CS 343 Fall 2013 – Assignment 2
Instructors: Bernard Wong and Peter Buhr
Due Date: Monday, October 7, 2013 at 22:00
Late Date: Wednesday, October 9, 2013 at 22:00

September 22, 2013

This assignment 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 *full coroutine* that simulates the game of Hot Potato. The game consists of an umpire and a number of players. The umpire starts a *set* by tossing the *hot* potato to a player. What makes the potato *hot* is the timer inside it. The potato is then tossed among the players until the timer goes off. A player never tosses the potato to themself. The player holding the potato when the timer goes off is eliminated. The potato is then given back to the umpire, who 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 the item tossed around by the players and it also contains the timer that goes off after a random period of time. The interface for the Potato is (you may only add a public destructor and private members):

```
class Potato {
  public:
    Potato( unsigned int maxTicks = 10 );
    void reset( unsigned int maxTicks = 10 );
    bool 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 this maximum for the number of ticks. Member reset is called by the umpire to re-initialize the timer to reuse the potato. Member countdown is called by the players, and returns **true** if the timer has gone off and **false** otherwise. Rather than use absolute time to implement the potato's timer, make each call to countdown be one tick of the clock.

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

```
_Coroutine Player {
    void main();
  public:
    typedef ... PlayerList; // container type of your choice
    Player( Umpire &umpire, unsigned int Id, PlayerList &players );
    unsigned int getId();
    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. The member getId returns a player's identification number. If a player is not eliminated while holding the *hot* potato, then the player chooses another player, excluding itself, at random from the list of players and tosses it the potato using its toss member. Use the following approach for the random selection:

```
do {
    next = rand() % numPlayersLeft;
} while ( next == me );
```

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

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

The umpire creates the potato. Its constructor is passed the container with the players still in the game. 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 removes the eliminated player from the list of players, but does not delete the player because it might be used to play some other game. Hence, player coroutines are created and deleted in the main program. The umpire then resets the potato and tosses it 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. When present, assume each argument is an integer value, but it may not be in range; print an appropriate usage message and terminate the program if a value is missing or outside its range. The driver creates the umpire, the players players, with player identifiers from 0 to players–1, and the container holding all the players. It 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.

The output should show a dynamic display of the game, i.e., the set number, each player taking their turn (Id identifies a player), and who gets eliminated for each set and the winning player. Output must be both concise and informative, e.g.:

```
5 players in the match
  POTATO will go off after 9 tosses
Set 1:   U -> 4 -> 0 -> 3 -> 2 -> 0 -> 1 -> 4 -> 0 -> 2 is eliminated
  POTATO will go off after 8 tosses
Set 2:   U -> 4 -> 3 -> 0 -> 3 -> 0 -> 3 -> 0 -> 3 is eliminated
  POTATO will go off after 10 tosses
Set 3:   U -> 1 -> 4 -> 0 -> 4 -> 1 -> 4 -> 1 -> 0 -> 4 -> 0 is eliminated
  POTATO will go off after 10 tosses
Set 4:   U -> 1 -> 4 -> 1 -> 4 -> 1 -> 4 -> 1 -> 4 -> 1 -> 4 is eliminated
1 wins the Match!
```

WARNING: When writing coroutines, try to reduce or eliminate execution "state" variables and control-flow statements using them. Use of execution state variables in a coroutine usually indicates that you are not using the ability of the coroutine to remember execution location. *Little or no marks will be given for solutions explicitly managing "state" variables.* See Section 4.3.1 in *Understanding Control Flow: with Concurrent Programming using μC++* for details on this issue.

2. 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) (e.g., undergraduate machines linux006, linux008, linux012, and linux016 each have 6 CPUs).

   (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.

```
#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
        } // for
    }
};

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

(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 results between the uniprocessor and multiprocessor output.

3. (a) Merge sort is one of several sorting algorithms that takes optimal time (to within a constant factor) to sort N items. It also lends itself easily to concurrent execution by partitioning the data into ~~those greater than a pivot and those less than a pivot~~ two, and each half can be sorted independently and concurrently by another task.

Write an in-place concurrent merge sort with the following public interface (you may add only a public destructor and private members):

```
template<typename T> _Task Mergesort {
  public:
    Mergesort( T values[], unsigned int low, unsigned int high, unsigned int depth );
};
```

that sorts an array of non-unique values into ascending order. A naïve conversion of a sequential mergesort to a concurrent mergesort partitions the data values as normal, but instead of recursively invoking mergesort on each partition, a new mergesort task is created to handle each partition. (For this discussion, assume no other sorting algorithm is used for small partitions.) However, this approach creates a large number of tasks: approximately $2 \times N$, where N is the number of data values. The number of tasks can be reduced to approximately N by only creating a new mergesort task for one partition and recursively sorting the other partition in the current mergesort task.

In general, creating many more tasks than processors significantly reduces performance (try an example to see the effect) due to contention on accessing the processors versus any contention in the program itself. The only way to achieve good performance for a concurrent mergesort is to significantly reduce the number of mergesort tasks via an additional argument that limits the tree depth of the mergesort tasks. The depth argument is decremented on each recursive call and tasks are only created while this argument is greater than zero; otherwise sequential recursive-calls are use to sort each partition. Use a starting depth value of $\lfloor \log_2 \mathit{processors} \rfloor$.

Recursion can overflow a task's stack, since the default task size is only 32K or 64K bytes in μC++. To check for stack overflow, call verify() at the start of the recursive routine, which prints a warning message if the call is close to the task's stack-limit or terminates the program is the stack limit is exceeded. If a warning or an error is generated by verify, globally increase the stack size of all tasks by defining routine:

