UNIVERSITY OF
WATERLOO

Midterm Examination
Term: Fall Year: 2021

Computer Science 343
Concurrent and Parallel Programming
Sections 001, 002

Duration of Exam: 3 hours including download and upload
Number of Exam Pages (including cover sheet): 10
Total number of questions: 3
Total marks available: 95

Click below to view your time remaining in the Midterm
https://student.cs.uwaterloo.ca/~cs343/cgi-bin/midtermTime.cgi

Instructor: Peter Buhr

November 3, 2021

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:

```
grammar.cc
figure8.cc
diagsymmetric.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 files 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 you have created versus the work of 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. If necessary, do a private post.

Semi-coroutine

1. **25 marks** Write a *semi-coroutine* with the following public interface (you may only add a public destructor and private members):

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

which verifies a string of characters is in a language. The language is described by the grammar:

L : *Xs*
Xs : '(' 'x' *X** ')'
X : 'x' | *ABs*
ABs : '[' 'a' 'b' *AB** ']'
AB : 'a' 'b' | *Xs*

The quotation marks are meta symbols and not part of the described language, '|' means alternative, and '*' means 0 or more. Notice, at least one 'x' or 'ab' appears at the start of an *Xs* or *ABs*.

The following are some valid and invalid strings.

valid	invalid
(x)	(**a**)
(xxx)	(xx**a**)
(x[abab])	(**[**])
(x[ab(x)])	(x[ab**[**x]])
(x[ab(x)(x)ab(x[ab]x)])	(x[ab(x)(x)ab(x**(**ab)x)])

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 terminate*.

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 program main in file grammar.cc must perform the following:

- reads a line from cin into a string,
- creates a Grammar coroutine,
- passes characters from the string to the coroutine one at time (no newline `'\n'` is passed),
- 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 grammar and has the following shell interface:

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

An input string starts in column 1 and empty lines are ignored. 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, followed by any unparsed characters. The following is example output for the valid/invalid strings above:

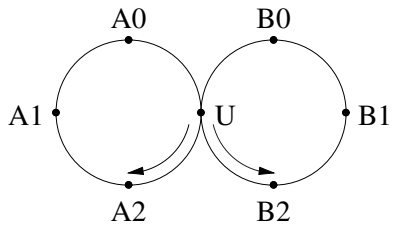
```
'(x)' : '(x)' : yes.
'(xxx)' : '(xxx)' : yes.
'(x[abab])' : '(x[abab])' : yes.
'(x[ab(x)])' : '(x[ab(x)])' : yes.
'(x[ab(x)(x)ab(x[ab]x)])' : '(x[ab(x)(x)ab(x[ab]x)])' : yes.
'(a)' : '(a' : no. Extra characters ')'
'(xxa)' : '(xxa' : no. Extra characters ')'
'([])' : '([' : no. Extra characters '])'
'(x[ab[x]])' : '(x[ab[' : no. Extra characters 'x]])'
'(x[ab(x)(x)ab(x(ab)x)])' : '(x[ab(x)(x)ab(x(' : no. Extra characters 'ab)x)])'
```

The program is compiled with command:

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

Full Coroutine

2. **28 marks** Write a *full coroutine* called Figure8 to simulate the figure-8 game with N A and B players in two circles, where the circles are connected by an umpire, e.g., $N = 3$.



The umpire selects a random number between 0 and $N-1$, called the *victim number*, and alternates tossing this number to the first A or B player in a cycle. The players in that cycle toss the victim number around the cycle. If a player's id matches the victim number, it raises a *remove exception* at the umpire containing a pointer to itself. When control gets back to the umpire (complete cycle), the umpire handles the remove exception by unlinking the player specified in the exception and *deleting it*. If no player's id matches the victim number, then no player is removed for that cycle. The game ends when all the A and B players are removed.

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

```
_Coroutine Player {
    // YOU ADD MEMBERS HERE
    void main() {
        // YOU WRITE THIS MEMBER
    } // Player::main
  protected:
    _Event Rm {                                    // remove exception
      public:
        Player * vict;
        Rm( Player * vict ) : vict( vict ) {}
    }; // Rm

