CS 343 Fall 2016 – Assignment 3
Instructors: Peter Buhr and Aaron Moss
Due Date: Monday, October 24, 2016 at 22:00
Late Date: Wednesday, October 26, 2016 at 22:00

October 15, 2016

This assignment examines synchronization and mutual exclusion, and introduces locks in μC++. 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) Transform routine do_work in Figure 1 so it preserves the same control flow but removes the **for** and **goto** statements, and replaces them with **ONLY** expressions (including & and |), **if/else** and **while** statements. The statements **switch**, **for**, **do**, **break**, **continue**, **goto**, **throw** or **return**, and the operators &&, || or ? are not allowed. In addition, setting a loop index to its maximum value, to force the loop to stop is not allowed. Finally, copying significant amounts of code or creating subroutines is not allowed, i.e., no transformation where the code grows exponentially with the number of nested loops. **Zero marks will be given to a transformation violating any of these restrictions.** New variables may be created to accomplish the transformation. Output from the transformed program must be identical to the original program.

   (b) Compare the original and transformed program with respect to performance by doing the following:

   - Remove (comment out) *all* the print (cout) statements in the original and transformed version.
   - Time the execution 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 argument (if necessary) to adjust the number of times the experiment is performed to get user times approximately in the range 0.1 to 100 seconds. (Timing results below 0.1 seconds are inaccurate.) Use the same command-line value for all experiments.
   - 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.
   - Explain the relative differences in the timing results with respect to the original and transformed program.
   - State the performance difference when compiler optimization is used?

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

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

```
#include <cstdlib>                      // atoi
#include <iostream>
using namespace std;

// volatile prevents dead-code removal
void do_work( int C1, int C2, int C3, int L1, int L2, volatile int L3 ) {
    for ( int i = 0; i < L1; i += 1 ) {
        cout << "S1 i:" << i << endl;
        for ( int j = 0; j < L2; j += 1 ) {
            cout << "S2 i:" << i << " j:" << j << endl;
            for ( int k = 0; k < L3; k += 1 ) {
                cout << "S3 i:" << i << " j:" << j << " k:" << k << " : ";
      if ( C1 ) goto EXIT1;
                cout << "S4 i:" << i << " j:" << j << " k:" << k << " : ";
          if ( C2 ) goto EXIT2;
                cout << "S5 i:" << i << " j:" << j << " k:" << k << " : ";
            if ( C3 ) goto EXIT3;
                cout << "S6 i:" << i << " j:" << j << " k:" << k << " : ";
            } // for
          EXIT3:;
            cout << "S7 i:" << i << " j:" << j << endl;
        } // for
      EXIT2:;
        cout << "S8 i:" << i << endl;
    } // for
  EXIT1:;
} // do_work

int main( int argc, char *argv[] ) {
    int times = 1, L1 = 10, L2 = 10, L3 = 10;
    switch ( argc ) {
      case 5:
        L3 = atoi( argv[4] );
        L2 = atoi( argv[3] );
        L1 = atoi( argv[2] );
        times = atoi( argv[1] );
        break;
      default:
        cerr << "Usage: " << argv[0] << " times L1 L2 L3" << endl;
        exit( EXIT_FAILURE );
    } // switch

    for ( int i = 0; i < times; i += 1 ) {
        for ( int C1 = 0; C1 < 2; C1 += 1 ) {
            for ( int C2 = 0; C2 < 2; C2 += 1 ) {
                for ( int C3 = 0; C3 < 2; C3 += 1 ) {
                    do_work( C1, C2, C3, L1, L2, L3 );
                    cout << endl;
                } // for
            } // for
        } // for
    } // for
} // main
```

Figure 1: Static Multi-level Exit

Create a task to calculate each row of the Z matrix from the appropriate X row and Y columns. To reduce the affect of Amdahl's law, do not start the tasks sequential. 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. Tasks must be created on the stack rather than calls to **new**, i.e., no dynamic allocation is necessary in matrixmultiply. Make sure to achieve maximum concurrency, i.e., do not prevent the creating task from summing its row while subtasks execute.

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

matrixmultiply *xrows xcols-yrows ycols [X-matrix-file Y-matrix-file]*

- The first three parameters are the dimensions of the $X_{xr,xcyr}$ and $Y_{xcyr,yc}$ matrices.
- If specified, the next two parameters are the X and Y input files to be multiplied. Each input 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 3×4 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 input 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.

