CS 343 Fall 2015 – Assignment 3
Instructors: Peter Buhr and Ashif Harji
Due Date: Monday, October 26, 2015 at 22:00
Late Date: Wednesday, October 28, 2015 at 22:00

October 17, 2015

This assignment introduces locks in μC++ and examines synchronization and mutual exclusion. Use it to become familiar with these new facilities, and ensure you use these concepts in your assignment solution. (You may freely use the code from these example programs.) **(Tasks may *not* have public members except for constructors and/or destructors.)**

1. (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 two, and each half can be sorted independently and concurrently by another task.

   Write a 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.

   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 verify produces a warning or an error, globally increase the stack size for all tasks by adding the following routine to your code before the next test:

```
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.

Add the following declaration to uMain::main after checking command-line arguments but before creating any tasks:

```
uProcessor p[ (1 << depth) - 1 ] __attribute__(( unused )); // 2^depth-1 kernel threads
```

to increase the number of kernel threads to access multiple processors (there is always one existing processor). This declaration must be in the same scope as the declaration of the initial mergesort task for the timing mode.

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

```
mergesort -s unsorted-file [ sorted-file ]
mergesort -t size (>= 0) [ depth (>= 0) ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) The type of the input values is provided as a preprocessor variable.

The program has two modes depending on the command option -s or -t (i.e., sort or time):

i. For the sort 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 5 lists with 8, 3, 0, 10, and 61 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 (this line not actually printed)
     blank line from list of length 0 (this line 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 22 values from a set of values.

ii. For the time mode, dimension an integer array to size, initialize the array to values size..1 (descending order), sort using $2^{depth} - 1$ processors, and print no values (used for timing experiments). Parameter depth is a non-negative number ($>= 0$). The default value if unspecified is 0. 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 arguments size and depth 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:

```
$ /usr/bin/time -f "%Uu %Ss %E" mergesort -t 100000000 0
14.13u 0.59s 0:14.68
```

  (Output from time differs depending on the shell, but all provide user, system and real time.) Compare the *user* (14.13u) and *real* (0:14.68) 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 the real time 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 depth from 0 1 2 3 4 5 6. 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.

2. (a) Implement a generalized FIFO bounded-buffer for a producer/consumer problem with the following interface (you may add only a public destructor and private members):

```
template<typename T> class BoundedBuffer {
  public:
    BoundedBuffer( const unsigned int size = 10 );
    void insert( T elem );
    T remove();
};
```

which creates a bounded buffer of size size, and supports multiple producers and consumers. You may *only* use uCondLock and uOwnerLock to implement the necessary synchronization and mutual exclusion needed by the bounded buffer.

Implement the BoundedBuffer in the following ways:

i. Use busy waiting when waiting for buffer entries to become free or empty. In this approach, new tasks may barge into the buffer taking free or empty entries from tasks that have been signalled to access these entries. This implementation uses two condition-locks, one for each of the waiting producer and consumer tasks. The reason there is barging in this solution is that uCondLock::wait re-acquires its argument owner-lock before returning. Now once the owner-lock is released by a task exiting insert or remove, there is a race to acquire the lock by a new task calling insert/remove and by a signalled task. If the calling task wins the race, it barges ahead of any signalled task. So the state of the buffer at the time of the signal is not the same as the time the signalled task re-acquires the argument owner-lock, because the barging task changes the buffer. Hence, the signalled task may have to wait again, and there is no guarantee of eventual progress (long-term starvation).

ii. Use no busy waiting when waiting for buffer entries to become free or empty. In this approach, new (barging) tasks must be prevented from taking free or empty entries if tasks have been unblocked to access these entries. This implementation uses three condition-locks, one for each of the waiting producer, consumer and barging tasks (*and has no looping*). Hint, one way to prevent barging is to use a flag variable to indicate when signalling is occurring; entering tasks test the flag to know if they are barging and wait on the barging condition-lock. When signalling is finished, barging tasks are unblocked. (Other solutions to prevent barging are allowed but loops are not allowed.)

Before inserting or removing an item to/from the buffer, perform an assert that checks if the buffer is not full or not empty, respectively. Both buffer implementations are defined in a single .h file separated in the following way:

```
#ifdef BUSY                     // busy waiting implementation
// implementation
#endif // BUSY

#ifdef NOBUSY                   // no busy waiting implementation
// implementation
#endif // NOBUSY
```

Test the bounded buffer with a number of producers and consumers. The producer interface is:

```
_Task Producer {
    void main();
  public:
    Producer( BoundedBuffer<BTYPE> &buffer, const int Produce, const int Delay );
};
```

The producer generates Produce integers, from 1 to Produce inclusive, and inserts them into buffer. Before producing an item, a producer randomly yields between 0 and Delay-1 times. Yielding is accomplished by calling yield(times) to give up a task's CPU time-slice a number of times. The consumer interface is:

```
_Task Consumer {
    void main();
  public:
    Consumer( BoundedBuffer<BTYPE> &buffer, const int Delay, const BTYPE Sentinel,
              BTYPE &sum );
};
```

The consumer removes items from buffer, and terminates when it removes a Sentinel value from the buffer. A consumer sums all the values it removes from buffer (excluding the Sentinel value) and returns this value through the reference variable sum. Before removing an item, a consumer randomly yields between 0 and Delay-1 times.

uMain::main creates the bounded buffer, the producer and consumer tasks. Use a buffer-element type, BTYPE, of **int** and a sentinel value of -1 for testing. After all the producer tasks have terminated, uMain::main inserts an appropriate number of sentinel values (the default sentinel value is -1) into the buffer to terminate the consumers. The partial sums from each consumer are totalled to produce the sum of all values generated by the producers. Print this total in the following way:

```
total: ddddd
```

The sum must be the same regardless of the order or speed of execution of the producer and consumer tasks.