    Umpire * umpire;
    unsigned int id;                               // player identity
    Player * cycle1;                               // A player cycle
    void unlink( Player *&p ) {                    // remove player from cycle
        ((Player *)&p->resumer())->cycle1 = p->cycle1;
    } // Player::unlink
  public:
    Player( Umpire * umpire, unsigned int id, Player * cycle1 ) :
            umpire( umpire ), id( id ), cycle1( cycle1 ) {}
    void toss( unsigned int v ) {
        // YOU WRITE THIS MEMBER
    } // Player::toss
}; // Player
```

The Rm exception is raised by a player at the umpire to indicate it matched the victim number. The unlink member is called by the umpire to unlink a player from its cycle. The cycle1 member is the link pointer to connect the players into a cycle (singly-linked list). The toss member is called by the umpire and players to move the victim number around the cycle.

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

```
_Coroutine Umpire : public Player {
    Player * cycle2;                                    // B player cycle
    unsigned int numPlayers;
    // YOU ADD MEMBERS HERE
    void main() {
        // YOU WRITE THIS MEMBER
    } // Player::main
  public:
    Umpire( unsigned int numPlayers ) :
            Player( nullptr, numPlayers, nullptr ), numPlayers( numPlayers ) {}
    void close( Player * cyc1, Player * cyc2 ) { cycle1 = cyc1; cycle2 = cyc2; }
}; // Umpire
```

The umpire inherits from Player so it can be connected into the two player cycles. The umpire uses the cycle1 member inherited from Player as the head pointer for the A cycle. The additional cycle2 member is the head pointer for the B cycle. (Note, the players in both cycles are linked only by the cycle1 member in Player.) The close member is called by the program main to set the two head pointers for the A and B cycles.

Note, when the umpire is handling a remove exception for the B players (using cycle2), it must do the following:

```
swap( cycle1, cycle2 );                                 // pretend to be cycle1
// unlink
swap( cycle2, cycle1 );                                 // reset cycle1
```

The umpire knows a cycle is empty when cycle1 or cycle2 points at the umpire (**this**).

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

The program main in file figure8.cc must perform the following:

- creates the umpire and appropriately sized A and B arrays,
- connects players through the cycle1 parameter in the Player constructor, where player 0 is linked to the umpire.
- calls umpire's close member to initialize the two head pointers, which close the two cycles,
- starts the game by calling the toss member of the umpire with any value (not used).

The executable program is named figure8 and has the following shell interface:

figure8 *[number-of-players]*

where the default number of players is 4 is no argument is specified. The following output is generated:

```
$ figure8 2
A -1 : -1 0
B -1 : -1 0
A -0 : -0
A empty
B -1 : 0
A empty
B -1 : 0
A empty
B -1 : 0
A empty
B -0 : -0
B empty
```

```
$ figure8 3
A -0 : 2 1 -0
B -1 : 2 -1 0
A -1 : 2 -1
B -2 : -2 0
A -0 : 2
B -0 : -0
B empty
A -0 : 2
B empty
A -2 : -2
A empty
B empty
```

where `A -1` means traverse cycle A with victim number `1`, followed by the walk around the cycle printing the player ids, with `-1` indicating player 1 signalled the umpire to be removed. Rows marked in **red** did not remove a player because the victim number is a repeat from a prior removal.

The program is compiled with command:

```
$ u++ -g figure8.cc -o figure8
```

Task

3. **42 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 a matrix of size $N \times N$ is a diagonally-symmetric matrix. (Notice, the matrix *must* be square and assume $N \leq 20$.) A diagonally-symmetric matrix has identical values along the diagonal and is equal to its transpose, i.e., $M = M^T$. That is, given $A = a_{i,j}$, then $a_{0,0} = a_{i,i}$ and $a_{i,j} = a_{j,i}$, for all indices i and j. The following are all diagonally-symmetric matrices.

$$\begin{pmatrix} 1 \end{pmatrix} \quad \begin{pmatrix} 3 & 2 \\ 2 & 3 \end{pmatrix} \quad \begin{pmatrix} 7 & 2 & 3 \\ 2 & 7 & 4 \\ 3 & 4 & 7 \end{pmatrix} \quad \begin{pmatrix} -1 & 2 & 3 & 4 & 5 \\ 2 & -1 & 4 & 5 & 6 \\ 3 & 4 & -1 & 6 & 7 \\ 4 & 5 & 6 & -1 & 8 \\ 5 & 6 & 7 & 8 & -1 \end{pmatrix}$$

   Write a sequential function with the following interface to check if a row is diagonally symmetric:

