

CS 343 Fall 2020 – Assignment 3

Instructor: Peter Buhr

Due Date: Monday, October 26, 2020 at 22:00

Late Date: Wednesday, October 28, 2020 at 22:00

September 30, 2020

This assignment examines synchronization and mutual exclusion, and introduces locks in $\mu\text{C++}$. 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. Given the C++ program in Figure 1, compare stack versus heap allocation in a concurrent program.
 - (a) Compare the versions of the program and different numbers of tasks with respect to performance by doing the following:
 - Run the program after compiling with preprocessor variables DARRAY, VECTOR1, VECTOR2 and STACK. Use compiler flags `-O2 -multi -nodebug`.
 - Time the executions using the time command:

```
$ /usr/bin/time -f "%Uu %Ss %Er %Mkb" ./a.out 2 10000000
3.21u 0.02s 0:03.32r 4228kb
```

(Output from time differs depending on the shell, so use the system time command.) Compare the *user* (3.21u) and *real* (0:3.32) 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.
 - Use the second command-line argument (as necessary) to adjust the real time into the range 1 to 100 seconds. (Timing results below 1 second are inaccurate.) Use the same command-line values for all experiments, if possible; otherwise, increase/decrease the arguments as necessary and scale the difference in the answer.
 - Run the 4 experiments with the number of tasks set to 1, 2, and 4.
 - Include all 12 timing results to validate your experiments.
 - (b) State the performance difference (larger/smaller/by how much) with respect to scaling the number of tasks for each version.
 - (c) Very briefly (2-4 sentences) speculate on the performance scaling among the versions.
2. (a) Quick sort is one of the best sorting algorithms in terms of execution speed on randomly arranged data. It also lends itself easily to concurrent execution by partitioning the data into those greater than a pivot and those less than a pivot so each partition can be sorted independently and concurrently by another task (divide-and-conquer/embarrassingly concurrent).

Write a concurrent quick-sort function with the following interface:

```
template<typename T>
void quicksort( T values[], unsigned int low, unsigned int high, unsigned int depth );
```

that sorts an array of non-unique values into ascending order. Note, except for include files, all code must appear within quicksort. Choose the pivot as follows:

```
pivot = array[low + ( high - low ) / 2];
```

Implement the concurrent quicksort using:

- i. COBEGIN/COEND statements
- ii. **_Actor** type
All information about the array must be passed to the actor in an initial message not via the actor's constructor.

```

#include <iostream>
#include <vector>
#include <memory> // unique_ptr
using namespace std;

int tasks = 1, times = 10000000; // default values

_Task Worker {
    enum { size = 100 };
    void main() {
        for ( int t = 0; t < times; t += 1 ) {
#ifdef DARRAY
            unique_ptr<volatile int []> arr( new volatile int[size] );
            for ( int i = 0; i < size; i += 1 ) arr[i] = i;
#elif defined( VECTOR1 )
            vector<int> arr( size );
            for ( int i = 0; i < size; i += 1 ) arr.at(i) = i;
#elif defined( VECTOR2 )
            vector<int> arr;
            for ( int i = 0; i < size; i += 1 ) arr.push_back(i);
#elif defined( STACK )
            volatile int arr[size] __attribute__(( unused )); // prevent unused warning
            for ( int i = 0; i < size; i += 1 ) arr[i] = i;
#else
            #error unknown data structure
#endif
        } // for
    } // Worker::main
}; // Worker

int main( int argc, char * argv[] ) {
    bool printExtras = ! getenv( "NO_PRINT_EXTRAS" ); // print extra output
    try { // process command-line arguments
        switch ( argc ) {
            case 3:
                times = stoi( argv[2] ); if ( times <= 0 ) throw 1;
            case 2:
                tasks = stoi( argv[1] ); if ( tasks <= 0 ) throw 1;
        } // switch
    } catch( ... ) {
        cout << "Usage: " << argv[0] << " [ tasks (> 0) [ times (> 0) ] ]" << endl;
        exit( 1 );
    } // try
    uProcessor p[tasks - 1]; // add CPUs (start with one)
    {
        Worker workers[tasks]; // add threads
    }
    if ( printExtras ) { malloc_stats(); }
} // main

```

Figure 1: Stack versus Dynamic Allocation

iii. `_Task` type

A naïve concurrent quicksort partitions the data values as normal, but instead of recursively invoking quicksort on each partition, a new quicksort 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 quicksort task for one partition and recursively sorting the other partition in the current quicksort task. Finally, to maximize efficiency, quicksort tasks must not be created by calls to `new`, i.e., no dynamic allocation is necessary for quicksort tasks.

In general, creating many more tasks than processors significantly reduces performance due to contention on accessing the processors versus any contention in the program itself. The only way to achieve good performance for a concurrent quicksort is to significantly reduce the number of quicksort tasks via an additional argument that limits the tree depth of the quicksort tasks (see details below). 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 used to sort each partition.

