UNIVERSITY OF
WATERLOO

# Midterm Examination
# Winter 2019

## Computer Science 343
## Concurrent and Parallel Programming
## Sections 001

**Duration of Exam: 1 hour 50 minutes**
**Number of Exam Pages (including cover sheet): 6**
**Total number of questions: 6**
**Total marks available: 113**

**CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED**

**Instructor: Peter Buhr**

**March 6, 2019**

1. (a) **2 marks** Rewrite the following code fragment to remove the last **if/else** statement but still retain the same behaviour:

```
for ( i = 0; ; i += 1 ) {              // linear search for key in list
    if ( i == size ) break;            // key not found
    if ( key == list[i] ) break;       // key found
}
// REMOVE
if ( i == size ) C1;                   // key not found
else C2;                               // key found
```

   (b) **1 mark** Describe a situation where a *flag variable* is necessary.

   (c) **4 marks** Given the following code fragment:

```
void f( int );
void g( void (*h)( int ) ) {
    try {
        ... f( 3 ); ...
        ... h( 3 ); ...
    } catch( E ) { ... }
}
```

   - Is the call to f a static or dynamic call?
   - Is the call to h a static or dynamic call?
   - Is the return from the **catch** clause a static or dynamic return?
   - Is the return from g a static or dynamic return?

   (d) **2 marks** Explain the problem in the following code fragment and how $\mu$C++ fixes the problem.

```
class B {};
class D : public B {};      // INHERITANCE
void f( B & t ) { ... throw t; ... } // RAISE
D m;
try {
    f( m );
} catch( D ) {...}          // CATCH
```

   (e) **2 marks** Explain when to use termination and resumption propagation.

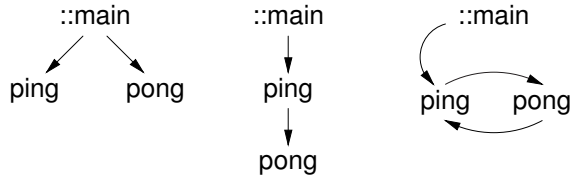
   (f) **2 marks** What is a potential problem using std::vector in a concurrent program?

2. (a) **2 marks** Explain the fundamental change in call/return control-flow provided by coroutines.

   (b) **2 marks** A $\mu$C++ coroutine has its own stack. What missing stack property does the $\mu$C++ coroutine stack have that can cause problems? Suggest a best possible solution to the problem.

   (c) **2 marks** Coroutines context switch during suspend/resume and resume/resume, but there are two special cases where context switching also occurs. Explain both.

   (d) **3 marks** Given the following diagram:




   - Name the creating coroutine and what it creates.
   - Name the starter coroutines and what they start.
   - Name the full coroutines and what makes them full.

(e) **2 marks** Generator/iterator coroutines often have restricted control-flow. What is the restriction and why does the restriction exist?

(f) **2 marks** After executing:

```
_Resume E() _At coroutine;
```

why does nothing happen in coroutine and what has to be done to make it happen?

3. (a) **2 marks** Explain two scenarios where i = 1 is not thread safe for another thread.

(b) **1 mark** Explain an advantage for using user-level threading.

(c) **3 marks** Name the 3 factors affecting speedup in a concurrent program? (Do not explain them.)

(d) **1 mark** Can synchronization and mutual exclusion be created without existing atomicity? (Yes/No)

(e) The following is the alternation solution to mutual exclusion, which beaks rule 3 of the mutual exclusion game:

> 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.

```
int Last = 0;                    // global
// each thread runs
while ( ::Last == me ) {}        // entry protocol
CriticalSection();               // critical section
::Last = me;                     // exit protocol
```

i. **1 mark** Explain how rule 3 is broken.

ii. **2 marks** Fix this solution using a test-and-set hardware instruction represented by this C interface: **int** TestSet( **int** & lock ).

(f) **1 mark** For tournament software-solutions for mutual exclusion, what special behaviour must the exit protocol do when retracting intents?

4. (a) **1 mark** Explain the notion of *try acquire* for a lock.

