CS 343 Fall 2016 – Assignment 4
Instructors: Peter Buhr and Aaron Moss
Due Date: Monday, November 7, 2016 at 22:00
Late Date: Wednesday, November 9, 2016 at 22:00

October 26, 2016

This assignment introduces complex locks in μC++ and continues examining synchronization and mutual exclusion. Use it to become familiar with these new facilities, and ensure you use these concepts in your assignment solution. **(Tasks may *not* have public members except for constructors and/or destructors.)**

1. Given the C++ program in Figure 1, run the program separately without any preprocessor variables, and with preprocessor variables DARRAY, VECTOR1 and VECTOR2 for 1, 2, and 4 tasks. Use compiler flags -O2 -multi -nodebug.

   Compare the versions of the program and different numbers of tasks with respect to performance by doing the following:

   - Time the execution using the time command:
     ```
     $ /usr/bin/time -f "%Uu %Ss %E" ./a.out 2 10000000
     3.21u 0.02s 0:03.32 100.0%
     ```
     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 (if necessary) to get the real time in the range 5 to 50 seconds. (Timing results below 0.1 second are inaccurate.) It may be necessary to scale the times for the different versions.
   - Run 4 experiments for each version with the number of tasks set to 1, 2, and 4. Include all 12 timing results to validate your experiments.
   - Explain the relative differences in the timing results with respect to scaling the number of tasks for each version.
   - Very briefly (2-4 sentences) speculate on the performance difference among the versions.

2. Consider the following situation involving a tour group of V tourists. The tourists arrive at the Louvre Museum for a tour. However, a tour can only be composed of G people at a time, otherwise the tourists cannot hear what the guide is saying. As well, there are 2 kinds of tours available at the Louvre: picture and statue art. Therefore, each group of G tourists must vote amongst themselves to select the kind of tour they want to take. During voting, tasks must block until all votes are cast, i.e., assume a secret ballot. Once a decision is made, the tourists in that group proceed on the tour.

   To simplify the problem, the program only has to handle cases where the number of tourists in a group is odd, precluding a tie vote, and the total number of tourists is evenly divisible by the tour-group size so all groups contain the same number of tourists.

   Implement a vote-tallier for G-way voting as a class using *only*:

   a) uCondLock and uOwnerLock to provide mutual exclusion and synchronization (eliminate busy waiting using a signaller flag and extra condition lock).
   b) uBarrier to provide mutual exclusion and synchronization. Note, a uBarrier has implicit mutual exclusion so it is only necessary to manage the synchronization. As well, only the basic aspects of the uBarrier are needed to solve this problem.
   c) uSemaphore used as binary semaphore to provide mutual exclusion and synchronization.

```
#include <vector>
#include <memory>               // unique_ptr
#include <cstdlib>              // atoi
using namespace std;

int tasks = 1, times = 10000000;

_Task Worker {
    enum { size = 100 };
    void main() {
        for ( int i = 0; i < times; i += 1 ) {
#if defined( IMPLKIND_DARRAY )
            unique_ptr<volatile int []> arr( new volatile int[size] );
            for ( int i = 0; i < size; i += 1 ) arr[i] = i;
#elif defined( IMPLKIND_VECTOR1 )
            vector<int> arr( size );
            for ( int i = 0; i < size; i += 1 ) arr.at(i) = i;
#elif defined( IMPLKIND_VECTOR2 )
            vector<int> arr;
            for ( int i = 0; i < size; i += 1 ) arr.push_back(i);
#else // STACK ARRAY
            volatile int arr[size] __attribute__ (( unused )); // prevent unused warning
            for ( int i = 0; i < size; i += 1 ) arr[i] = i;
#endif
        }
    }
};
void uMain::main() {
    switch ( argc ) {
      case 3:
        times = atoi( argv[2] );
      case 2:
        tasks = atoi( argv[1] );
    }
    uProcessor p[tasks - 1];
    Worker workers[tasks];
}
```

Figure 1: Stack versus Dynamic Allocation

No busy waiting is allowed and barging tasks can spoil an election and must be prevented. Figure 2 shows the different forms for each μC++ vote-tallier implementation (you may add only a public destructor and private members), where the preprocessor is used to conditionally compile a specific interface. This form of header file removes duplicate code. When the vote-tallier is created, it is passed the size of a group and a printer for printing state transitions. There is only one vote-tallying object created for all of the voters, who share a reference to it. Each tourist task calls the vote method with their id and a vote of either Picture or Statue, indicating their desire for a picture or statue tour. The vote routine does not return until group votes are cast; after which, the majority result of the voting (Picture or Statue) is returned to each voter. The groups are formed based on voter arrival; e.g., for a group of 3, if voters 2, 5, 8 cast their votes first, they form the first group, etc. Hence, all voting is serialized. An appropriate preprocessor variable is defined on the compilation command using the following syntax:

```
u++ -DIMPLTYPE_SEM -c TallyVotesSEM.cc
```

