UNIVERSITY OF WATERLOO

Midterm Examination
Fall 2020

Computer Science 343
Concurrent and Parallel Programming
Sections 041, 042

Duration of Exam: 3 hours including download and upload
Number of Exam Pages (including cover sheet): 9
Total number of questions: 3
Total marks available: 99

Click here to view your time remaining in the Midterm

Instructor: Peter Buhr

November 2, 2020

Instructions

The midterm duration is **3 hours** including download of the exam and uploading/submitting the programs. Your time begins at the exam download and is compared with the date-stamps on the files you submit.

The midterm consists of 3 *complete* running programs, which will be tested after the exam.

You are given three files with a compile command, sample test data, and the program main for each question:

```
phone.cc
merge.cc
schmilblick.cc
```

Answer each question in the appropriate file using the given program main. Do NOT subdivide a program into separate .h and .cc files.

After writing and testing the 3 programs, submit the three file on the undergraduate environment using the submit command:

```
$ submit cs343 midterm directory-name
```

As a precaution, submit often not just at the end of the midterm.

The following aids are allowed:

- computer to write, compile, and test the midterm programs.
- course notes
- course textbook
- your previous assignments
- μC++ Annotated Reference Manual
- man pages
- cppreference.com / cplusplus.com to look up C++ syntax

The following aids are NOT allowed:

- any answers from prior midterm exams because you did not create them
- any web searching, like stack overflow or Wikipedia
- any interaction with another person
- any use of another person's documents or programs

Basically, you are to complete the midterm by yourself using only course-related material or work that you have created versus the work others.

There is no way for us to fairly answer questions over the 24 hours of the exam, so state any assumptions with a comment in the program and press on.

Do not post on Piazza during the 24-hour exam period. In a desperate situation, do a private post.

Semi-coroutine

1. **37 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
    // YOU ADD MEMBERS HERE
   void main() {
       // YOU WRITE THIS MEMBER
   } // Phone::main
  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();
    } // Phone::next
}; // Phone
```

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

phoneno : $area_{opt}$ *trunk* "-" *number* "\n"
area : $country_{opt}$ "(" *3-digit-number* ")"
country : "+1"
trunk : *3-digit-number*
number : *4-digit-number*
digit : "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

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
+1(876)343-8760	+1189-6543
(876)343-8760	789 6543
(800)555-1212	(888)345-879
456-9807	-8790
786-5555	5555

After creation, the coroutine is resumed with a series of characters from a string (one character at a time). The coroutine raises one of the following exceptions at its resumer:

- Match means the characters form a valid string.
- Error means the last character forms an invalid string.

After the coroutine raises an exception at its last resumer, it must NOT be resumed again; sending more characters to the coroutine after this point is undefined and should generate an error. Assume the C library routine isdigit(c), which returns true if c is a digit and false otherwise

No documentation or error checking of any form is required in the program.

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.

The *given* program main in file phone.cc performs the following:

- reads a line from cin into a string,
- creates a Phone coroutine,
- passes characters from the string to the coroutine one at time, plus a newline `'\n'` character after all characters are passed and there is no match or error,
- prints an appropriate message to cout when the coroutine returns exception Match or Error,
- terminates the coroutine, and
- repeats these steps until end-of-file.

The executable program is named phone and has the following shell interface:

```
phone < infile-file     # '<' is shell indirection to cin
```

An input phone number has no leading or trailing (extra) characters, and there are no empty lines. For each input line, the input line is printed, as much of the line parsed, and the string yes if the string is valid and no otherwise. The following is example output:

```
'+1(876)343-8760' : '+1(876)343-8760' yes
'(876)343-8760' : '(876)343-8760' yes
'(800)555-1212' : '(800)555-1212' yes
'456-9807' : '456-9807' yes
'786-5555' : '786-5555' yes
'+1189-6543' : '+11' no
'789 6543' : '789 ' no
'(888)345-879' : '(888)345-879' no
'-8790' : '-' no
'5555' : '5555' no
```

The program is compiled with command:

```
$ u++ -g phone.cc -o phone
```

Full Coroutine

2. **23 marks** Write a *full coroutine* called Merge that works with another instance of itself to merge two sorted arrays of values in ascending order to produce a third sorted array. Duplicate values may appear in the merged array.

   The Merge coroutine has the following public interface (you may add only a public destructor and private members):

