CS 343 Fall 2012 – Assignment 5
Instructor: Martin Karsten
Due Date: Monday, November 19, 2012 at 22:00
Late Date: Wednesday, November 21, 2012 at 22:00

November 8, 2012

This assignment has advanced usage of monitors and introduces task communication in μC++. Use it to become familiar with these new facilities, and ensure you use these concepts in your assignment solution.

1. 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:

   (a) μC++ monitor using external scheduling,

   (b) μC++ monitor using internal scheduling,

   (c) μC++ monitor that simulates a general automatic-signal monitor,

   (d) Java monitor using internal scheduling. To prevent barging tasks, use a ticket approach dividing entering tasks into different groups. You may not use java.util.concurrent available in Java 1.5 or later. (You may freely use the Java concurrency code in the example programs.)

   ALTERNATIVE: μC++ class simulating non-priority monitor using uOwnerLock and uCondLock. In this case, you do not need to build target q1Driver.**class**, but support IMPLTYPE_LOCK (see below).

   (e) μC++ server task performing the maximum amount of work on behalf of the client (i.e., very little code in member vote). The output for this implementation differs from the monitor output because all voters print blocking and unblocking messages, since they all block allowing the server to form a group.

   No busy waiting is allowed and barging tasks can spoil an election and must be prevented. Figure 1 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 true or false, indicating their desire for the picture or statue tour, respectively. The vote routine does not return until group votes are cast; after which, the majority result of the voting (true or false) 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_INT -c TallyVotesINT.cc
   ```

   μC++ does not provide an automatic-signal monitor so it must be simulated using the explicit-signal mechanisms. For the simulation, create an include file, called AutomaticSignal.h, which defines the following preprocessor macros:

```
#if defined( IMPLTYPE_LOCK )          // mutex/condition solution
// includes for this kind of vote-tallier
class TallyVotes {
    // private declarations for this kind of vote-tallier
#elif defined( IMPLTYPE_EXT )         // external scheduling monitor solution
// includes for this kind of vote-tallier
_Monitor TallyVotes {
    // private declarations for this kind of vote-tallier
#elif defined( IMPLTYPE_INT )         // internal scheduling monitor solution
// includes for this kind of vote-tallier
_Monitor TallyVotes {
    // private declarations for this kind of vote-tallier
#elif defined( IMPLTYPE_AUTO )        // automatic-signal monitor solution
// includes for this kind of vote-tallier
_Monitor TallyVotes {
    // private declarations for this kind of vote-tallier
#elif defined( IMPLTYPE_TASK )        // internal/external scheduling task solution
_Task TallyVotes {
    // private declarations for this kind of vote-tallier
#else
    #error unsupported voter type
#endif
    // common declarations
  public:                             // common interface
    TallyVotes( unsigned int group, Printer &prt );
    bool vote( unsigned int id, bool ballot );
};
```

Figure 1: Tally Voter Interfaces

```
#define AUTOMATIC_SIGNAL ...
#define WAITUNTIL( pred, before, after ) ...
#define RETURN( expr... ) ...     // gcc variable number of parameters
```

These macros must provide a *general* simulation of automatic-signalling, i.e., the simulation cannot be specific to this question. Macro AUTOMATIC_SIGNAL is placed only once in an automatic-signal monitor as a private member, and contains any private variables needed to implement the automatic-signal monitor. Macro WAITUNTIL is used to wait until the pred evaluates to true. If a task must block waiting, the expression before is executed before the wait and the expression after is executed after the wait. Macro RETURN is used to return from a public routine of an automatic-signal monitor, where expr is optionally used for returning a value. For example, a bounded buffer implemented as an automatic-signal monitor looks like:

```
_Monitor BoundedBuffer {
    AUTOMATIC_SIGNAL;
    int front, back, count;
    int Elements[20];
  public:
    BoundedBuffer() : front(0), back(0), count(0) {}
    _Nomutex int query() { return count; }

    void insert( int elem ) {
        WAITUNTIL( count < 20, , );       // empty before/after
        Elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
        RETURN();
    }
```

```
        int remove() {
            WAITUNTIL( count > 0, , );          // empty before/after
            int elem = Elements[front];
            front = ( front + 1 ) % 20;
            count -= 1;
            RETURN( elem );
        }
    };
```

Make absolutely sure to *always* have a RETURN() macro at the end of each mutex member. As well, the macros must be self-contained, i.e., no direct manipulation of variables created in AUTOMATIC_SIGNAL is allowed from within the monitor.

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', Complete = 'C', Finished = 'F' };
        Voter( unsigned int id, TallyVotes &voteTallier, Printer &prt );
    };
```

The task main of a voting task looks like:

- print start message
- yield a random number of times, between 0 and 19 inclusive, so all tasks do not start simultaneously
- randomly calculate a ballot (true or false)
- vote at the vote-tallier

Note that each task votes only once. In μC++, yielding is accomplished by calling yield(times) to give up a task's CPU time-slice a number of times. In Java, yielding is accomplished by calling yield() an appropriate number of times, where the number of yields may have to be larger than 19 to get intermixing of execution among threads.