The interface for the voting task is (you may add only a public destructor and private members):

```
_Task Voter {
  public:
    enum States { Start = 'S', Vote = 'V', Block = 'B', Unblock = 'U', Barging = 'b',
                  Complete = 'C', Finished = 'F' };
    Voter( unsigned int id, TallyVotes &voteTallier, Printer &printer );
};
```

```
#if defined( IMPLTYPE_MC )          // mutex/condition solution
// includes for this kind of vote-tallier
class TallyVotes {
    // private declarations for this kind of vote-tallier
#elif defined( IMPLTYPE_SEM )       // semaphore solution
// includes for this kind of vote-tallier
class TallyVotes {
    // private declarations for this kind of vote-tallier
#elif defined( IMPLTYPE_BAR )       // barrier solution
// includes for this kind of vote-tallier
_Cormonitor TallyVotes : public uBarrier {
    // private declarations for this kind of vote-tallier
#else
    #error unsupported voter type
#endif
    // common declarations
  public:                           // common interface
    TallyVotes( unsigned int group, Printer &printer );
    enum Tour { Picture, Statue };
    Tour vote( unsigned int id, Tour ballot );
};
```

Figure 2: Tally Voter Interfaces

The task main of a voting task looks like:

- yield a random number of times, between 0 and 19 inclusive, so all tasks do not start simultaneously
- print start message
- yield once
- vote
- yield once
- print finish message

Note that each task votes only once. Yielding is accomplished by calling yield(times) to give up a task's CPU time-slice a number of times.

All output from the program is generated by calls to a printer, excluding error messages. The interface for the printer is (you may add only a public destructor and private members).

```
_Monitor / _Cormonitor Printer {    // chose one of the two kinds of type constructor
  public:
    Printer( unsigned int voters );
    void print( unsigned int id, Voter::States state );
    void print( unsigned int id, Voter::States state, TallyVotes::Tour vote );
    void print( unsigned int id, Voter::States state, unsigned int numBlocked );
};
```

(Monitors will be discussed shortly, and are classes with public methods that implicitly provide mutual exclusion.) The printer attempts to reduce output by storing information for each voter until one of the stored elements is overwritten. When information is going to be overwritten, all the stored information is flushed and storing starts again. Output must look like that in Figure 3.

Each column is assigned to a voter with an appropriate title, e.g., "Voter0", and a column entry indicates its current status:

State	Meaning
S	starting
V *b*	voting with ballot *b* (p/s)
B *n*	blocking during voting, *n* voters waiting (including self)
U *n*	unblocking after group reached, *n* voters still waiting (not including self)
b	barging into voter and having to wait for signalled tasks
C	group is complete and voting result is computed
F *b*	finished voting and result of vote is *b* (p/s)

```
 1  $ vote 3 1 103
 2  Voter0  Voter1  Voter2
 3  ======= ======= =======
 4  S               S
 5  V p
 6  C               V p
 7                  C
 8  F p     ...     ...
 9          S
10  ...     ...     F p
11          V s
12          C
13  ...     F s     ...
14  =================
15  All tours started
```

```
 1  $ vote 6 3 16219
 2  Voter0  Voter1  Voter2  Voter3  Voter4  Voter5
 3  ======= ======= ======= ======= ======= =======
 4                                  S
 5                                  V p
 6                          S       B 1
 7                          V p
 8  S                       B 2             S
 9  V s
10  C                               U 1
11  F p     ...     ...     ...     ...     ...
12                  S                       b
13  ...     ...     ...     ...     F p     ...
14          S       b       U 0
15  ...     ...     ...     F p     ...     ...
16                                          V s
17          b       V s                     B 1
18          V p     B 2
19          C                               U 1
20  ...     F s     ...     ...     ...     ...
21                  U 0
22  ...     ...     ...     ...     ...     F s
23  ...     ...     F s     ...     ...     ...
24  =================
25  All tours started
```

Figure 3: Voters: Example Output

Information is buffered until a column is overwritten for a particular entry, which causes the buffered data to be flushed. If there is no new stored information for a column since the last buffer flush, an empty column is printed. When a task finishes, the buffer is flushed immediately, the state for that object is marked with F, and all other objects are marked with "...". After a task has finished, no further output appears in that column. All output spacing can be accomplished using the standard 8-space tabbing. Buffer any information necessary for printing in its internal representation; **do not build and store strings of text for output**. Calls to perform printing may be performed from the vote-tallier and/or a voter task (you decide where to print).