```
template<typename T> _Coroutine Merge {
    // YOU ADD MEMBERS HERE
    void main() {
        // YOU WRITE THIS MEMBER
    } // Merge::main
  public:
    _Event Sentinel {};         // end of merge
    // YOU WRITE THESE MEMBERS
    void start( Merge & partner, unsigned int size, const T values[ ], T mvalues[ ] );
    void mergeTo( T lowest, unsigned int nextPosn );
};
```

   The Sentinel exception indicates end of merging. The start member initializes a coroutine and has the following parameters: a reference to the partner coroutine, the size of the following sorted arrays, the sorted array to be merged, and the array into which the merged values are placed. The start member stores its parameter values but does *not* resume the coroutine. The mergeTo member resumes the coroutine.

   When a coroutine is activated via mergeTo, it copies from its sorted array into the merged array all values that are less than or equal to lowest. When a larger value is found, the current coroutine calls its partner's mergeTo, passing the value that caused the current instance to stop merging, i.e., the first value in its array that is greater than lowest) and the next empty location in array mvalues. The activated partner then does the same as the first, i.e., copy values from its sorted array into mvalues until a value is greater than lowest. The two coroutines pass control back and forth until one runs out of values and then it raises exception Sentinel at its partner and terminates. When the partner catches the Sentinel exception, it adds any of its remaining values into the merge array and terminates.

   No documentation or error checking of any form is required in the program.

   NOTE: there is an existing UNIX command called merge so during testing ensure you invoke your program and not the UNIX one.

The *given* program main in file merge.cc performs the following:

- reads the two sorted arrays from the given input files and prints them to the given output file,
- creates the merge array and two instances of Merge,
- calls the start member to initialize each coroutine.
- starts the merge by comparing the first values in each of the sorted arrays, and then calls the mergeTo member of the coroutine with the smaller value, passing it the larger of the first array values and 0, respectively.
- prints the merged values in the mvalues array.

The program main handles an arbitrary number of values for each array. The program is passed the preprocessor variable TYPE containing the type of the merged values. Assume TYPE has all the necessary operators to perform a merge.

The executable program is named merge and has the following shell interface:

```
merge sorted-infile1 sorted-infile2 merged-outfile
```

The program main assume all 3 files are specified, exist, and the input data is correctly formed. As well, there is at *least one data value* to be merged in each input file. An input file contains a list of sorted values, where each list starts with the number of values in that list.

For example, given the input file containing a list of 8 integer values:

```
8 -5 6 7 8 25 98 100 101
```

and the input file containing a list of 5 integer values:

```
5 -3 6 7 19 99
```

the program generates the following output:

```
input1 8: -5 6 7 8 25 98 100 101
input2 5: -3 6 7 19 99
merged 13: -5 -3 6 6 7 7 8 19 25 98 99 100 101
```

For example, given the input file containing a list of 8 floating-point values:

```
8 -5.5 6 7.7 8 25.99 98.16 100 101.99
```

and the input file containing a list of 5 floating-point values:

```
5 -3.4 6 7.7 19 99
```

the program generates the following output:

```
input1 8: -5.5 6 7.7 8 25.99 98.16 100 101.99
input2 5: -3.4 6 7.7 19 99
merged 13: -5.5 -3.4 6 6 7.7 7.7 8 19 25.99 98.16 99 100 101.99
```

The program is compiled with commands:

```
$ u++ -g merge.cc -DTYPE=int -o merge
$ u++ -g merge.cc -DTYPE=double -o merge
```

Task

3. **39 marks** 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 a *concurrent* program to *efficiently* check if any row of a matrix of size $N \times M$ contains at least two Schmilblicks. For example, in:

$$\begin{pmatrix} 1 & \mathbf{-1} & 3 & 4 & \mathbf{-1} \\ 2 & 1 & 4 & \mathbf{-1} & 6 \\ 3 & \mathbf{-1} & \mathbf{-1} & 6 & \mathbf{-1} \\ \mathbf{-1} & 6 & 7 & \mathbf{-1} & 1 \\ 4 & \mathbf{-1} & 6 & 1 & 8 \end{pmatrix}$$

   the Schmilblick value is -1, and rows 0, 2, 3 contain at least two Schmilblicks.

   Write a sequential function with the following interface to check the row of a matrix for a Schmilblick:

