CS 343 Fall 2016 – Assignment 2
Instructors: Peter Buhr and Aaron Moss
Due Date: Sunday, October 9, 2016 at 22:00
Late Date: Thursday, October 13, 2016 at 22:00

October 3, 2016

This assignment has complex semi-coroutines, and introduces full-coroutines and concurrency in μC++. Use it to become familiar with these new facilities, and ensure you use these concepts in your assignment solution, i.e., writing a C-style solution for questions is unacceptable, and will receive little or no marks. (You may freely use the code from these example programs.)

1. Write a *semi-coroutine* to sort a set of values into ascending order using a binary-tree insertion method. This method constructs a binary tree of the data values, which can subsequently be traversed to retrieve the values in sorted order. Construct a binary tree without balancing it, so that the values 25, 6, 9, 5, 99, 100, 101, 7 produce the tree:

```
           25
     6           99
 5       9           100
      7                   101
```

   By traversing the tree in infix order — go left if possible, return value, go right if possible — the values are returned in sorted order. Instead of constructing the binary tree with each vertex having two pointers and a value, build the tree using a coroutine for each vertex. Hence, each coroutine in the tree contains two other coroutines and a value. (A coroutine must be self-contained, i.e., it cannot access any global variables in the program.)

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

```
template<typename T> _Coroutine Binsertsort {
    const T Sentinel;           // value denoting end of set
    T value;                    // communication: value being passed down/up the tree
    void main();                // YOU WRITE THIS ROUTINE
  public:
    Binsertsort( T Sentinel ) : Sentinel( Sentinel ) {}
    void sort( T value ) {      // value to be sorted
        Binsertsort::value = value;
        resume();
    }
    T retrieve() {              // retrieve sorted value
        resume();
        return value;
    }
};
```

   Assume type T has operators == and <, and public default and copy constructors.

   Each value for sorting is passed to the coroutine via member sort. When passed the first value, v, the coroutine stores it in a local variable, pivot. Each subsequent value is compared to pivot. If v < pivot, a Binsertsort coroutine called less is resumed with v; if v => pivot, a Binsertsort coroutine called greater resumed with v. Each of the two coroutines, less and greater, creates two more coroutines in turn. The result is a binary tree of identical coroutines. The coroutines less and greater must not be created by calls to **new**, i.e., no dynamic allocation is necessary in this coroutine.

   The end of the set of values is signaled by passing the value Sentinel; Sentinel is initialized when the sort coroutine is created via its constructor. (The Sentinel is not considered to be part of the set of values.) When a coroutine receives a value of Sentinel, it indicates end-of-unsorted-values, and the coroutine resumes its left branch (if it exists) with a value of Sentinel, prepares to receive the sorted values from the left branch and pass them back up the tree until it receives a value of Sentinel from that branch. The coroutine then passes up its pivot value. Next, the coroutine resumes its right branch (if it exists) with a value of Sentinel, prepares to receive the sorted values from the right branch and pass them back up the tree until it receives a value of Sentinel from that branch. Finally, the coroutine passes up a value of Sentinel to indicate end-of-sorted-values and the coroutine terminates. (Note, the coroutine does not print out the sorted values it simply passes them to its resumer.)

Remember to handle the special cases where a set of values is 0 or 1, e.g., Sentinel being passed as the first or second value to the coroutine. These cases must be handled by the coroutine versus special cases in the uMain::main program.

