UNIVERSITY OF WATERLOO

Midterm Examination
Winter 2023

Computer Science 343
Concurrent and Parallel Programming
Section 001

Duration of Exam: 2 hours
Number of Exam Pages (including cover sheet): 7
Total number of questions: 6
Total marks available: 97

CLOSED BOOK, NO ADDITIONAL MATERIAL ALLOWED

Instructor: Caroline Kierstead

March 8, 2023

1. (a) **5 marks** We have seen several different mechanisms used to modify control flow in a section of code. Complete the following table in your answer booklet, putting a checkmark (✓) if the mechanism can be used to perform the specified type of exit and an X (×) if it cannot.

Mechanism	**Exit type**		
	static multi exit	*static multi-level exit*	*dynamic multi-level exit*
unlabelled **break**			
labelled **break**			
goto			
longjmp			
exception			

(b) **2 marks** If we try to modularize the μC++ code snippet in the left-hand column of the following table to the code appearing in the right-hand column, the modified code fails to compile. Explain what problem causes the compilation failure.

Version 1

```
void foo( /* parameter list */ ) {
    B1: for (int i = 0; i < 10; i += 1) {
        // do something
        B2: for (int j = 0; j < 15; j += 1) {
            // do something
        if ( /* some condition */ ) break B1;
            // do something else
        } // B2
        // do more work
    } // B1
} // foo
```

Version 2

```
int rtn( /* parameter list */ ) {
    B2: for (int j = 0; j < 15; j += 1) {
        // do something
    if ( /* some condition */ ) break B1;
        // do something else
    } // B2
} // rtn

void foo( /* parameter list */ ) {
    B1: for (int i = 0; i < 10; i += 1) {
        // do something
        w = rtn( /* parameters */ );
        // do more work
    } // B1
} // foo
```

(c) When discussing traditional error handling approaches, we discussed *return unions*, of which the C++-17 variant type is an example.

i. **1 mark** Define the term *return union*.

ii. **1 mark** Describe one advantage of its use.

iii. **1 mark** Describe one disadvantage of its use.

2. In the following code, objects a and b are created and manipulated. The program main is ::main. Use these names when answering the parts (a) to (c) of the question.

```
 1  _Event Stop;
 2  _Coroutine B {
 3      A * partner;
 4      void main() {
 5          suspend();
 6          resume();
 7          partner->next();
 8          partner->next();
 9      }
10    public:
11      B() {
12          resume();
13      }
14      void setPartner( A & a ) {
15          partner = &a;
16      }
17      void next() {
18          resume();
19      }
20  };
```

```
21  _Coroutine A {
22      B & partner;
23      void A::main() {
24          partner.next();
25      }
26    public:
27      A( B & b ) : partner( b) {}
28      void next() {
29          resume();
30      }
31  };
32  int main() { // program main
33      B b;
34      A a( b );
35      b.setPartner( a );
36      b.next();
37  }
```

(a) **2 marks** Fill in the table in the answer booklet with the line of code that, when executed, causes the specified object to change into a coroutine, as well as the identity of its *cocaller*.

(b) **4 marks** Fill in the table in the answer booklet with the name of the coroutine that becomes inactive, and the one that becomes active, when the specified line of code is executed for the first time.

(c) **2 marks** Do all of the coroutines terminate? (Circle either "Yes" or "No" as appropriate in the answer booklet.) Of those that do terminate, what is the order of termination? (Write the names from left to right in the order in which they terminate.)

(d) We wish to modify the given code by having coroutine b raise a non-local exception Stop at coroutine a between lines 8 and 9

i. **1 mark** Write the code to raise the non-local exception.

ii. **2 marks** Describe the next step after raising the exception for A::main to receive the non-local exception.

iii. **2 marks** Describe the changes in A::main for the non-local exception to be propagated and handled.

iv. **3 marks** Explain in general what happens if a coroutine raises a non-local *resumption* exception and has no handlers at all anywhere in the program.

(e) Python has *stackless* coroutines which have restrictions.

i. **1 mark** What is the limitation on the placement of the Python yield statement? (yield is like μC++ suspend)

ii. **1 mark** What class of problems do the restrictions exclude/disallow?

3. (a) When introducing the topic of concurrency, we defined the terms *parallel execution* and *concurrent execution*.
   i. **1 mark** Which of these two forms of execution can occur when there is only 1 CPU/processor?
   ii. **2 marks** Explain why it can, and the other cannot.

   (b) **2 marks** For software solutions for mutual exclusion, explain what the terms *unbounded* and *bounded* overtaking mean in terms of declaring and retracting intent.

   (c) The *Hehner and Shyamasundar* version of the Bakery algorithm for N-tasks uses a ticket-taking mechanism.
   i. **1 mark** Is it possible for two or more competing tasks to end up with the same ticket value? (Circle either "Yes" or "No" in the answer booklet.)
   ii. **2 marks** If your answer is "Yes", explain how mutual exclusion is preserved; if your answer is "No", explain how unique values are guaranteed.

4. (a) **2 marks** As we saw in the bounded-buffer producer-consumer problem on assignment 3, in certain situations, a *busy-waiting* solution for mutual exclusion may have better performance than a non-busy-waiting approach. Explain what is happening in that situation that would have a negative impact on the user (real time) speed of the solution.