Recursion can overflow a task's stack, since the default task size is only 256K bytes in $\mu\text{C++}$. To check for stack overflow, call `uThisTask().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 if 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 program main, which is automatically called by $\mu\text{C++}$ at task creation to set the stack size.

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

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

```
quicksort [ unsorted-file | 'd' [ sorted-file | 'd' [ depth (>= 0) ] ] ]
quicksort -t size (>= 0) [ depth (>= 0) ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) If no unsorted file is specified, use standard input. If no sorted file is specified, use standard output. If no depth is specified, use 0. **The input-value type is provided by the preprocessor variable `TYPE` (see the Makefile).**

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

sort mode: read the number of input values, read the input values, sort using 1 processor (which is the default), and output the sorted values. Input and output is specified as follows:

- The unsorted input 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 the program 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.

- The sorted output is the original unsorted 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

```

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.

time mode: dimension an integer array to size, initialize the array to values size..1 (descending order), randomize the values using:

```

unsigned int times = sqrt( size );
for ( unsigned int counter = 0; counter < times; counter += 1 ) {
    swap( values[0], values[rand() % size] );
} // for

```

sort using $2^{\text{depth}} - 1$ processors, and print no values (used for timing experiments).

Parameter depth is a non-negative number (≥ 0). The default value if unspecified is 0.

Create the additional processors by placing the following declaration in the same block as the quicksort call:

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

Keep the number of processors small as the [undergraduate machines](#) have a limited number of cores and other students are using the machines.

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 quicksort algorithm with respect to performance by doing the following:

- Bracket the call to the sort in the program main with the following to measure the real time for the sort *within* the program.

```

uTime start = uClock::currTime();
quicksort( ... );
cout << "Sort time " << uClock::currTime() - start << " sec." << endl;

```

- Compile with makefile option `OPT="-O2 -multi -nodebug"`.

- Time the execution using the time command:

```

$ /usr/bin/time -f "%Uu %Ss %E %Mkb" quicksort -t 300000000 0
14.13u 0.59s 0:14.68 1958468kb

```

(Output from time differs depending on the shell, so use the system time command.) 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 10 to 20 seconds for depth 0. Use the same array size for all experiments.
- For each of CBEGIN, ACTOR, and TASK, run 6 experiments varying the value of depth from 0 1 2 3 4 5. Include all 18 timing results to validate your experiments.

ii. State the performance difference (larger/smaller/by how much) with respect to scaling when using different numbers of processors to achieve parallelism.

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

3. (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 on a full or empty buffer. In this approach, tasks that have been signalled to access empty or full entries may find them taken by new tasks that barged into the buffer. This implementation uses one owner and two condition locks, where the waiting producer and consumer tasks block on the separate condition locks. (If necessary, you may add more locks.) 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 (looping), and there is no guarantee of eventual progress (long-term starvation).
- ii. Use *no* busy waiting when waiting for buffer entries to become empty or full. In this approach, use *barging avoidance* so a barging task cannot take empty or full buffer entries if another task has been unblocked to access these entries. This implementation uses one owner and two condition locks, where the waiting producer and consumer tasks block on the separate condition locks, but there is (*no looping*). (If necessary, you may add more locks.) Hint, one way to prevent overtaking by bargers 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 an appropriate condition-lock. When signalling is finished, an appropriate task is unblocked.
- iii. Briefly explain why it is impossible to solve this problem using *barging prevention*.

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
```

The kind of buffer is specified externally by a preprocessor variable of `BUSY` or `NOBUSY`.

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

```
_Task Producer {
    void main();
public:
    Producer( BoundedBuffer<int> & 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` 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<int> & buffer, const int Delay, const int Sentinel, int &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` times.

The program main creates the bounded buffer, the producer and consumer tasks, and an array of subtotal counters, one for each consumer. After all the producer tasks have terminated, the program 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: dddd...

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:

```
buffer [ Cons [ Prods [ Produce [ BufferSize [ Delays [ Processors ] ] ] ] ] ]
```

(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 bounded 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.

Processors: positive number of kernel threads to create for parallelism. The default value if unspecified is 1. Use this number in the following declaration placed in the program main immediately after checking command-line arguments but before creating any tasks:

```
uProcessor p[Processors - 1] __attribute__((unused)); // create more kernel thread
```

The program starts with one kernel thread so only N - 1 additional kernel threads are added.

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. The type of the buffer element is **int** throughout the program, and the integer sentinel value is specified externally by preprocessor variable **SENTINEL**.

