879

Assume the C library routine isdigit; isdigit(c) returns true if c is a digit;

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 Phone::Match at the last resumer;
- the last character results in a string not in the language, it then raises the exception Phone::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 Phone::main.)

Write **ONLY** Phone::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. 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.

   (a) **4 marks** Write a routine to find the minimum and maximum in the row of an array:

   ```
   void minmax( const int row[], const unsigned int cols, int & min, int & max ) {
       // find minimum/maximum in row of length cols, and assign to min/max
   }
   ```

   (b) **2 marks** Given routine minmax above and the following declarations, write a COFOR statement to concurrently find the minimum/maximum in each row of matrix M.

   ```
   int M[rows][cols], rmin[rows], rmax[rows];
   COFOR( ...                          // YOU WRITE THIS STATEMENT
       ...
   );
   ```

   (c) **28 marks** Using routine minmax and the declarations above, write a **complete** μC++ program using *task objects* to find the minimum and maximum values of a matrix. For example, in:

   $$\begin{pmatrix} 1 & 21 & 3 & 4 & 5 \\ -11 & 2 & 32 & 4 & 50 \\ 1 & 2 & 3 & 45 & 5 \\ -11 & 2 & 3 & 4 & 5 \end{pmatrix}$$

   The minimum value is -11 and the maximum value is 50.

   Using the following interface (you may only add a public destructor and private members):

   ```
   _Event Equal {};                    // concurrent exception
   _Task MinMax {
       ...                             // YOU ADD HERE
       void main();                    // YOU WRITE THIS ROUTINE
     public:
       _Event Stop {};                 // concurrent exception
       MinMax( const int row[],        // row of matrix to search
               const int cols,         // number of columns in row
               int & min,              // location to store minimum value
               int & max               // location to store maximum value
               uBaseTask & prgMain     // program main task
             );                        // YOU WRITE THIS ROUTINE
   };
   ```

   create one MinMax task per row of the matrix to concurrently find the minimum/maximum values for that particular row.

   The program main reads from standard input the matrix dimensions ($N \times M$), reads (from standard input) and prints (to standard output) the matrix, concurrently finds the minimum/maximum values for each row, determines the overall minimum/maximum value for the matrix from the row minimum/maximum values, and prints the overall matrix minimum/maximum values. **No documentation or error checking of any form is required.**

   If a MinMax task finds its minimum and maximum values are equal, it raises the global concurrent exception Equal() at the pgmMain and then returns, and when the program main receives this

concurrent exception, it raises exception MinMax::Stop at any nondeleted MinMax tasks. When the concurrent Stop exception is propagated in a MinMax task, it stops looking and returns.

An example of the program input is:

```
4  5                    matrix dimensions
1 21 3 4 5              matrix values
-11 2 32 4 50
1 2 3 45 5
-11 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. You may assume the existence of the constants INT_MIN and INT_MAX, which never appear in the input.

An example of the program output is:

```
1, 21, 3, 4, 5,          original matrix
-11, 2, 32, 4, 50,
1, 2, 3, 45, 5,
-11, 2, 3, 4, 5,
min:-11  max:50
```

```
1, 1, 3, 7, 1,           original matrix
2, 2, 2, 2, 2,
1, 2, 3, 45, 5,
-1, -1, -1, -1, -1,
equal min/max
```

(The phrase "*original matrix*" does not appear in the output.) Note, the comma is a terminator not a separator.