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

```
binsertsort unsorted-file [ sorted-file ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) The type of the input values and the sentinel value are provided as preprocessor variables.

- If the unsorted input file is not specified, print an appropriate usage message and terminate. The input file contains lists of unsorted values. Each list starts with the number of values in that list. For example, the input file:

  ```
  8 25 6 8 5 99 100 101 7
  3 1 3 5
  0
  10 9 8 7 6 5 4 3 2 1 0
  ```

  contains 4 lists with 8, 3, 0 and 10 values in each list. (The line breaks are for readability only; values can be separated by any white-space character and appear across any number of lines.)

  Assume the first number in the input file is always present and correctly specifies the number of following values. Assume all following values are correctly formed so no error checking is required on the input data, and none of the following values are the Sentinel value.

- If no output file name is specified, use standard output. Print the original input list followed by the sorted list, as in:

  ```
  25 6 8 5 99 100 101 7
  5 6 7 8 25 99 100 101

  1 3 5
  1 3 5

  blank line from list of length 0 (not actually printed)
  blank line from list of length 0 (not actually printed)

  9 8 7 6 5 4 3 2 1 0
  0 1 2 3 4 5 6 7 8 9
  ```

  for the previous input file. End each set of output with a blank line.

Print an appropriate error message and terminate the program if unable to open the given files.

Because Binsertsort is a template, show an example where it can sort any type (like a structure with multiple values forming the key) that provides operators == and < (or similar operations depending on your implementation). Include this example type in the same file as uMain::main.

WARNING: When writing coroutines, try to reduce or eliminate execution "state" variables and control-flow statements using them. A state variable contains information that is not part of the computation and exclusively used for control-flow purposes (like flag variables). Use of execution state variables in a coroutine usually indicates you are not using the ability of the coroutine to remember prior execution information. *Little or no marks will be given for solutions explicitly managing "state" variables.* See Section 3.1.3 in the Course Notes for details on this issue.

2. Write a *full coroutine* that simulates the game of Hot Potato. What makes the potato *hot* is that it explodes after its internal count-down timer reaches zero. The game consists of an umpire and *N* players. The umpire starts a *set* by tossing the *hot* potato to a random player. The potato is then randomly tossed among the players until the timer goes off and it explodes. A player never tosses the potato to themself. The player holding the potato when the timer goes off (potato explodes) is eliminated. The potato is then given back to the umpire (by the dead player). The umpire resets the timer in the potato, and begins the game again with the remaining players, unless only one player remains, which the umpire declares the winner.

   The potato is tossed among the players and it also contains the count-down timer that goes off after a random number of clock ticks. The interface for the Potato is (you may only add a public destructor and private members):

```
class Potato {
    // YOU ADD MEMBERS HERE
  public:
    _Event Explode {};
    Potato( unsigned int maxTicks = 10 );
    void reset( unsigned int maxTicks = 10 );
    void countdown();
};
```

   The constructor is optionally passed the *maximum* number of ticks until the timer goes off. The potato chooses a random value between 1 and the maximum for the number of ticks. Member reset is called by the umpire to re-initialize the timer for the next set. Member countdown is called by the players, and throws exception Explode, if the timer has reached zero. Rather than absolute time to implement the potato's timer, make each call to countdown be one tick of the clock (relative time).

   The interface for a Player is (you may only add a public destructor and private members):

```
_Coroutine Player {
    // YOU ADD MEMBERS HERE
    void main();
  public:
    typedef ... PlayerList; // container type of your choice
    Player( Umpire &umpire, unsigned int Id, const PlayerList &players );
    void toss( Potato &potato );
};
```

   The type PlayerList is a C++ standard-library container of your choice, containing all the players still in the game. The constructor is passed the umpire, an identification number assigned by the main program, and the container of players still in the game. If a player is *not* eliminated while holding the *hot* potato, then the player randomly chooses another player, excluding itself, from the list of players and tosses it the potato using its toss member. Should a player randomly select itself, it then selects the player to the right (player-Id position plus one) among the remaining plays, modulo the number of remaining players.

   The interface for the Umpire is (you may only add a public destructor and private members):