```
unsigned int uDefaultStackSize() {
    return 512 * 1000;          // set task stack-size to 512K
}
```

which is automatically called by μC++ at task creation to set the stack size.

To maximize efficiency, mergesort tasks must not be created by calls to **new**, i.e., no dynamic allocation is necessary for mergesort tasks. However, two dynamically sized arrays are required: one to hold the initial unsorted data and one for copying values during a merge. Both of these arrays can be large, so creating them in a task can overflows the task's stack. Hence, the driver dynamically allocates the storage for the unsorted data, and the top-level task of the mergesort allocates the copy array passing it by reference to its child tasks.

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

```
mergesort unsorted-file [ sorted-file | -processors ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) Add the following declaration to uMain::main after checking command-line arguments but before creating any tasks:

```
uProcessor p[ processors - 1 ] __attribute__(( unused )); // use "processors" kernel threads
```

to increase the number of kernel threads to access multiple processors. This declaration must be in the same scope as the declaration of the mergesort task.

The program has two modes depending on the second command-argument (i.e., file or processors):

i. For the first mode: input number of values, input values, sort using 1 processor, output sorted values. Input and output is specified as follows:

- 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
  61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33
  32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 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.) Since the number of data values can be (very) large, dynamically allocate the array to hold the values, otherwise the array can exceed the stack size of uMain::main.

  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.

- 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
-3 1 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

60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36
  35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11
  10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
  25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
  50 51 52 53 54 55 56 57 58 59 60
```

for the previous input file. End each set of output with a blank line, and start a newline with 2 spaces after printing 25 values from a set of value.

ii. For the second mode: input array size *S*, initialize array to values *S*..1 (descending order), sort using processors processors, and print no values (used for timing experiments). The input file contains only one value, which is an array size. Parameter processors is a positive number (> 0). The default value if unspecified is 1. This mode is used to time the performance of the mergesort over a fixed set of values in descending order using different numbers of processors.

Print an appropriate error message and terminate the program if unable to open the given files. Check command argument processors for correct form (integer) and range; print an appropriate usage message and terminate the program if a value is invalid.

(b) i. Compare the speedup of the mergesort algorithm with respect to performance by doing the following:

- Time the execution using the time command:

  ```
  % time mergesort size -1
  16.100u 0.470s 0:04.24 390.8%
  ```

  (Output from time differs depending on your shell, but all provide user, system and real time.) Compare the *user* (16.1u) and *real* (0:04.24) time among runs, which is the CPU time consumed solely by the execution of user code (versus system) and the total time from the start to the end of the program.
- Adjust the array size to get execution times in the range 5 to 20 seconds. (Timing results below 1 second are inaccurate.) Use the same array size for all experiments.
- After establishing an array size, run 7 experiments varying the value of processors from 1 2 4 8 16 32 64. Include all 7 timing results to validate your experiments.

ii. State the observed performance difference with respect to scaling when using different numbers of processors to achieve parallelism.

iii. Very briefly (2-4 sentences) speculate on the program behaviour.

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. q1*.{h,cc,C,cpp} – code for question question 1, p. 1. **Program documentation must be present in your submitted code. No user or system 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.**

2. q1*.testtxt – test documentation for question 1, which includes the input and output of your tests, documented by the use of script and after being formatted by scriptfix. **Poor documentation of how and/or what is tested can results in a loss of all marks allocated to testing.**
3. q2interference.txt – contains the information required by question question 2, p. 2.
4. q3*.{h,cc,C,cpp} – code for question question 3a, 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. q3mergesort.txt – contains the information required by question question 3b. **Poor documentation of how and/or what is tested can results in a loss of all marks allocated to testing.**
6. Use the following Makefile to compile the programs for question question 1, p. 1 and 3a, p. 3:

```
TYPE:=int
OPT:=

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

OBJECTS1 = # list of object files for question 1 prefixed with "q1"
EXEC1 = hotpotato

OBJECTS3 = # list of object files for question 3 prefixed with "q3"
EXEC3 = mergesort

OBJECTS = ${OBJECTS1} ${OBJECTS3}                # all object files
DEPENDS = ${OBJECTS:.o=.d}                       # substitute ".o" with ".d"
EXECS = ${EXEC1} ${EXEC3}                        # all executables

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

.PHONY : all clean

all : ${EXECS}                                   # build all executables

${EXEC1} : ${OBJECTS1}                           # link step 1st executable
	${CXX} ${CXXFLAGS} $^ -o $@

-include ImplType

ifeq (${IMPLTYPE},${TYPE})                       # same implementation type as last time ?
${EXEC3} : ${OBJECTS3}
	${CXX} ${CXXFLAGS} $^ -o $@
else
ifeq (${TYPE},)                                  # no implementation type specified ?
# set type to previous type
TYPE=${IMPLTYPE}
${EXEC3} : ${OBJECTS3}
	${CXX} ${CXXFLAGS} $^ -o $@
else                                             # implementation type has changed
.PHONY : ${EXEC3}
${EXEC3} :
	rm -f ImplType
	touch q3${EXEC3}.h
	sleep 2
	${MAKE} TYPE="${TYPE}"
endif
endif

ImplType :
	echo "IMPLTYPE=${TYPE}" > ImplType
```

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

${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 hotpotato
$ hotpotato ...
$
$ make quicksort TYPE=int
$ quicksort ...
$ make quicksort TYPE=double
$ quicksort ...
$ make quicksort TYPE=char
$ quicksort ...
$ make quicksort OPT="-O2" TYPE=int
$ quicksort ...
```

Put this Makefile in the directory with the programs, name the source files as specified above, and then type make cardgame or make mergesort 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. **If the makefile fails or does not produce correctly named executables, or if a program does not compile, you receive zero for all "Testing" marks.**

Follow these guidelines. Your grade depends on it!