```
bool schmilblickCheck( const int row[ ], int cols, int schmilblick );
```

   where row is the matrix row, cols is the number of columns in the row, and schmilblick is the Schmilblick value. The function returns true if the row contains a Schmilblick and false otherwise.

   The matrix is checked concurrently along its rows using:

   (a) a COFOR statement, where each iteration of the COFOR uses function schmilblickCheck to check a particular row of the matrix for a Schmilblick.

   COFOR *logically* creates end – start threads, indexed start. .end–1 one per loop body.

```
// for ( int r = start; r < end; r += 1 ) { . . . }
COFOR( r, start, end,
    . . . // thread body, where r is the index for that thread
);
```

   (b) a series of actors and messages with the following interface:

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

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

   Each actor is started with a message containing the information needed to call function schmilblickCheck to check a particular row of the matrix for a Schmilblick. Create the actors on the stack and dynamically allocate the messages.

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

```
_Task Schmilblick {                       // check row of matrix
    // YOU ADD MEMBERS HERE
     void main() {
          yield( rand() % 100 );          // random task starts
          // YOU WRITE THIS MEMBER
    } // Schmilblicks::main
  public:
     _Event Stop {};                      // concurrent exceptions
     _Event SCHMILBLICK {};
     Schmilblick(                         // YOU WRITE THIS MEMBER
          int r,                          // row number
          const int row[ ],               // matrix row
          int cols,                       // columns in row
          uBaseTask & pgmMain,            // contact when Schmilblicks found
          int schmilblick                 // schmilblick value
     );
};
```

As an optimization, each Schmilblick task that finds a Schmilblick raises the concurrent exception SCHMILBLICK at the pgmMain and then returns. When the program main receives this concurrent exception, it raises exception Schmilblick::Stop at any non-deleted Schmilblick tasks. When the concurrent Stop exception is propagated in a Schmilblick task, it stops performing the Schmilblicks check, prints "told to stop" with row number, and returns.

In the program main, put the following call after the delete of each task.

```
// delete task
uEHM::poll();                             // poll for async exception
```

No documentation or error checking of any form is required in the program.

The *given* program main in file schmilblick.cc performs the following:

- reads from cin the Schmilblick value and matrix dimensions ($N \times M$),
- declares any necessary matrix, arrays and variables,
- reads from cin and prints to cout the matrix,
- creates an additional uProcessor and concurrently checks the matrix values in each row,
- and prints a message to standard output if two Schmilblicks are found in any matrix row.

The executable program is named schmilblick and has the following shell interface:

```
schmilblick < input-matrix     # '<' is shell indirection to cin
```

An example of an input file is:

```
-1 5 5          schmilblick value -1 and matrix dimensions 5 × 5
1 -1 3 4 -1     matrix values
2 1 4 -1 6
3 -1 1 6 -1
4 -1 6 1 8
-1 6 7 -1 1
```

(Phrases "*schmilblick value ...*" and "*matrix values*" do not appear in the input.)

An example output for CFOR and ACTOR is:

```
1, -1, 3, 4, -1, original matrix | 1, 2, 3, 4, 5,       original matrix
2, 1, 4, -1, 6,                  | 2, 1, 4, 5, 6,
3, -1, 1, 6, -1,                 | 3, 4, 1, 6, 7,
4, -1, 6, 1, 8,                  | 4, 5, 6, 1, 8,
-1, 6, 7, -1, 1,                 | 5, 6, 7, 8, 1,
                                 |
Schmilblicks found               | Schmilblicks not found
```

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

An example output for TASK is:

```
1, -1, 3, 4, -1,
2, 1, 4, -1, 6,
3, -1, 1, 6, -1,
4, -1, 6, 1, 8,
-1, 6, 7, -1, 1,

told to stop 2
told to stop 4
Schmilblicks found
```

The program is compiled with the following commands:

```
$ u++ -g -multi schmilblick.cc -DCFOR
$ u++ -g -multi schmilblick.cc -DACTOR
$ u++ -g -multi schmilblick.cc -DTASK
```