```
_Coroutine Umpire {
    // YOU ADD MEMBERS HERE
    void main();
  public:
    Umpire( Player::PlayerList &players );
    void set( unsigned int player );
};
```

   Its constructor is passed the container with the players still in the game. The umpire creates the potato. When a player determines it is eliminated, i.e., the timer went off while holding the potato, it calls set, passing its player identifier so the umpire knows the set is finished. The umpire deletes and removes the eliminated player from the list of players. Hence, player coroutines are created in the main program but deleted by the umpire. The umpire then resets the potato ~~to the new maximum number of players~~, and tosses the potato to a randomly selected player to start the next set; this toss counts with respect to the timer in the potato.

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

```
hotpotato  players  [ seed ]
```

players is the number of players in the game and must be between 2 and 20, inclusive. seed is a value for initializing the random number generator using routine srand. When given a specific seed value, the program must generate the same results for each invocation. If the seed is unspecified, use a random value like the process identifier (getpid) or current time (time), so each run of the program generates different output. Issue appropriate runtime error-messages for incorrect usage or if values are out of range. The driver (main program) creates all necessary data structures and then starts the umpire by calling set with one of the player identifiers, which implies that player has terminated, but since it is still on the list, the game starts with all players.

Make sure that a coroutine's public methods are used for passing information to the coroutine, but not for doing the coroutine's work, which must be done in the coroutine's main..

Generate the following kind of output showing a dynamic display of the game:

```
$ hotpotato 5 1007
5 players in the match
  POTATO goes off after 4 tosses
Set 1:   U -> 3 -> 4 -> 0 -> 4 is eliminated
  POTATO goes off after 10 tosses
Set 2:   U -> 1 -> 3 -> 2 -> 1 -> 0 -> 3 -> 0 -> 3 -> 2 -> 1 is eliminated
  POTATO goes off after 10 tosses
Set 3:   U -> 2 -> 0 -> 2 -> 3 -> 2 -> 3 -> 2 -> 3 -> 2 -> 3 is eliminated
  POTATO goes off after 8 tosses
Set 4:   U -> 0 -> 2 -> 0 -> 2 -> 0 -> 2 -> 0 -> 2 is eliminated
0 wins the Match!
```

To obtain repeatable results, all random numbers are generated using class PRNG. There are exactly three calls to the random-number generator. One in the potato::reset, one in Umpire::main and one in Player::main.

WARNING: When writing coroutines, try to reduce or eliminate execution "state" variables and control-flow statements using them. A state variable contains information that is not part of the computation and exclusively used for control-flow purposes (like flag variables). Use of execution state variables in a coroutine usually indicates you are not using the ability of the coroutine to remember prior execution information. *Little or no marks will be given for solutions explicitly managing "state" variables.* See Section 3.1.3 in the Course Notes for details on this issue.

3. Compile the program in Figure 1 using the u++ command, without and with compilation flag -multi and ***no optimization***, to generate a uniprocessor and multiprocessor executable. Run both versions of the program 10 times with command line argument 10000000 on a multi-core computer with at least 2 CPUs (cores).

   (a) Show the 10 results from each version of the program.

   (b) Must all 10 runs for each version produce the same result? Explain your answer.

   (c) In theory, what are the smallest and largest values that could be printed out by this program with an argument of 10000000? Explain your answers. (**Hint:** one of the obvious answers is wrong.)

   (d) Explain the difference in the size of the values between the uniprocessor and multiprocessor output.

Submission Guidelines

Please follow these guidelines very carefully. Review the Assignment Guidelines and C++ Coding Guidelines *before* starting each assignment. **Each text file, i.e., *.*txt file, must be ASCII text and not exceed 500 lines in length, where a line is a maximum of 120 characters.** Programs should be divided into separate compilation units, i.e., *.{h,cc,C,cpp} files, where applicable. Use the submit command to electronically copy the following files to the course account.

1. q1binsertsort.h, q1*.{h,cc,C,cpp} – code for question 1, p. 1. The file q1binsertsort.h contains the template code of the binary-tree insertion-sort. **Program documentation must be present in your submitted code. No user or system ~~or test~~ documentation is to be submitted for this question. Output for this question is checked via a marking program, so it must match exactly with the given program.**