Print an appropriate error message and terminate the program if there are an invalid number of arguments, the dimension values are less than one, or unable to open the given input files. All matrices in the program are variable sized, and hence, allocated dynamically on the heap.

(b) i. Test for any benefits of concurrency by running the program in parallel:

- Put the following declaration after the arguments have been analysed:

  ```
  uProcessor p[xrows - 1] __attribute__(( unused ));  // number of CPUs
  ```

  This declaration allows the program to access multiple virtual CPUs (cores). One virtual CPU is used for each task calculating a row of the Z matrix. The program starts with one virtual CPU so only xrows - 1 additional CPUs are necessary. Do not run the program with more than 32 rows. Compile the program with the μC++ -multi flag and no optimization.
- Run the program on a multi-core computer with at least 16 or more actual CPUs (cores), with arguments of xrows in the range [1,2,4,8,16] with xcols=yrows of 5000 and ycols of 10000.
- Time each 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. (Increase the number of xcols=yrows and ycols if the timing results are below .1 second.)
- Include all 5 timing results to validate your experiments.

ii. Comment on the user and real times for the experiments.

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 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 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 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 one owner and three condition locks, where the waiting producer, consumer, and barging tasks block on the separate condition locks, and (*has no looping*). (If necessary, you may add more locks.) 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<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.

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. q1nostaticexits.{cc,C,cpp} – code for question 1a, p. 1. **No program documentation needs to be present in your submitted code. No test, 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. q1nostaticexits.txt – contains the information required by question 1b, p. 1.

3. 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.**

4. q2*.txt – contains the information required by question question 2b, p. 3. **Poor documentation of how and/or what is tested can results in a loss of all marks allocated to testing.**

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

6. q3buffer.h, 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.**

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

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

```
SENTINEL:=-1
KIND:=NOBUSY
OPT:= # -multi -O2

CXX = u++                                       # compiler
CXXFLAGS = -g -Wall -Wno-unused-label ${OPT} -MMD -std=c++11 \
        -DSENTINEL=${SENTINEL} -D${KIND}  # compiler flags
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}} # makefile name

OBJECTS01 = q1staticexits.o                     # optional build of given program
EXEC01 = staticexits                            # 0th executable name

OBJECTS1 = q1nostaticexits.o
EXEC1 = nostaticexits                           # 1st executable name

OBJECTS2 = # object files forming 2nd executable with prefix "q2"
EXEC2 = matrixmultiply                          # 2nd executable name

OBJECTS3 = # object files forming 3rd executable with prefix "q3"
EXEC3 = buffer                                  # 3rd executable name

OBJECTS = ${OBJECTS1} ${OBJECTS2} ${OBJECTS3} # all object files
DEPENDS = ${OBJECTS:.o=.d}                      # substitute ".o" with ".d"
EXECS = ${EXEC1} ${EXEC2} ${EXEC3}              # all executables
```

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

.PHONY : all clean

all : ${EXECS}                          # build all executables

q1.o : q1.cc                            # change compiler 1st executable, ADJUST SUFFIX (.cc)
    g++-4.9 ${CXXFLAGS} -c $< -o $@

q1%.o : q1%.cc                          # change compiler 1st executable, ADJUST SUFFIX (.cc)
    g++-4.9 ${CXXFLAGS} -c $< -o $@

${EXEC01} : ${OBJECTS01}
    g++-4.9 ${CXXFLAGS} $^ -o $@

${EXEC1} : ${OBJECTS1}
    g++-4.9 ${CXXFLAGS} $^ -o $@

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

-include ImplKind

ifeq (${IMPLKIND},${KIND})              # same implementation type as last time ?
${EXEC3} : ${OBJECTS3}
    ${CXX} ${CXXFLAGS} $^ -o $@
else
ifeq (${KIND},)                         # no implementation type specified ?
# set type to previous type
KIND=${IMPLKIND}
${EXEC3} : ${OBJECTS3}
    ${CXX} ${CXXFLAGS} $^ -o $@
else                                    # implementation type has changed
.PHONY : ${EXEC3}
${EXEC3} :
    rm -f ImplKind
    touch q3${EXEC3}.h
    sleep 1
    ${MAKE} ${EXEC3} KIND="${KIND}"
endif
endif

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

This makefile is used as follows:

```
$ make matrixmultiply
$ matrixmultiply 3 4 3 xfile yfile
$ matrixmultiply 8 5000 10000
$ 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 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!