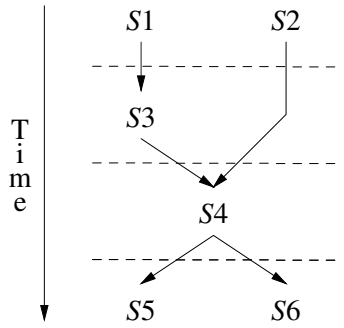
(b) **2 marks** Name the two basic implementation components present in most blocking locks.

(c) **1 mark** When multiple threads block on a blocking lock, what is the requirement on the unblocking order?

(d) **1 mark** Why are synchronization locks often the weakest form of blocking locks.

(e) **1 mark** How does the blocking binary-semaphore support both synchronization and mutual exclusion?

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

```
_Coroutine Grammar {
    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 matches the language $X_i^+ (YZ)_{i+1}^+ W_{i+2}^+$, i.e., 1 or more $X$ characters totalling $i$ characters, followed by 1 or more pairs of characters $YZ$ totalling $i+1$ pairs of characters, followed by 1 or more $W$ characters totalling $i+2$ characters, where $X \neq Y$ and $Y \neq W$, e.g.:

| valid strings | invalid strings |
|---:|---:|
| xyzyzxxx | xyzyf |
| aabcbcbccccc | abcbcbbbb |
| aabbbbbbcccc | aabbbbbccccc |
| 3330000000033333 | 3330000000034333 |
| ##$%$%$%@@@@ | ##$%$%$%$%##### |

After creation, the coroutine is resumed with a series of characters (1 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, and 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. 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.

An identity matrix has 1's along the major diagonal and 0's elsewhere, e.g., the following are all identity matrices:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$

(a) **4 marks** Write a routine to *efficiently* check if a row of a matrix of size *cols* has the identity property:

```
bool identityCheck( const unsigned int r, const int row[], const unsigned int cols ) {
    // return true if row "r" has the identity property; false otherwise
}
```

(b) **3 marks** Using routine identityCheck above and the following declarations, write a COFOR statement to concurrently check if each row of matrix M has the identity property:

```
bool identity = true;
int M[rows][cols];
COFOR( ...                              // YOU WRITE THIS STATEMENT
    ...
);
```

(c) **11 marks** Using routine identityCheck and the declarations identity and M above, write a message and actor to concurrently check if each row of matrix M has the identity property:

```
struct WorkMsg : public uActor::Message {
    // WRITE THIS TYPE
};
_Actor Identity {
    // WRITE THIS TYPE
};
// USING THE ACTOR AND MESSAGE TYPES TO PERFORM AN IDENTITY CHECK
```

All information needed to check a row is passed in the message to the actor.

(d) **7 marks** Using routine identityCheck and the declarations identity and M above, write a task type (you may only add a public destructor and private members) to concurrently check if each row of matrix M has the identity property:

```
_Event NotIdentity {};                  // concurrent exception
_Task IdentityCheck {
    ...                                 // ADD HERE
    void main();                        // WRITE THIS ROUTINE
  public:
    _Event Stop {};                     // concurrent exception
    IdentityCheck(
            const int r,                // row number of matrix
            const int row[],            // row of matrix to check
            const int cols,             // number of columns in row
            uBaseTask & prgMain         // program main task
          );                            // WRITE THIS ROUTINE
};
```

If a row does not have the identity property, the global concurrent exception NotIdentity is raised at the pgmMain and the task returns (terminates). If the task receives the concurrent exception IdentityCheck::Stop, it stops checking and returns (terminates).

(e) **19 marks** Using task IdentityCheck and the declarations identity and M above, write a **complete** $\mu$C++ program using *task objects* to concurrently check if each row of M has the identity property:

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 checks if the matrix has the identity property for each row,
- if it receives a NotIdentity exception, raises the concurrent IdentityCheck::Stop exception at each nondeleted IdentityCheck task.
- and prints the overall result of the identity check.

**No documentation or error checking of any form is required.**

An example of the program input is:

```
4 5                 matrix dimensions
1 0 0 0 0           matrix values
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
```

(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 the program output is:

```
1 0 0 0 0           original matrix
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
identity
```

```
1 0 0 2 0           original matrix
0 1 0 0 0
7 0 1 0 0
0 0 0 3 0
not identity
```

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