This assignment examines synchronization and mutual exclusion, and introduces locks in \( \mu \)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:
     \[
     \text{\$ /usr/bin/time -f \"%Uu %Ss %E\" ./a.out 2 1000000 3.21u 0.02s 0:03.32}
     \]
     (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. Define a dense, single variable polynomial with integer coefficients as the infinite sequence

   \[ c_0x^0 + c_1x^1 + c_2x^2 + \ldots \]

   where the largest non-zero \( c_i \) defines the polynomial degree. The polynomial coefficients are stored in an array, where the coefficient degree is the array index. For example, the polynomial \( 3x^3 + x^2 + 5 \) is represented by the values \((5, 0, 1, 0, 3)\) (reverse order).

   (a) Write a concurrent polynomial multiplication routine with the following interface:

   ```
   typedef ... poly_t; // create an appropriate array type using C arrays (no std::vector)
   void polymultiply( const poly_t & a, const poly_t & b, poly_t & c, const size_t delta );
   ```

   which calculates the product of two polynomials \( a \) and \( b \) as:

   \[
   (p_0, p_1, p_2, \ldots) = (a_0, a_1, a_2, \ldots) \otimes (b_0, b_1, b_2, \ldots)
   \]
```cpp
#include <iostream>
#include <vector>
#include <memory>
// unique_ptr
using namespace std;

int tasks = 1, times = 1000000; // default values

_Task Worker {
   enum { size = 100 };
   void main() {
      for ( int t = 0; t < times; t += 1 ) {
         #if defined( 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
      }
   }
}

int main( int argc, char * argv[] ) {
   try {
      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
}
```

Figure 1: Stack versus Dynamic Allocation

where

\[
\begin{align*}
    p_0 &= a_0 b_0 \\
    p_1 &= a_0 b_1 + a_1 b_0 \\
    p_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\
    & \vdots \\
    p_n &= a_0 b_n + a_1 b_{n-1} + a_2 b_{n-2} + \cdots + a_{n-1} b_1 + a_n b_0
\end{align*}
\]

For simplicity, assume the polynomials \( a \) and \( b \) are the same degree, i.e., the array of coefficients for each have the same length.

Implement the concurrent polynomial-multiply by creating a task to calculate a subset of the product polynomial. In particular, rather than calculating a consecutive section of the product polynomial, a task “spaces” out the elements it is calculating by working on elements \( i + \text{delta} \), where \( i \) is the starting position in the product vector and \( \text{delta} \) is the number of elements to skip over before calculating the next element. For example, if polynomials \( a \) and \( b \) have 5,000 elements, \( p \) has \( 2 \times 5000 - 1 \) elements. Use \( \lfloor \sqrt{|p|} \rfloor \) for
delta (99 in this case) and the number of tasks to perform the calculation. Hence, task_0 calculates the values for elements 0, 99, 198, etc., while task_1 calculates the values for elements 1, 100, 199, etc.

Create concurrency using:

i. implicit COFOR statement.
ii. implicit _Actor type.
   All information for the actor to compute its information must be passed to the actor in an initial message not via the actor’s constructor.
iii. explicit _Task type.
   The public task interface is:
   
   ```cpp
   _Task Multiply {
       public:
       Multiply( const poly_t & a, const poly_t & b, poly_t & c,
                  const size_t startIndex, const size_t endIndex, const size_t delta );
   }
   ```

   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 segment of the product vector. Make sure to achieve maximum concurrency, i.e., do not prevent the creating task from working 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 polymultiply routine, except for message creation in the actor version. Minimal dynamic allocation is allowed in the program main.

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

```
polymult [-p processors (> 0)] [-t numCoeff (> 0)]
```

The optional -p command option specifies the amount of parallelism (see below). The presence of the time option -t indicates the program mode.

i. Without -t, the program mode reads two polynomials and prints the product coefficients. Input and output is specified as follows:

   - The input file contains lists of coefficient values. Each list starts with the number of values in that list. For example, the input file:
     ```
     5 5 0 1
     0 3
     ```
     contains the 5 coefficients 5 0 1 0 3. (Values can be separated by white-space characters and appear across any number of lines.)

     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.

     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.

   - Print the original polynomials and the product in reverse order as follows:
     ```
     3x^4 + 1x^2 + 5x^0
     2x^4 + 1x^3 + 4x^0
     6x^8 + 3x^7 + 2x^6 + 1x^5 + 22x^4 + 5x^3 + 4x^2 + 20x^0
     ```

ii. With -t, the program mode creates two polynomials of size numCoeff, initializes each to the values 1..numCoeff, and prints no values (used for timing experiments). Note, data structures like std::vector with subscript checking double the execution time of programs with significant subscripting, which is why an array must be used.

