

# CS 343 Winter 2019 – Assignment 3

Instructor: Peter Buhr

**Due Date: Monday, February 25, 2019 at 22:00**

**Late Date: Wednesday, February 27, 2019 at 22:00**

January 7, 2019

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 %E" ./a.out 2 10000000
3.21u 0.02s 0:03.32
```

(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. Multiplying two matrices is a common operations in many numerical algorithms. Matrix multiply lends itself easily to concurrent execution because data can be partitioned, and each partition can be processed concurrently without interfering with tasks working on other partitions (divide and conquer).

(a) Write a concurrent matrix-multiply with the following interface:

```
void matrixmultiply( int *Z[], int *X[], unsigned int xr, unsigned int xc, int *Y[], unsigned int yc );
```

which calculates  $Z_{xr,yc} = X_{xr,xc} \cdot Y_{xc,yc}$ , where matrix multiply is defined as:

$$X_{i,j} \cdot Y_{j,k} = \left( \sum_{c=1}^j X_{row,c} Y_{c,column} \right)_{i,k}$$

Implement the concurrent matrix-multiply by creating a task to calculate each row of the Z matrix from the appropriate X row and Y columns. Create concurrency using:

```

#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[] ) {
    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
} // main

```

Figure 1: Stack versus Dynamic Allocation

- i. implicit COFOR statement.
- ii. implicit **\_Actor** type.  
All information for the actor to compute its row of *Z* must be passed to the actor in an initial message not via the actor's constructor.
- iii. explicit **\_Task** type.  
To reduce the affect of Amdahl's law for the **\_Task** type version, do not start the tasks sequentially. Instead, start the tasks exponentially by having the first task create at most two more tasks, and each of these tasks create at most two tasks, etc. Hence, there is a binary tree of tasks, one for each row of the *Z* matrix. Make sure to achieve maximum concurrency, i.e., do not prevent the creating task from summing its row while subtasks execute.

The implementations are selected by the existence of the preprocessor variable CFOR (note the missing first "O"), ACTOR or TASK. No dynamic allocation is allowed in the matrixmultiply routine.

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

```
matrixmultiply xr xc yc [ processors | X-matrix-file Y-matrix-file ]
```

The first three parameters are the dimensions of the  $X_{xr,xc}$  and  $Y_{xc,yc}$  matrices. Print an appropriate usage message and terminate the program if there are missing/invalid number arguments (e.g., the dimension/processor values are less than one) or unable to open the given input files.

- If the  $X$  and  $Y$  input files are specified, each file contains a matrix with appropriate values based on the dimension parameters; e.g., the input file:

```
1  2  3  4
5  6  7  8
9 10 11 12
```

is a  $3 \times 4$  matrix. Assume the correct number of input values in each matrix file and all matrix values are correctly formed. After reading in the two matrices, multiply them, and print the product on standard output using this format:

```
$ matrixmultiply 3 4 3 xfile yfile
|          1          2          3
|          4          5          6
|          7          8          9
|         10         11         12
-----*-----
1          2          3          4 |        70        80        90
5          6          7          8 |       158       184       210
9         10         11         12 |       246       288       330
```

Where the matrix on the bottom-left is  $X$ , the matrix on the top-right is  $Y$ , and the matrix on the bottom-right is  $Z$ .

- If no matrix files are specified, create the appropriate  $X$  and  $Y$  matrices with each value initialized to 37, multiply them, **but print no output**. This case is used for timing the cost of parallel execution.

The matrices in the program may be large so allocate them on the heap.

Add the following declaration to the program main immediately after checking command-line arguments but before creating any concurrency or starting the actor system:

```
uProcessor p[processors - 1]; // number of kernel threads
```

to adjust the amount of parallelism for computation. The default value for processors is 1. Since the program starts with one kernel thread, only processors - 1 additional kernel threads are needed.

- (b)
  - i. Test for any benefits of concurrency by running the program in parallel:
    - Run the program on a multi-core computer with at least 16 actual CPUs (cores), with processors in the range [1, 2, 4, 8, 16] and dimensions 500 600 10000. Compile the program with the  $\mu$ C++ -multi flag, no optimization, and all 3 preprocessor variables.
    - Time the executions using the time command:
 