For example, in line 4 of the left hand example in Figure 3, Voter0 has the value "S" in its buffer slot, Voter1 is empty, and Voter2 has value "S". When Vote0 attempts to print "V p", which overwrites its current buffer value of "S", the buffer must be flushed generating line 4. Voter0's new value of "V p" is then inserted into its buffer slot. When Voter0 attempts to print "C", which overwrites its current buffer value of "V p", the buffer must be flushed generating line 5, and no other values are printed on the line because the print is consecutive (i.e., no intervening call from another object). Then Voter0 inserts value "C" and Voter2 inserts value "V p" into the buffer. When Vote2 attempts to print "C", which overwrites its current buffer value of "V p", the buffer must be flushed generating line 6, and so on.

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

```
vote [ V [ G [ Seed ] ] ]
```

V is the size of a tour, i.e., the number of voters (tasks) to be started (multiple of G); if V is not present, assume a value of 6. G is the size of a tour group (odd number); if G is not present, assume a value of 3. Seed is the seed for the random-number generator and must be greater than 0. If the seed is unspecified, use a random value like the process identifier (getpid) or current time (time), so each run of the program generates different output. Use the monitor MPRNG to safely generate random values (monitors will be discussed shortly). Note, because of the non-deterministic execution of concurrent programs, multiple runs with a common seed may not generate the same output. Nevertheless, shorts runs are often the same so the seed can be useful for testing. 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.

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. q1*.txt – contains the information required by question 1, p. 1.
2. MPRNG.h – random number generator (provided)
3. q2tallyVotes.h, q2*.{h,cc,C,cpp} – code for question question 2, p. 1. **Program documentation must be present in your submitted code. No user or system documentation is to be submitted for this question.**
4. q2*.testtxt – test documentation for question 2, p. 1, which includes the input and output of your tests. **Poor documentation of how and/or what is tested can results in a loss of all marks allocated to testing.**
5. Use the following Makefile to compile the programs for question 2, p. 1:

```
KIND:=STACK
TYPE:=MC

CXX = u++                                        # compiler
CXXFLAGS = -g -multi -Wall -Wno-unused-label -MMD -O2 -DIMPLKIND_${KIND} \
                -DIMPLTYPE_${TYPE}
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}} # makefile name

OBJECTS01 = q1new.o                              # optional build of given program
EXEC01 = new

OBJECTS2 = q2tallyVotes${TYPE}.o # list of object files for question 1 prefixed with "q2"
EXEC2 = vote

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

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

.PHONY : all clean

all : ${EXECS}                                   # build all executables

-include ImplKind

q1new.o : q1new.cc                  # add flags 1st executable, ADJUST SUFFIX (.cc)
     ${CXX} ${CXXFLAGS} -nodebug -c $< -o $@

ifeq (${IMPLKIND},${KIND})               # same implementation type as last time ?
${EXEC01} : ${OBJECTS01}
     ${CXX} ${CXXFLAGS} -nodebug $^ -o $@
else
ifeq (${KIND},)                     # no implementation type specified ?
# set type to previous type
KIND=${IMPLKIND}
${EXEC01} : ${OBJECTS01}
     ${CXX} ${CXXFLAGS} -nodebug $^ -o $@
else                               # implementation type has changed
.PHONY : ${EXEC01}
${EXEC01} :
     rm -f ImplKind
     touch q1new.cc
     sleep 1
     ${MAKE} ${EXEC01} KIND="${KIND}"
endif
endif
```

```
ImplKind :
	echo "IMPLKIND=${KIND}" > ImplKind
	sleep 1

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

-include ImplType

ifeq (${IMPLTYPE},${TYPE})                  # same implementation type as last time ?
${EXEC2} : ${OBJECTS2}
	${CXX} ${CXXFLAGS} $^ -o $@
else
ifeq (${TYPE},)                             # no implementation type specified ?
# set type to previous type
TYPE=${IMPLTYPE}
${EXEC2} : ${OBJECTS2}
	${CXX} ${CXXFLAGS} $^ -o $@
else                                        # implementation type has changed
.PHONY : ${EXEC2}
${EXEC2} :
	rm -f ImplType
	touch q2tallyVotes.h
	sleep 1
	${MAKE} ${EXEC2} TYPE="${TYPE}"
endif
endif

ImplType :
	echo "IMPLTYPE=${TYPE}" > ImplType
	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 ${EXEC01} ${EXECS} ImplKind ImplType
```

This makefile will be invoked as follows:

```
$ make new KIND=STACK
$ new ...
$ make new KIND=DARRAY
$ new ...
$ make vote TYPE=MC
$ vote ...
$ make vote TYPE=SEM
$ vote ...
$ make vote TYPE=BAR
```

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