CS 343 Winter 2018 – Assignment 5
Instructor: Peter Buhr
Due Date: Wednesday, March 21, 2018 at 22:00
Late Date: Friday, March 23, 2018 at 22:00

February 12, 2018

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

1. Watch the video clip http://www.youtube.com/watch?v=ByPrDPbdRhc from the Dr. Who episode "Blink". **Warning: do not watch it alone!** At the climax, is there a livelock or deadlock among the Angels? Explain the livelock/deadlock in detail. (You have to be generous as to what the Angels can see.)

2. (a) 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 3 kinds of tours available at the Louvre: pictures, statues and gift shop. Therefore, each group of G tourists must vote among themselves to select the kind of tour to take. Voting is a *ranked ballot*, where each tourist ranks the 3 tours with values 0, 1, 2, where 2 is the highest rank. Tallying the votes sums the ranks for each kind of tour and selects the highest ranking. If tie votes occur among rankings, prioritize the results by gift shop, pictures, and then statues. During voting, tasks 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 specified tour.

   To simplify the problem, the program only has to handle cases where the 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:

   i. μC++ monitor using external scheduling,

   ii. μC++ monitor using internal scheduling,

   iii. μC++ monitor using only internal scheduling but simulates a Java monitor,
   In a Java monitor, there is only *one* condition variable and calling tasks can barge into the monitor ahead of signalled tasks. To simulate barging use the following routines in place of normal calls to condition-variable wait and signal:

```
void TallyVotes::wait() {
    bench.wait();                          // wait until signalled
    while ( rand() % 2 == 0 ) {            // multiple bargers allowed
        _Accept( vote ) {                  // accept barging callers
        } _Else {                          // do not wait if no callers
        } // _Accept
    } // while
}

void TallyVotes::signalAll() {             // also useful
    while ( ! bench.empty() ) bench.signal(); // drain the condition
}
```

   This code randomly accepts calls to the interface routines, if a caller exists. Only condition variable bench may be used and it may only be accessed via member routines wait() and signalAll(). Hint: to control barging tasks, use a ticket counter.

   iv. μC++ monitor that simulates a general automatic-signal monitor,
   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:

```
_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();                        // no return value
    }

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

Figure 1: Automatic signal monitor

```
#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, 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 the *optional* return value. Figure 1 shows a bounded buffer implemented as an automatic-signal monitor.

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.

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.

v. 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 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 number of voters, size of a voting 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 ranked vote, indicating their desire for a picture, statue, or gift-shop tour. The vote routine does not return until group votes are cast; after which, the majority result of the voting (Picture, Statue or GiftShop) is returned to each voter. The groups are formed based on voter arrival; e.g., for a group of 3, if voters

```
#if defined( EXT )                      // external scheduling monitor solution
// includes for this kind of vote-tallier
_Monitor TallyVotes {
    // private declarations for this kind of vote-tallier
#elif defined( INT )                    // internal scheduling monitor solution
// includes for this kind of vote-tallier
_Monitor TallyVotes {
    // private declarations for this kind of vote-tallier
#elif defined( INTB )                   // internal scheduling monitor solution with barging
// includes for this kind of vote-tallier
_Monitor TallyVotes {
    // private declarations for this kind of vote-tallier
    uCondition bench;                   // only one condition variable (you may change the variable name)
    void wait();                        // barging version of wait
    void signalAll();                   // unblock all waiting tasks
#elif defined( AUTO )                   // automatic-signal monitor solution
// includes for this kind of vote-tallier
_Monitor TallyVotes {
    // private declarations for this kind of vote-tallier
#elif defined( 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 voters, unsigned int group, Printer & printer );
    struct Ballot { unsigned int picture, statue, giftshop; };
    enum Tour { Picture = 'p', Statue = 's', GiftShop = 'g', Failed = 'f' };
    Tour vote( unsigned int id, Ballot ballot );
    void done();
};
```

Figure 2: Tally Voter Interfaces

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++ -DINT -c TallyVotesINT.cc
```

When a tourist finishes taking tours and leaves the Louvre Museum, it calls done.

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

```
_Task Voter {
    TallyVotes::Ballot cast() {         // cast 3-way vote
        // O(1) random selection of 3 items without replacement using divide and conquer.
        static unsigned int voting[3][2][2] = { { {2,1}, {1,2} }, { {0,2}, {2,0} }, { {0,1}, {1,0} } };
        unsigned int picture = mprng( 2 ), statue = mprng( 1 );
        return (TallyVotes::Ballot){ picture, voting[picture][statue][0], voting[picture][statue][1] };
    }
  public:
    enum States { Start = 'S', Vote = 'V', Block = 'B', Unblock = 'U', Barging = 'b',
                  Complete = 'C', Finished = 'F', Failed = 'X' };
    Voter( unsigned int id, unsigned int nvotes, TallyVotes & voteTallier, Printer & printer );
};
```

The task main of a voting task performs the following nvotes times:

- yield a random number of times, between 0 and 19 inclusive, so all tasks do not start simultaneously
- print start message
- yield a random number of times, between 0 and 4 inclusive
- vote
- yield a random number of times, between 0 and 4 inclusive