- (b)
 - i. Compare the busy and non-busy waiting versions of the program with respect to *uniprocessor* performance by using 1 kernel thread:
 - Use the `μC++ -nodebug` flag in all the experiments.
 - Time the executions using the time command:


```
$ /usr/bin/time -f "%Uu %Ss %Er %Mkb" ./a.out
3.21u 0.02s 0:03.32r 33640kb
```

 (Output from time differs depending on the shell, so use the system time command.) 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 55 50 10000 20 10 1] and adjust the Produce amount (if necessary) to get program execution into the range 1 to 100 seconds. (Timing results below 1 second are inaccurate.) Use the same command-line values for all experiments, if possible; otherwise, increase/decrease the arguments as necessary and scale the difference in the answer.
 - Run both the experiments again after recompiling the programs with compiler optimization turned on (i.e., compiler flag `-O2`).
 - Include 4 timing results to validate the experiments.
 - ii. State the performance difference (larger/smaller/by how much) between uniprocessor busy and nobusy waiting execution, without and with optimization.
 - iii. Compare the busy and non-busy waiting versions of the program with respect to *multiprocessor* performance by repeating the above experiment with 4 kernel threads.
 - Include 4 timing results to validate the experiments.

- iv. State the performance difference (larger/smaller/by how much) between multiprocessor busy and nobusy waiting execution, without and with optimization.
- v. Speculate as to the reason for the performance difference between busy and non-busy execution.
- vi. Speculate as to the reason for the performance difference between uniprocessor and multiprocessor execution.

Submission Guidelines

Follow these guidelines carefully. Review the [Assignment Guidelines](#) and [C++ Coding Guidelines](#) *before* starting each assignment. **Each text or test-document file, e.g., *.txt, doc file, must be ASCII text and not exceed 500 lines in length, using the command `fold -w120 *.doc | wc -l`.** 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. q1new.txt – contains the information required by question 1, p. 1.
2. q2quicksort.h, q2*.h, cc, C, cpp – code for question question question 2a, 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.**
3. q2*.txt – contains the information required by questions 2b, p. 4.
4. MPRNG.h – random number generator (provided)
5. q3buffer.h, q3*.h, cc, C, cpp – code for question question 3a, p. 4. **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.**
6. q3*.txt – contains the information required by questions 3(a)iii, p. 5 and 3b.
7. Modify the following Makefile to compile the programs for question 2a, p. 1 and 3a, p. 4 by inserting the object-file names matching your source-file names.

```
OPT:= # -O2 -nodebug
SIMPL:=CBEGIN
TYPE:=int
BIMPL:=NOBUSY
SENTINEL:=-1

CXX = u++                                # compiler
CXXFLAGS = -g -multi ${OPT} -Wall -Wextra -MMD -D"${SIMPL}" -D"${BIMPL}" \
          -DTYPE="${TYPE}" -DSENTINEL="${SENTINEL}" # compiler flags
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}} # makefile name

OBJECTS1 = # object files forming 1st executable with prefix "q2"
EXEC1 = quicksort                        # 1st executable name

OBJECTS2 = # object files forming 2nd executable with prefix "q3"
EXEC2 = buffer                          # 2nd executable name

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 QuickImpl
```



```

ifeq (${shell if [ "${IMPLTYPE}" = "${TYPE}" -a "${IMPLSIMPL}" = "${SIMPL}" ] ; \
    then echo true ; fi },true)
${EXEC1} : ${OBJECTS1}
    ${CXX} ${CXXFLAGS} $^ -o $@
else
.PHONY : ${EXEC1}
${EXEC1} :
    rm -f QuickImpl
    touch q2quicksort.h
    ${MAKE} ${EXEC1} SIMPL="${SIMPL}" TYPE="${TYPE}"
endif

QuickImpl :
    echo "IMPLSIMPL=${SIMPL}\nIMPLTYPE=${TYPE}" > QuickImpl
    sleep 1

-include Buflmpl

ifeq (${BUFIMPL},${BIMPL})                                # same implementation type as last time ?
${EXEC2} : ${OBJECTS2}
    ${CXX} ${CXXFLAGS} $^ -o $@
else
.PHONY : ${EXEC2}
${EXEC2} :
    rm -f Buflmpl
    touch q3buffer.h
    sleep 1
    ${MAKE} ${EXEC2} BIMPL="${BIMPL}"
endif

Buflmpl :
    echo "BUFIMPL=${BIMPL}" > Buflmpl
    sleep 1

#####

${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} QuickImpl Buflmpl

This makefile is used as follows:

$ make quicksort SIMPL=CBEGIN
$ quicksort ...
$ make quicksort SIMPL=ACTOR TYPE=int
$ quicksort ...
$ make quicksort SIMPL=TASK TYPE=double
$ quicksort ...
$ make buffer BIMPL=BUSY                                     # use SENTINEL:=-1
$ buffer ...
$ make buffer BIMPL=NOBUSY SENTINEL=0 OPT="-O2" # switch to SENTINEL=0
$ buffer ...

```

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