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, bool vote );
        void print( unsigned int id, Voter::States state, unsigned int numBlocked );
    };
```

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 2.

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* (0/1)
B *n*	blocking during voting, *n* voters waiting (including self)
U *n*	unblocking after group reached, *n* voters still waiting (not including self)
C	group is complete and voting result is computed
F *b*	finished voting and result of vote is *b* (0/1)

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

```
 1  % vote 3 1
 2  Voter0  Voter1  Voter2
 3  ======= ======= =======
 4  S       S       S
 5                  V 0
 6                  C
 7  ...     ...     F 0
 8  V 1
 9  C
10  F 1     ...     ...
11          V 1
12          C
13  ...     F 1     ...
14  =================
15  All tours started
```

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

Figure 2: Voters: Example Output

output spacing can be accomplished using the standard 8-space tabbing. Buffer any information necessary for printing in 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 right hand example in Figure 2, Voter1 has the value "S" in its buffer slot, and the first buffer slot is full and the last 4 slots are empty. When Vote1 attempts to print "V 1", which overwrites its current buffer value of "S", the buffer must be flushed generating line 4. Voter1's new value of "V 1" is then inserted into its buffer slot. When Vote1 attempts to print "B 1", which overwrites its current buffer value of "V 1", 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).

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) for μC++ and Date.getTime() for Java, so each run of the program generates different output. Use the following monitor to safely generate random values:

```
_Monitor PRNG {
  public:
    PRNG( unsigned int seed = 1009 ) { srand( seed ); }  // set seed
    void seed( unsigned int seed ) { srand( seed ); }    // set seed
    unsigned int operator()() { return rand(); }          // [0,UINT_MAX]
    unsigned int operator()( unsigned int u ) { return operator()() % (u + 1); } // [0,u]
    unsigned int operator()( unsigned int l, unsigned int u ) { return operator()( u - l ) + l; } // [l,u]
};
```

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.

See Understanding Control Flow with Concurrent Programming using μC++ , Sections 9.11.1, 9.11.3.3, 9.13.5, for information on automatic-signal monitors and Section 9.12 for a discussion of simulating an automatic-signal monitor with an explicit-signal monitor.

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 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. MPRNG.h, AutomaticSignal.h, q1tallyVotes.h, q1*.{h,cc,C,cpp,java} – code for question question 1, p. 1. **Program documentation must be present in your submitted code. No user or system documentation is to be submitted for this question.**
2. q1tallyVotes.testtxt – test documentation for question 1, p. 1, which includes the input and output of your tests, documented by the use of script and after being formatted by scriptfix. **Poor documentation of how and/or what is tested can results in a loss of all marks allocated to testing.**
3. Use the following Makefile for question 1, p. 1 to allow the different implementation types to be compiled:

```
TYPE:=EXT

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

OBJECTS = q1tallyVotes${TYPE}.o # object files forming 1st executable with prefix "q1"
DEPENDS = ${OBJECTS:.o=.d}                    # substitute ".o" with ".d"
EXEC = vote

.PHONY : clean

all : ${EXEC} q1Driver.class                  # build all executables

-include ImplType

ifeq (${IMPLTYPE},${TYPE})                    # same implementation type as last time ?
${EXEC} : ${OBJECTS}
	${CXX} $^ -o $@
else
ifeq (${TYPE},)                               # no implementation type specified ?
# set type to previous type
TYPE=${IMPLTYPE}
${EXEC} : ${OBJECTS}
	${CXX} $^ -o $@
else                                          # implementation type has changed
.PHONY : ${EXEC}
${EXEC} :
	rm -f ImplType
	touch q1tallyVotes.h
	${MAKE} ${EXEC} TYPE="${TYPE}"
endif
endif
```

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

${OBJECTS} : ${MAKEFILE_NAME}                 # OPTIONAL : changes to this file => recompile

-include ${DEPENDS}

q1Driver.class : # class files forming 2nd executable with prefix "q1"

%.class : %.java ${MAKEFILE_NAME}
    javac $<

clean :                                       # remove files that can be regenerated
    rm -f *.d *.o ${EXEC} *.class ImplType
```

This makefile will be invoked as follows:

```
make vote TYPE=LOCK   # if implemented
vote ...
make vote TYPE=EXT
vote ...
make vote TYPE=INT
vote ...
make vote TYPE=AUTO
vote ...
make vote TYPE=TASK
vote ...
make q1Driver.class   # if implemented
java q1Driver ...
```

Put this Makefile in the directory with the programs, name the source files as specified above, and enter the appropriate make to compile a specific version of 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. **If the makefile fails or does not produce correctly named executables, or if a program does not compile, you receive zero for all "Testing" marks.**

Follow these guidelines. Your grade depends on it!