Print an appropriate usage message and terminate the program if there are missing/invalid number arguments (e.g., the processor/numCoeff values are less than one) or unable to open the given input files.

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; otherwise, it is
set by using the -p option. 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 \( N = \{1, 2, 4, 8, 16\} \) and a polynomial of degree 75,000. Compile the program, in turn,
     with each of the 3 preprocessor variables, using the \( \mu \text{C++} \) multi flag and no optimization.
   • Time the executions using the \texttt{time} command:
     \[
     \texttt{\$ /usr/bin/time -f "\%Uu \%Ss \%Er \%Mkb\" polymult -p N -t \{75000\}}
     \]
     Output from \texttt{time} differs depending on the shell, so use the system \texttt{time} command. Compare the
     real time (0:05.67\( r \)) only, which is the time to complete the computation.
   • If necessary, change the command-line parameters to adjust program execution into the range 1
to 100 seconds. (Timing results below 1 second are inaccurate.)
   • 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 implemen-
tations.

(c)  i. Repeat the experiment in 2b adding the -O2 compilation flag and using a polynomial of degree
     150,000.
   • 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 implemen-
tations.

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

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

which creates a bounded buffer of size \( \mathrm{size} \), and supports multiple producers and consumers. You may
\textit{only} use \texttt{uCondLock} and \texttt{uOwnerLock} to implement the necessary synchronization and mutual exclusion
needed by the bounded buffer.

Implement the \texttt{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 con-
   sumer tasks block on the separate condition locks. (If necessary, you may add more locks.) The
   reason there is barging in this solution is that \texttt{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 \textit{no} busy waiting when waiting for buffer entries to become empty or full. In this approach, use
   \textit{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 bargaining 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:

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

#ifdef NOBUSY
    // no busy waiting 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:

```c
_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:

```c
_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: 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:

```
```

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

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

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

(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:
  ```bash
  $ /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 nonbusy 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**

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. q2polymult.h, q2*.{h,cc,C,cpp} – code for question question question 2, 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 and 2c, p. 4.

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

5. q3buffer.h, q3*.{h,cc,C,cpp} – code for question 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 2, p. 1 and 3a, p. 4 by inserting the object-file names matching your source-file names.

```makefile
# compiler
CXX = u++
CXXFLAGS = -g $(OPT) -Wall -Wextra -MMD -D"${PIMPL}" -D"${BIMPL}" \
        -DSENTINEL="${SENTINEL}" # compiler flags
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}} # makefile name
OBJECTS1 = # object files forming 1st executable with prefix “q2”
EXEC1 = polymult # 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 PolyImpl
ifeq ($(POLYIMPL),$(PIMPL)) # same implementation type as last time ?
$(EXEC1) : $(OBJECTS1)
   $(CXX) $(CXXFLAGS) $(CXXFLAGS) $(CXX) $(OBJECTS1) $(EXEC1) $(EXEC2) $(EXEC1) $(EXEC2) $(EXEC1)
   $(MAKE) $(EXEC1) PIMPL=$(PIMPL)
else # implementation type has changed => rebuilt
.PHONY : $(EXEC1)
$(EXEC1) :
   rm -f PolyImpl
   touch q2polymult.h
   sleep 1
   $(MAKE) $(EXEC1) PIMPL=$(PIMPL)
endif

PolyImpl :
   echo "POLYIMPL=$(PIMPL)" > PolyImpl
   sleep 1
```
include BufImpl

ifeq (${BUFIMPL},${BIMPL})
    ${EXEC2} : ${OBJECTS2}
    ${CXX} ${CXXFLAGS} $^ -o $@
else
    .PHONY : ${EXEC2}
    ${EXEC2} :
        rm -f BufImpl
touch q3buffer.h
    sleep 1
    ${MAKE} ${EXEC2} BIMPL="${BIMPL}"
endif

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

$(OBJECTS) : $(MAKEFILE_NAME)
    # OPTIONAL : changes to this file => recompile
    -include $(DEPENDS)
    # include *.d files containing program dependences

clean :
    rm -f *.d *.o $(EXECS) PImpl BufImpl

This makefile is used as follows:

$ make polymult PIMPL=CFOR
$ polymult ...
$ make polymult PIMPL=ACTOR
$ polymult ...
$ make polymult PIMPL=TASK
$ polymult ...
$ 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 polymult 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!