CS 343 Fall 2023 – Assignment 5
Instructor: Peter Buhr
Due Date: Wednesday, November 22, 2023 at 23:00
Late Date: Friday, November 24, 2023 at 23:00

November 17, 2023

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: it is scary but not violent.** 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 group can only be composed of G people at a time, otherwise the tourists cannot hear the guide. As well, there are 3 kinds of tours available at the Louvre: pictures, statues and gift shop. Therefore, each tour group must vote 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, e.g.:

```
          P S G                              P S G
tourist1  0 1 2                    tourist1  2 1 0
tourist2  2 1 0                    tourist2  1 2 0
tally      2 2 2 all ties, select G  tally      3 3 0 two ties, select P
```

   During voting, a tourist blocks until all G votes are cast, i.e., assume a secret ballot. Once a decision is made, the tourists in that group proceed on the specified tour. Tourists may take multiple tours, but because of voting, can take the same kind of tour.

   The tour size G may not evenly divide the number of tourists, resulting in a *quorum* failure when the remaining tourists is less than G.

   Implement a general vote-tallier 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. Figure 1 shows a μC++ simulation of Java barging by replacing the normal calls to condition-variable wait and signal with these calls. 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:

```
#define AUTOMATIC_SIGNAL ...
#define WAITUNTIL( pred, before, after ) ...
#define EXIT() ...
```

   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

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

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

Figure 1: Java Simulation

```
_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;
        EXIT();
    }

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

Figure 2: Automatic signal monitor

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 EXIT must be called on return from a public routine of an automatic-signal monitor. Figure 2 shows a bounded buffer implemented as an automatic-signal monitor.

Make absolutely sure to *always* execute the EXIT() macro at the end of each mutex member, either normal or exceptional return. As well, the macros must be abstract (hidden), 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, as they all block allowing the server to form a group.

No unbounded busy-waiting is allowed in any solution, and barging tasks can spoil an election and must

```
#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 (variable name may be changed)
    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
    _Event Failed {};
    struct Ballot { unsigned int picture, statue, giftshop; };
    enum TourKind : char { Picture = 'p', Statue = 's', GiftShop = 'g' };
    struct Tour { TourKind tourkind; unsigned int groupno; };

    TallyVotes( unsigned int voters, unsigned int group, Printer & printer );
    Tour vote( unsigned int id, Ballot ballot );
    void done(
        #if defined( TASK )
        unsigned int id
        #endif
    );
};
```

Figure 3: Tally Vote Interfaces

be avoided/prevented.

Figure 3 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 **(see the Makefile)**. This form of header file removes duplicate code.

At creation, a vote-tallier 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 voter 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, along with a number to identify the tour group (where tours are numbered 1 to N). 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. When a tourist finishes taking tours and leaves the Louvre Museum, it *always* calls done (even if it has a quorum failure).

TallyVotes detects a quorum failure when the number of **remaining voters in the Louvre** is less than the group size. At this point, any new calls to vote immediately raise exception Failed, and any waiting voters must be unblocked so they can raise exception Failed. When a voter calls done, it must cooperate if there is a quorum failure by helping to unblock waiting voters.

```
#include "BargingCheckVote.h"
_Monitor TallyVotes {
    . . .                                   // regular declarations
    BCHECK_DECL;
  public:
    . . .                                   // regular declarations
    Tour vote( unsigned int id __attribute__(( unused )), Ballot ballot ) {
        // acquire mutual exclusion
        VOTER_ENTER( tour-group-size );
        . . .                               // voter code
        VOTER_LEAVE( tour-group-size );
        // release mutual exclusion
        return . . .
    }
};
```

Figure 4: Barging Check Macros: INTB