The shell interface for the boundedBuffer program is:

```
boundedBuffer [ Cons [ Prods [ Produce [ BufferSize [ Delays ] ] ] ] ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) Where the meaning of each parameter is:

Cons: positive number of consumers to create. The default value if unspecified is 5.

Prods: positive number of producers to create. The default value if unspecified is 3.

Produce: positive number of items generated by each producer. The default value if unspecified is 10.

BufferSize: positive number of elements in (size of) the bounder buffer. The default value if unspecified is 10.

Delays: positive number of times a producer/consumer yields *before* inserting/removing an item into/from the buffer. The default value if unspecified is Cons + Prods.

Use the monitor MPRNG to safely generate random values (monitors will be discussed shortly). Check all command arguments for correct form (integers) and range; print an appropriate usage message and terminate the program if a value is missing or invalid.

(b) i. Compare the busy and non-busy waiting versions of the program with respect to performance by doing the following:

- Time the execution using the time command:

```
% time ./a.out
3.21u 0.02s 0:03.32 100.0%
```

(Output from time differs depending on your shell, but all provide user, system and real time.) Compare the *user* time (3.21u) only, which is the CPU time consumed solely by the execution of user code (versus system and real time).

- Use the program command-line arguments 50 55 10000 30 10 and adjust the Produce amount (if necessary) to get execution times in the range 0.1 to 100 seconds. (Timing results below 0.1 seconds are inaccurate.) Use the same command-line values for all experiments.
- Run both the experiments again after recompiling the programs with compiler optimization turned on (i.e., compiler flag -O2). Include all 4 timing results to validate your experiments.

ii. State the observed performance difference between busy and nobusy waiting execution, without and with optimization.

iii. Speculate as to the reason for the performance difference between busy and nobusy waiting execution.

iv. Add the following declaration to uMain::main after checking command-line arguments but before creating any tasks:

```
#ifdef __U_MULTI__
    uProcessor p[3] __attribute__(( unused )); // create 3 kernel thread for a total of 4
#endif // __U_MULTI__
```

to increase the number of kernel threads to access multiple processors. This declaration must be in the same scope as the declaration of the producer and consumer tasks. Compile the program with the -multi flag and no optimization on a multi-core computer with at least 4 CPUs (cores), and run the same experiment as above. Include timing results to validate your experiment.

v. State the observed performance difference between uniprocessor and multiprocessor execution.

vi. Speculate as to the reason for the performance difference between uniprocessor and multiprocessor execution.

Submission Guidelines

Please follow these guidelines 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. q1mergesort.h, q1*.{h,cc,C,cpp} – code for question question question 1a, p. 1. **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.**

2. q1*.txt – contains the information required by question question 1b, p. 3. **Poor documentation of how and/or what is tested can results in a loss of all marks allocated to testing.**

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

4. q2boundedBuffer.h, q2*.{h,cc,C,cpp} – code for question question 2a, 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. q2*.txt – contains the information required by question 2b.

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

```
TYPE:=int
BTYPE:=int
SENTINEL:=-1
KIND:=NOBUSY
OPT:=-O2

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

OBJECTS1 = # object files forming 1st executable with prefix “q1”
EXEC1 = mergesort

OBJECTS2 = # object files forming 2st executable with prefix “q2”
EXEC2 = buffer

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} ${EXEC1} TYPE="${TYPE}"
endif
endif

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

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

-include ImplKind

ifeq (${IMPLKIND},${KIND})                  # same implementation type as last time ?
${EXEC2} : ${OBJECTS2}
	${CXX} ${CXXFLAGS} $^ -o $@
else
ifeq (${KIND},)                             # no implementation type specified ?
# set type to previous type
KIND=${IMPLKIND}
${EXEC2} : ${OBJECTS2}
	${CXX} ${CXXFLAGS} $^ -o $@
else                                        # implementation type has changed
.PHONY : ${EXEC2}
${EXEC2} :
	rm -f ImplKind
	touch q2${EXEC2}.h
	sleep 1
	${MAKE} ${EXEC2} KIND=${KIND}
endif
endif

ImplKind :
	echo "IMPLKIND=${KIND}" > ImplKind

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

${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 ImplKind
```

This makefile is used as follows:

```
$ make mergesort TYPE=int
$ mergesort -s unsorted-file
$ make mergesort TYPE=double
$ mergesort -s unsorted-file
$ make mergesort TYPE=char
$ mergesort -s unsorted-file sorted-file
$ make mergesort OPT="-O2" TYPE=int
$ mergesort -t 100000000 4
$ make buffer KIND=BUSY
$ buffer ...
$ make buffer KIND=NOBUSY OPT="-multi -O2"
$ buffer ...
```

Put this Makefile in the directory with the programs, name the source files as specified above, and then type make mergesort or make buffer 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!