   (b) You are given the following lock implementation:

```
class MyLock {
    SpinLock lock;
    bool avail = true;
    uBaseTask * owner = nullptr;
  public:
    void acquire();
    void release();
};
void MyLock::acquire() {
    lock.acquire();
    if ( ! avail && owner != currThread() ) {
        // add self to lock' s blocked list
        yieldNoSchedule( lock );
        // DO NOT REACQUIRE LOCK
    } else {
        avail = false;
        lock.release();
    }
    owner = currThread();
}
```

```
void MyLock::release() {
    lock.acquire();
    owner = nullptr;
    if ( ! blocked.empty() ) {
        // remove task from blocked list
        // and make ready
    } else {
        avail = true;
    }
    lock.release();
}
```

   i. **1 mark** Does the lock implementation use *barging avoidance* or *barging prevention*?
   ii. **2 marks** Justify your choice.

5. **18 marks** Write a *semi-coroutine* filter with the following public interface (you may only add a public destructor and private members, which includes helper methods):

```
_Coroutine HexDump {
    enum { GROUP = 4, SUBGROUP = 2, PAIRS = 2 }; // defined constants
    const char s[17] = "0123456789abcdef";       // hexadecimal characters
    char ch;                                      // character passed by cocaller
    void main();
  public:
    _Event Eof {};
    void next( char c ) {
        ch = c;
        resume();
    };
}; // HexDump
```

that receives a sequence of characters. It replaces each character in the stream with its corresponding ASCII 2-hexadecimal digit value and outputs it to standard output (cout). For example, the character `'a'` is transformed into the two characters `'6'` and `'1'`, as 61 is the ASCII hexadecimal value for character `'a'`. The HexDump coroutine also formats its output by:

- transforming two characters into 4 hexadecimal digits followed by one space (␣),
- transforming the next two characters into 4 hexadecimal digits followed by three spaces (␣␣␣),
- repeating this sequence four times, and adding a newline at the end of the group.

For example, this input sequence:

```
The quick brown fox jumps over the lazy dog.\n
```

generates the following:

```
5468␣6520␣␣␣7175␣6963␣␣␣6b20␣6272␣␣␣6f77␣6e20
666f␣7820␣␣␣6a75␣6d70␣␣␣7320␣6f76␣␣␣6572␣2074
6865␣206c␣␣␣617a␣7920␣␣␣646f␣672e␣␣␣0a
```

Note, the spaces are separators not terminators, and hence, there are no spaces at the end of a line. As well, a newline must be produced if the coroutine terminates without having output a newline, but never outputs two newline characters in a row.

The exception Eof is raised at coroutine HexDump when there are no more characters, indicating the coroutine must terminate. **You must ensure that upon receiving Eof the coroutine main terminates.**

It is possible to convert a character to its hexadecimal value using a simple, short expression. No complex library routines are required. For example, an 8-bit byte (character) is composed of two 4-bit (hex) "nibbles", so

```
ch >> 4; // shift and return the (high-order) first nibble, a value in the range 0 to 15
ch & 0x0f; // mask and return the (low-order) second nibble, a value in the range 0 to 15
```

Write **ONLY** HexDump::main, do **NOT** write the program main that uses it! **No documentation or error-checking of any form is required.**

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.

   Write the following μC++ code fragments to *efficiently* check if a set of files contains the std::string key. Perform this check using the minimal number of (logical) threads required to achieve maximum concurrency.

   Assume the following code is the program main.

```
int main() {
    unsigned int numFiles;
    string key;
    cin >> key >> numFiles;
    string fileNames[numFiles];
    // read ordered list of file names and store in array fileNames
    bool found = false; // used for output below
    uBaseTask & prgMain = uThisTask();   // program main′ s task id for search

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

    cout << "key" << ( found ? " " : " not " ) << "found" << endl;
}
```

   (a) **2 marks** Complete the sequential function with the following interface that checks if the specified file contains the key.

```
_Event Found {};

void search( const string & filename, const string & key, uBaseTask & pgmMain ) {
    ifstream input{ filename.c_str() };
    string line;

    for ( ;; ) {
        getline( input, line );
      if ( input.fail() ) break;
        if ( line.find( key ) != string::npos ) {
            // Found the key, so notify pgmMain!
            // PLACE YOUR ANSWER HERE IN THE ANSWER BOOKLET.
        }
    } // for
```

   where filename is the name of the file to search, key is the string to search for in the file, and pgmMain is the program-main task, ::main. If the function determines the key exists, it raises the exception Found 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) **2 marks** Write a fragment of the program main using a COFOR statement that calls the search function to check the files concurrently.

(c) **6 marks** Write a message and actor type with the following interface that uses the search function to check the files concurrently.

```
struct WorkMsg : public uActor::Message {
    // YOU WRITE THIS TYPE
}; // WorkMsg

_Actor Search {
    Allocation receive( Message & msg ) {
        // YOU WRITE THIS MEMBER
    } // Search::receive
}; // Search
```

The program main in question 6(e)i) creates these messages/actors and passes each actor a message containing all the information needed to call function search.

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

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

The program main in question 6(e)ii) creates these tasks and passes via its constructor all the information needed to call function search.

As an optimization, if the program main receives the concurrent Found exception, it raises exception Search::Stop at any non-deleted Search tasks. When the concurrent Stop exception is propagated in a Search task, it stops performing its search, 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 Found 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 Found exception, and raise exception Search::Stop at any non-deleted Search tasks. **Note:** *all* tasks must be created, even if a Found exception is raised during creation.

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