```
$ /usr/bin/time -f "%Uu %Ss %Er %Mkb" matrixmultiply 500 600 10000
3.21u 0.02s 0:05.67r 64244kb
```

Output from time differs depending on the shell, so use the system time command. Compare the *real* time (0:05.67r) only, which is the time to complete the computation.
    - If necessary, change the command-line parameters xcols-yrows and ycols to adjust 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.
    - Include 15 timing results to validate the experiments, 5 for each preprocessor variable.
  - ii. State the performance difference (larger/smaller/by how much) as kernel threads increase.
  - iii. State the performance difference (larger/smaller/by how much) among the 3 concurrency implementations.
- (c)
  - i. Repeat the experiment in 2b adding the -O2 compilation flag.
    - Include 15 timing results to validate the experiments, 5 for each preprocessor variable.
  - ii. State the performance difference (larger/smaller/by how much) for optimization.

- iii. State the performance difference (larger/smaller/by how much) among the 3 concurrency implementations.
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:

- Use busy waiting when waiting for buffer entries to become empty or full. In this approach, new tasks may barge into the buffer taking empty or full entries from tasks that have been signalled 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. (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).
- 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.
- 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:

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

#ifndef 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-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<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-1 times.

The program main creates the bounded buffer, the producer and consumer tasks. Use a buffer-element type of `int` and a sentinel value of -1 for testing. 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 ] ] ] ] ]
```

(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.

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 elements and the sentinel value are specified externally by preprocessor variables `TYPE` and `SENTINEL`, respectively.

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

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

to increase the number of kernel threads to access multiple processors. The program starts with one kernel thread so only 4 - 1 additional kernel threads are necessary.

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

- Time the executions using the time command:

```
$ /usr/bin/time -f "%Uu %Ss %E" ./a.out
3.21u 0.02s 0:03.32
```

(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 50 55 10000 30 10 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 the `-multi` flag.
    - 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

Please follow these guidelines carefully. Review the [Assignment Guidelines](#) and [C++ Coding Guidelines](#) *before* starting each assignment. **Each text or document file, e.g., \*.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. q1new.txt – contains the information required by question [1](#), [p. 1](#).
2. q2matrixmultiply.h, q2\*.{h,cc,C,cpp} – code for 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. 3](#) and [2c](#), [p. 3](#).
4. MPRNG.h – random number generator (provided)
5. q3buffer.h, q3\*.{h,cc,C,cpp} – code for 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. 4](#) 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:= -multi # -O2
MIMPL:=COFOR_
BIMPL:=NOBUSY
TYPE:=int
SENTINEL:=-1

CXX = u++                                # compiler
CXXFLAGS = -g ${OPT} -Wall -Wextra -MMD -std=c++14 -D"${MIMPL}" -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 = matrixmultiply                                # 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 MatrixImpl

ifeq (${MATRIXIMPL},${MIMPL})                          # same implementation type as last time ?
${EXEC1} : ${OBJECTS1}
    ${CXX} ${CXXFLAGS} $^ -o $@
else
ifeq (${MIMPL},)                                        # no implementation type specified ?
# set type to previous type
MIMPL=${MATRIXIMPL}
${EXEC1} : ${OBJECTS1}
    ${CXX} ${CXXFLAGS} $^ -o $@
else                                                    # implementation type has changed
.PHONY : ${EXEC1}
${EXEC1} :
    rm -f MatrixImpl
    touch q2matrixmultiply.h
    ${MAKE} ${EXEC1} MIMPL=${MIMPL}
endif
endif

MatrixImpl :
    echo "MATRIXIMPL=${MIMPL}" > MatrixImpl
    sleep 1

-include BuImpl

ifeq (${BUFIMPL},${BIMPL})                              # same implementation type as last time ?
${EXEC2} : ${OBJECTS2}
    ${CXX} ${CXXFLAGS} $^ -o $@
else
ifeq (${BIMPL},)                                        # no implementation type specified ?
# set type to previous type
BIMPL=${BUFIMPL}
${EXEC2} : ${OBJECTS2}
    ${CXX} ${CXXFLAGS} $^ -o $@
else                                                    # implementation type has changed
.PHONY : ${EXEC2}
${EXEC2} :
    rm -f BuImpl
    touch q3buffer.h
    sleep 1
    ${MAKE} ${EXEC2} BIMPL="${BIMPL}"
endif
endif

BuImpl :
    echo "BUFIMPL=${BIMPL}" > BuImpl
    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} MatrixImpl BuflImpl
```

This makefile is used as follows:

```
$ make matrixmultiply MIMPL=IMPLICIT
$ matrixmultiply ...
$ 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 nostaticexits`, `make matrixmultiply` 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!**