CS 343 Fall 2016 – Assignment 5
Instructors: Peter Buhr and Aaron Moss
Due Date: Monday, November 21, 2016 at 22:00
Late Date: Wednesday, November 23, 2016 at 22:00

November 16, 2016

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. Given the μC++ program in Figure 1, run the program separately with preprocessor variable PAD not defined and defined. Use compiler flags -O2 -multi -nodebug.

   Compare the two versions of the program with respect to performance by doing the following:

   - Time the execution using the time command:

     ```
     $ /usr/bin/time -f "%Uu %Ss %E" ./a.out 10000000
     3.21u 0.02s 0:03.32 100.0%
     ```

     Compare the *user* time (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 program command-line argument (if necessary) to get the real time in the range 0.1 to 100 seconds. (Timing results below 0.1 seconds are inaccurate.) Use the same command-line value for all experiments.
   - Run 2 experiments with preprocessor variable PAD not defined and defined. Include both timing results to validate your experiments.
   - Explain the relative differences in the timing results with respect to memory location of the counters.
   - Explain the order of addresses of the global variables.
   - Explain why there is a **void** * cast before each counter when printing their address.

2. 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.)

3. 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 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:

```
#include <iostream>
using namespace std;

#ifdef PAD
#define ALIGN __attribute__(( aligned (64) )) // align on 64-byte boundaries
#else
#define ALIGN
#endif // PAD

volatile long int                               // volatile prevents dead-code removal
    pad1 ALIGN,                                 // bracket counters with alignment
    counter1 ALIGN, counter2 ALIGN,
    pad2 ALIGN;

unsigned long int times = 100000000;

_Task Worker {
    volatile long int &counter;
    void main() {
        for ( unsigned int i = 0; i < times; i += 1 ) {
            counter += 1;
        }
        cout << counter << endl;
    }
  public:
    Worker( volatile long int &counter ) : counter( counter ) {}
};

void uMain::main() {
    switch ( argc ) {
      case 2:
        times = atol( argv[1] );
    }
    cout << (void *)&pad1 << ' ' << (void *)&counter1 << ' '
         << (void *)&counter2 << ' ' << (void *)&pad2 << ' ' << endl;

    uProcessor p;                               // add virtual processor
    Worker w1( counter1 ), w2( counter2 );      // create threads
}
```

Figure 1: Counters

```
void TallyVotes::wait() {
    bench.wait();                               // wait until signalled
    while ( rand() % 5 == 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. Hint: to control barging tasks, use a ticket counter.

(d) 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 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 the *optional* return 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();                        // 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
    }
};
```

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.

(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 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_INT -c TallyVotesINT.cc
```

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

```
#if 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_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( 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 &printer );
    enum Tour { Picture, Statue };
    Tour vote( unsigned int id, Tour ballot );
};
```

Figure 2: Tally Voter Interfaces

```
_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 );
};
```

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 );
};
```

```
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
```

```
$ vote 6 3 16219
Voter0  Voter1  Voter2  Voter3  Voter4  Voter5
======= ======= ======= ======= ======= =======
                                S
                                V p
                        S       B 1
                        V p
S                       B 2             S
V s
C                       U 1
F p     ...     ...     ...     ...     ...
                S
...     ...     ...     F p     ...     ...
        S                       U 0     b
...     ...     ...     ...     F p     ...
                V p
        V p     B 1
        B 2                             V s
        U 1                             C
...     ...     ...     ...     ...     F p
                U 0
...     F p     ...     ...     ...     ...
...     ...     F p     ...     ...     ...
=================
All tours started
```

Figure 3: Voters: Example Output

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)

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. 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. q2*.txt – contains the information required by question 2, p. 1.
3. MPRNG.h – random number generator (provided)
4. AutomaticSignal.h, q3tallyVotes.h, q3*.{h,cc,C,cpp} – code for question question 3, p. 1. **Program documentation must be present in your submitted code. No user or system documentation is to be submitted for this question.**
5. q3vote.testtxt – test documentation for question 3, 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.**
6. Makefile – construct a makefile to compile the program for question 3, p. 1. Put this Makefile in the directory with the program, name the source files as specified above, and when make vote is entered it compiles the program. 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. This makefile will be 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!