```
1  $ vote 3 1 1
2  V0       V1       V2
3  *******  *******  *******
4  S                 S
5  V 0,2,1
6  C                 V 0,2,1
7  F s      S        C
8           V 1,0,2  F s
9           C
10          F g
11 *****************
12 All tours started
```

```
$ vote 6 3 2
V0       V1       V2       V3       V4       V5
*******  *******  *******  *******  *******  *******
S        S                                   S
                                             V 1,0,2
V 2,0,1                                      B 1
B 2      V 2,0,1
U 0      C        S                          U 1
F p
S                                            F p
V 2,0,1                                      S
B 1               V 2,1,0
                  B 2                        V 0,1,2
         F p                                 C
U 1      S        U 0              S        F p
         V 0,2,1
F p      B 1               S       V 0,1,2
                  F p              B 2
                  S
                  V 2,1,0
                  C
         U 1      F s      b
         F s                       U 0
                                   F s
                           X       S
                                   X
*****************
All tours started
```

Figure 3: Voters: Example Output

- print finish message

Casting a vote is accomplished by calling cast. 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 tour );
    void print( unsigned int id, Voter::States state, TallyVotes::Ballot ballot );
    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 3.

Each column is assigned to a voter with an appropriate title, "V$_i$", and a column entry indicates its current status:

State	Meaning
S	starting
V p,s,g	voting with ballot containing 3 rankings
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 t	finished voting and selected tour is t (p/s/g)
X	failed to form a group

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. 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 of Figure 3, V0 has the value "S" in its buffer slot, V1 is empty, and V2 has value "S". When V0 attempts to print "V 0,2,1", which overwrites its current buffer value of "S", the buffer must be flushed generating line 4. V0's new value of "V 0,2,1" is then inserted into its buffer slot. When V0 attempts to print "C", which overwrites its current buffer value of "V 0,2,1", 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). Then V0 inserts value "C" and V2 inserts value "V 0,2,1" into the buffer. When V2 attempts to print "C", which overwrites its current buffer value of "V 0,2,1", the buffer must be flushed generating line 6, and so on. Note, a group size of 1 means a voter never has to block/unblock.

Interestingly, a vote can fail to establish a quorum (group) even when the number of voters is a multiple of the group size. For example, one voter can execute fast while another executes slow, so the fast voter finishes its voting early and terminates. As a result, the slow voter can encounter a situation where there are insufficient voters to form a group. TallyVotes detects a failure situation when a voter calls done and the number of remaining voters is less than the group size. At this point, any new calls to vote immediately return Failed, and any waiting voters are unblocked and return Failed. For the barrier, a voter calling done in the failure case, must block on the barrier to force waiting voters to unblock.

For example, in the right-hand example of Figure **??**, p. ??, there are 6 voters, 3 voters in a group, and each voter votes twice. Voters V3 and V4 are delayed (e.g., they went to Tom's for a coffee and donut). By looking at the F codes, V0, V1, V5 vote together, V0, V2 V5 vote together, and V1, V2, V4 vote together. Hence, V0, V1, V2, and V5 have voted twice and finished. V3 needs to vote twice and V4 needs to vote again. However, there are now insufficent voters to form a group, so both V3 and V4 fail with X.

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

```
vote [ voters | 'x' [ group | 'x' [ votes | 'x' [ seed | 'x' ] ] ]
```

voters is the size of a tour (> 0), i.e., the number of voters (tasks) to be started (multiple of group). If x or no value for voters is specified, assume 6.

group is the size of a tour group (> 0). If x or no value for group is specified, assume 3.

votes is the number of tours (> 0) each voter takes of the museum. If x or no value for votes is specified, assume 1.

seed is the starting seed for the random-number generator (> 0). If x or no value for seed is specified, initialize the random number generator with an arbitrary seed value (e.g., getpid() or time), so each run of the program generates different output.

Use the monitor MPRNG to safely generate random values. 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.

(b) i. Compare the performance among the five kinds of monitors/task by eliding all output (not even calls to the printer) and doing the following:

- Time the execution using the time command:

```
$ /usr/bin/time -f "%Uu %Ss %Er" vote 100 10 100000 1003
10.17u 0.04s 0:10.19r
```

(Output from time differs depending on your shell, but all provide user, system and real time.) Compare the *user* (10.17u) and *real* (0:10.19r) 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.

- Adjust the number of votes to get real time in the range 10 to 20 seconds. (Timing results below 1 second are inaccurate.) Use the same number of votes for all experiments.
- Include all 3 timing results to validate your experiments.

ii. Very briefly (2-4 sentences) speculate on any performance difference among the locks.

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. AutomaticSignal.h, q2tallyVotes.h, q2*.{h,cc,C,cpp} – code for question question 2a, 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*.testdoc – test documentation for question 2a, 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. q2*.txt – contains the information required by question 2b.
6. Makefile – construct a makefile **similar to those presented in the course** to compile the program for question 2a, p. 1. This makefile is invoked as follows:

```
$ make vote TYPE=EXT
$ vote ...
$ make vote TYPE=INT
$ vote ...
$ make vote TYPE=INTB
$ vote ...
$ make vote TYPE=AUTO
$ vote ...
$ make vote TYPE=TASK
$ vote ...
```

   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.

Follow these guidelines. Your grade depends on it!