```
#include <iostream>
using namespace std;

volatile int iterations = 10000000, shared = 0;    // ignore volatile, prevent dead-code removal

_Task increment {
    void main() {
        for ( int i = 1; i <= iterations; i += 1 ) {
            shared += 1;    // no -O2 to prevent atomic increment instruction
        }
    }
};
void uMain::main() {
    if ( argc == 2 ) iterations = atoi( argv[1] );
#ifdef __U_MULTI__
    uProcessor p;                          // create 2nd kernel thread
#endif // __U_MULTI__
    {
        increment t[2];
    } // wait for tasks to finish
    cout << "shared:" << shared << endl;
}
```

Figure 1: Interference

2. q1*.testtxt – test documentation for question 1, p. 1, which includes the input and output of your tests. **Poor documentation of how and/or what is tested can results in a loss of all marks allocated to testing.**

3. PRNG.h – random number generator (provided)

4. q2*.{h,cc,C,cpp} – code for question 2, p. 3. **Program documentation must be present in your submitted code. No user, system or test documentation is to be submitted for this question. Output for this question is checked via a marking program, so it must match exactly with the given program.**

5. q3*.txt – contains the information required by question 3.

6. Use the following Makefile to compile the programs for question 1, p. 1 and 2, p. 3:

```
TYPE:=int
SENTINEL:=-1

CXX = u++                                        # compiler
CXXFLAGS = -g -Wall -Wno-unused-label -MMD -DTYPE="${TYPE}" -DSENTINEL="${SENTINEL}"
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}} # makefile name

OBJECTS1 = # object files forming 1st executable with prefix "q1"
EXEC1 = binsertsort

OBJECTS2 = # object files forming 2st executable with prefix "q2"
EXEC2 = hotpotato

OBJECTS = ${OBJECTS1} ${OBJECTS2}                # all object files
DEPENDS = ${OBJECTS:.o=.d}                       # substitute ".o" with ".d"
EXECS = ${EXEC1} ${EXEC2}                        # all executables

.PHONY : all clean

all : ${EXECS}                                   # build all executables

#############################################################
```

```
-include ImplType

ifeq (${IMPLTYPE},${TYPE})                  # same implementation type as last time ?
${EXEC1} : ${OBJECTS1}
	${CXX} ${CXXFLAGS} $^ -o $@
else
ifeq (${TYPE},)                             # no implementation type specified ?
# set type to previous type
TYPE=${IMPLTYPE}
${EXEC1} : ${OBJECTS1}
	${CXX} ${CXXFLAGS} $^ -o $@
else                                        # implementation type has changed
.PHONY : ${EXEC1}
${EXEC1} :
	rm -f ImplType
	touch q1${EXEC1}.h
	sleep 1
	${MAKE} TYPE="${TYPE}" SENTINEL="${SENTINEL}"
endif
endif

ImplType :
	echo "IMPLTYPE=${TYPE}" > ImplType
	sleep 1

${EXEC2} : ${OBJECTS2}                      # link step 2nd executable
	${CXX} ${CXXFLAGS} $^ -o $@

#############################################################

${OBJECTS} : ${MAKEFILE_NAME}               # OPTIONAL : changes to this file => recompile

-include ${DEPENDS}                         # include *.d files containing program dependences

clean :                                     # remove files that can be regenerated
	rm -f *.d *.o ${EXECS} ImplType
```

This makefile is used as follows:

```
$ make binsertsort
$ binsertsort ...
$ make hotpotato
$ hotpotato ...
```

Put this Makefile in the directory with the programs, name the source files as specified above, and then type make binsertsort or make hotpotato in the directory to compile the programs. This Makefile must be submitted with the assignment to build the program, so it must be correct. Use the web tool Request Test Compilation to ensure you have submitted the appropriate files, your makefile is correct, and your code compiles in the testing environment.

Follow these guidelines. Your grade depends on it!