```
_Event NotDS {};  // not diagonally symmetric
void diagSymmetricCheck(                // YOU WRITE THIS FUNCTION
    const int (*M)[20],                 // matrix
    int row,                            // check row
    int cols,                           // row columns
    uBaseTask & pgmMain                 // contact if not diagonal symmetric
);
```

   where M is the matrix, row is the matrix row to test, cols is the number of columns in a row, and pgmMain is the address of the program-main task. If the function determines the tested row is *NOT* part of a diagonally symmetric matrix, it raises the exception NotDS 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 (see below).

   The matrix is checked concurrently along its rows using:

   (a) a COFOR statement, where each iteration of the COFOR uses function diagSymmetricCheck to test its matrix row for diagonal symmetry. **Warning**, uThisTask() inside the COFOR returns the task id of a thread created by COFOR not the program-main task.

   In the program main, put the following **_Enable** after the COFOR statement.

```
COFOR( ... );
_Enable {}
```

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

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

Each actor is started with a message containing the information needed to call function diagSymmetricCheck to test its matrix row for diagonal symmetry. Create the actors on the stack and dynamically allocate the messages.

In the program main, put the following **_Enable** after the uActor::stop() call.

```
... // start actor system
uActor::stop();
_Enable {}
```

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

```
_Task DiagSymmetric {                       // check row of matrix
  public:
    _Event Stop {};                         // concurrent exception
  private:
    // YOU ADD MEMBERS HERE
    void main() {
        // YOU WRITE THIS MEMBER
    } // DiagSymmetric::main
  public:
    DiagSymmetric(                          // YOU WRITE THIS MEMBER
        const int (*M)[20],                 // matrix
        int row,                            // check row
        int cols,                           // row columns
        uBaseTask & pgmMain,                // contact if not diagonal symmetric
    );
};
```

As an optimization, if the program main receives the concurrent NotDS exception, it raises exception DiagSymmetric::Stop at any non-deleted DiagSymmetric tasks. When the concurrent Stop exception is propagated in a DiagSymmetric task, it stops performing its row check, prints "stopped" with row number, and returns.

Note, the program main must create *all* tasks, even if a NotDS exception is raised during creation. As well, tasks *must* be deleted from $N-1$ to 0 because the task for row $N-1$ does the least amount of work. Finally, put **_Enable**s around the **delete** call.

```
_Enable {}
delete ...
_Enable {}
```

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

The program main in file diagsymmetric.cc must perform the following:

- read from cin the matrix dimension N,
- declare matrix and any necessary variables,
- read matrix from cin and print to cout,
- create additional uProcessor,
- concurrently check matrix values in each row,
- print a message to standard output if the matrix is or is not diagonally symmetry.

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

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

An example input file is:

```
4                   matrix dimension
7 2 3 4             matrix values
2 7 4 5
3 4 7 6
4 5 6 7
```

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

An example output for CFOR and ACTOR is:

```
7, 2, 3, 4,     original matrix          7, 2, 3, 2,     original matrix
2, 7, 4, 5,                              2, 7, 4, 5,
3, 4, 7, 6,                              3, 4, 6, 6,
4, 5, 6, 7,                              4, 5, 6, 7,
matrix is diagonally symmetric           matrix is not diagonally symmetric
```

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

An example output for TASK is:

```
7, 2, 3, 4,                              7, 2, 3, 2,
2, 7, 4, 5,                              2, 7, 4, 5,
3, 4, 7, 6,                              3, 4, 7, 6,
4, 5, 6, 7,                              4, 5, 6, 7,
matrix is diagonally symmetric           stopped 2
                                         stopped 1
                                         matrix is not diagonally symmetric
```

Note, for small matrices, it is difficult to get a stopped message. Try a 20×20 matrix.

The program is compiled with commands:

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