```
_Task Voter {
    TallyVotes::Ballot cast() __attribute__(( warn_unused_result )) {  // cast 3−way vote
        // O(1) random selection of 3 items without replacement using divide and conquer.
        static const unsigned int voting[3][2][2] = { { {2,1}, {1,2} }, { {0,2}, {2,0} }, { {0,1}, {1,0} } };
        unsigned int picture = prng( 3 ), statue = prng( 2 );
        return (TallyVotes::Ballot){ picture, voting[picture][statue][0], voting[picture][statue][1] };
    }
  public:
    enum States : char { Start = 'S', Vote = 'V', Block = 'B', Unblock = 'U', Barging = 'b',
         Done = 'D', Complete = 'C', Going = 'G', Failed = 'X', Terminated = 'T' };
    Voter( unsigned int id, unsigned int nvotes, TallyVotes & voteTallier, Printer & printer );
};
```

Figure 5: Voter Interface

Note, even when V *is* a multiple of G *and* tourists take multiple tours, a quorum failure can occur. For example, one tour is faster than another or a tourist leaves a tour early and comes back to vote on another tour, so the quick tourist finishes all their tours and terminates. The slower tourists then encounter a situation where there are insufficient tourists to form a quorum for later tours.

Figure 4 shows the macro placement that ***must*** be present only in the INTB tally-votes implementation to test for barging, and defining preprocessor variable BARGINGCHECK triggers barging testing **(see the Makefile)**. If barging is detected, a message is printed and the program continues, possibly printing more barging messages. To inspect the program with gdb when barging is detected, set BARGINGCHECK=0 to abort the program. (Barging checking in the other implementations is superfluous because the monitor/task have implicit barging prevention.)

Figure 5 shows the interface for a voting task (you may add only a public destructor and private members). The task main of a voting task first

- yields a random number of times, between 0 and 19 inclusive, so all tasks do not start simultaneously

and then performs the following nvotes times:

- 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
- print going on tour message
- after all tours, eventually report done and print terminate message

Casting a vote is accomplished by calling member 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*. Figure 6 shows the interface for the printer (you may add only a public destructor and private members). The printer attempts to reduce output by storing information for each voter until one of the stored elements is overwritten.

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

Figure 6: Printer Interface

State	Meaning
S	start
V p, s, g	vote with ballot containing 3 rankings
B n	block during voting, n voters waiting (including self)
U n	unblock after group reached, n voters still waiting (not including self)
b n gn	block barging task (avoidance only), n waiting for signalled tasks to unblock (including self), group number gn of last group that received a voting result
D	block by accepting done (EXT/TASK only)
C t	complete group and voting result is t (p/s/g)
G t gn	go on tour, t (p/s/g) in tour group number gn
X	failed to form a group (quorum failure)
T	voter terminates (after call to done)

Figure 7: Voter Status Entries

When information is going to be overwritten, all the stored information is flushed and storing starts again. Output must look like that in Figure 8.

Each column is assigned to a voter with the titles, "V$_i$", and Figure 7 shows the column entries indicating its current status. 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 terminated, no further output appears in that column. All output spacing can be accomplished using the standard 8-space tabbing (`'\t'`). 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 8 for monitor type EXT, V0 has the value "S" in its buffer slot, V1 is empty, and V2 has value "S". When V0 attempts to print "V 2,0,1", which overwrites its current buffer value of "S", the buffer must be flushed generating line 4. V0's new value of "V 2,0,1" is then inserted into its buffer slot. When V0 attempts to print "C", which overwrites its current buffer value of "V 2,0,1", 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.

For example, in the right-hand example of Figure 8 for monitor type EXT, 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, V2, V5 vote together (group 1), V1, V2 V4 vote together (group 2), and V0, V1, V5 vote together (group 3). Hence, V0, V1, V2, and V5 have voted twice and terminated. V3 needs to vote twice and V4 needs to vote again. However, there are now insufficient voters to form a group, so both V3 and V4 fail with X. Note, V4 it woken up 3 times by terminating tasks before it detects the quorum failure.

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

```
vote [ voters | 'd' [ group | 'd' [ votes | 'd' [ seed | 'd' [ processors | 'd' ] ] ] ] ]
```

```
$ vote 3 1 1
V0      V1      V2
******* ******* *******
S               S
V 2,0,1
C p             V 0,2,1
G p 1           C s
T               G s 2
        S       T
        V 2,0,1
        C p
        G p 3
        T
*****************
All tours started
```

```
$ vote 6 3 2
V0      V1      V2      V3      V4      V5
******* ******* ******* ******* ******* *******
S       S       S                       S
                                        V 1,0,2
                V 2,0,1                 B 1
V 1,2,0         B 2             S
C p             U 1
                G p 1
                S               V 1,0,2 U 0
G p 1                           B 1
S       V 1,0,2
        B 2     V 2,1,0
                C g                     G p 1
        U 1                     U 0     S
        G g 2
        S       G g 2                   V 2,1,0
                T                       B 1
                                G g 2   D
V 2,0,1                         S
B 2     V 0,2,1
U 1     C p             S               U 0
                                V 0,2,1 G p 3
G p 3   G p 3                   B 1     T
T                               D
        T                       D
                                D
                                U 0
                        X       X
                        T       T
*****************
All tours started
```

Figure 8: Voters: Example Output

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

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

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

seed is the starting seed for the random-number generator (> 0). If seed is specified, call set_seed(seed). If d or no value for seed is specified, do nothing as PRNG sets the seed to an arbitrary value.

processors is the number of processors for parallelism (> 0). If d or no value for processors is specified, assume 1. Use this number in the following declaration placed in the program main immediately after checking command-line arguments but before creating any tasks:

```
uProcessor p[processors - 1] __attribute__(( unused )); // create more kernel thread
```

to adjust the amount of parallelism for computation. The program starts with one kernel thread so only processors - 1 additional kernel threads are added.

To obtain semi-repeatable results, all random numbers are generated using the μC++ task-member prng (see Appendix C in the μC++ reference manual). 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) Recompile the program to elide output for timing experiments by adding the following code and using it to bracketing all printer calls **(see the Makefile)**.

```
#ifdef NOOUTPUT
#define PRINT( stmt )
#else
#define PRINT( stmt ) stmt
#endif // NOOUTPUT
PRINT( printer.print( id, Voter::Vote, ballot ) ); // elide printer call
```

i. Compare the performance among the 5 kinds of monitors/task:
   - Time the executions using the time command:
     ```
     $ /usr/bin/time -f "%Uu %Ss %Er %Mkb" vote 100 10 10000 1003
     3.21u 0.02s 0:05.67r 32496kb
     ```
     Output from time differs depending on the shell, so use the system time command. Compare the *user* (3.21u) and *real* (0:05.67r) 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.
   - If necessary, adjust the number of voters and then votes to get real time in range 1 to 100 seconds. (Timing results below 1 second are inaccurate.) Use the same number of votes for all experiments.
   - Include all 5 timing results to validate your experiments.
   - Repeat the experiment using 2 processors and include the 5 timing results to validate your experiments.

ii. State the performance difference (larger/smaller/by how much) among the monitors/task.

iii. As the kernel threads increase, very briefly speculate on any performance difference.

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,testdoc} file, must be ASCII text and not exceed 500 lines in length, using the command fold -w120 *.testdoc | 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. q1*.txt – contains the information required by question 1, p. 1.
2. BargingCheckVote.h – barging checker (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, system or test documentation is to be submitted for this question.**
4. q2*.txt – contains the information required by question 2b.
5. Makefile – construct a makefile **similar to those presented in the course** to compile the program for question 2a, p. 1. **This makefile must NOT contain hand-coded dependencies.** The makefile is invoked as follows:

   ```
   $ make vote VIMPL=EXT
   $ vote ...
   $ make vote VIMPL=INT
   $ vote ...
   $ make vote VIMPL=INTB
   $ vote ...
   $ make vote VIMPL=AUTO OUTPUT=OUTPUT
   $ vote ...
   $ make vote VIMPL=TASK OUTPUT=NOOUTPUT
   $ 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!