# CS 343 Fall 2020 – Assignment 4
# Instructor: Peter Buhr
# Due Date: Monday, November 9, 2020 at 22:00
# Late Date: Wednesday, November 11, 2020 at 22:00

October 29, 2020

This assignment introduces complex locks in $\mu$C++ 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, compare buffering using internal-data versus external-data format. Redirect the program output to /dev/null to discard the output; otherwise, the output distorts the performance measurements.

   (a) Compare the two versions of the program with respect to performance by doing the following:
   - Run the program with preprocessor variables -DINTERNAL and -DEXTERNAL.
   - Time the executions using the time command:

```
$ /usr/bin/time -f "%Uu %Ss %E" ./a.out > /dev/null   # ignore program output
3.21u 0.02s 0:03.32
```

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

int main( int argc, char *argv[] ) {
    int times = 1000000, size = 40;                         // defaults

    try {
        switch ( argc ) {
          case 3: size = stoi( argv[2] ); if ( size <= 0 ) throw 1;
          case 2: times = stoi( argv[1] ); if ( times <= 0 ) throw 1;
          case 1: break;                                    // use defaults
          default: throw 1;
        } // switch
    } catch( ... ) {
        cout << "Usage: " << argv[0] << " [ times (> 0) [ size (> 0) ] ]" << endl;
        exit( 1 );
    } // try

    for ( int i = 0; i < times; i += 1 ) {
#if defined( INTERNAL )
        int intbuf[size];                                   // internal-data buffer
        for ( int i = 0; i < size; i += 1 ) intbuf[i] = i;
        for ( int i = 0; i < size; i += 1 ) cout << intbuf[i] << ' '; // internal buffering
        cout << endl;
#elif defined( EXTERNAL )
        string strbuf;                                      // external-data buffer
        for ( int i = 0; i < size; i += 1 ) strbuf += to_string( i ) + ' '; // external buffering
        cout << strbuf << endl;
#else
#error unknown buffering style
#endif
    } // for
}
```

Figure 1: Internal versus External Buffering

```
enum Intent { WantIn, DontWantIn };
Intent * Last;

_Task Dekker {
    Intent & me, & you;

    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
1           for ( ;; ) {                          // entry protocol, high priority
2               me = WantIn;
3             if ( you == DontWantIn ) break;
4               if ( ::Last == &me ) {
5                   me = DontWantIn;
6                   while ( ::Last == &me ){}
                }
            }
8           CriticalSection();                    // critical section
9           ::Last = &me;                         // exit protocol
10          me = DontWantIn;
        }
    }
  public:
    Dekker( Intent & me, Intent & you ) : me(me), you(you) {}
};
int main() {
    Intent me = DontWantIn, you = DontWantIn;
    ::Last = &me;           // arbitrary who starts as last
    Dekker t0( me, you ), t1( you, me );
}
```

Figure 2: Dekker 2-Thread Mutual Exclusion

(Output from time differs depending on the shell, so use the system time command.) Compare the *user* time (3.21u) only, which is the CPU time consumed solely by the execution of user code (versus system and real time).

- Use the program command-line arguments (as necessary) to adjust program execution into the range 1 to 100 seconds. (Timing results below 1 second are inaccurate.) Use the same command-line values for all experiments, if possible; otherwise, increase/decrease the arguments as necessary and scale the difference in the answer.
- Run both the experiments again after recompiling the programs with compiler optimization turned on (i.e., compiler flag -O2).
- Include 4 timing results to validate the experiments.

(b) State the performance difference (larger/smaller/by how much) between the two versions of the program, and what caused the difference.

(c) State the performance difference (larger/smaller/by how much) between the original and transformed programs when compiler optimization is used.

(d) For interest, change endl to `'\n'` to see if there is any performance difference.

2. Figure 2 shows a Dekker solution to the mutual exclusion problem.

(a) Assume line 6 is replaced with **while** (you == WantIn).

i. Explain which rule of the critical-section game is broken and the steps resulting in failure.

ii. Explain why the broken rule is unlikely to be noticeable even during a test of 100,000 or more tries.

(b) Explain what property of Dekker's algorithm changes if lines 9 and 10 are interchanged and show the steps resulting in the change.

3. (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
tourist1  0 1 2
tourist2  2 1 0
tally     2 2 2 all ties, select G
```

During voting, a tourist blocks 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. Tourists may take multiple tours, but because of voting, end up taking 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 $G$. 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.

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

i. a single uOwnerLock and possibly multiple uCondLocks to provide mutual exclusion and synchronization.

ii. multiple uSemaphores, used as binary rather than counting, to provide mutual exclusion and synchronization.

iii. a single 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.

No busy waiting is allowed in any solution, and barging tasks can spoil an election and must be avoided/-prevented.

Figure 3 shows the different forms for each $\mu$C++ 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. An appropriate preprocessor variable is defined on the compilation command using the following syntax:

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

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 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. For the owner/condition lock, a voter calling done in the failure case may have to pretend to be a barger if signalling is in progress, and hence, must block with other bargers. For the barrier lock, a voter calling done in the failure case may have to block on the barrier to force waiting voters to unblock.

Figure 4 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

```
#if defined( MC )                       // mutex/condition solution
// includes for this kind of vote-tallier
class TallyVotes {
    // private declarations for this kind of vote-tallier
#elif defined( SEM )                    // semaphore solution
// includes for this kind of vote-tallier
class TallyVotes {
    // private declarations for this kind of vote-tallier
#elif defined( 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
    _Event Failed {};
    TallyVotes( unsigned int voters, unsigned int group, Printer & printer );
    struct Ballot { unsigned int picture, statue, giftshop; };
    enum TourKind { Picture = 'p', Statue = 's', GiftShop = 'g' };
    struct Tour { TourKind tourkind; unsigned int groupno; };
    Tour vote( unsigned int id, Ballot ballot );
    void done(
    #if defined( BAR )                  // barrier solution
        unsigned int id
    #endif
    );
};
```

Figure 3: Tally Votes Interfaces

```
_Task Voter {
    Ballot cast() {          // 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 = 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',
          Done = 'D', Complete = 'C', Going = 'G', Failed = 'X', Terminated = 'T' };
    Voter( unsigned int id, unsigned int nvotes, TallyVotes & voteTallier, Printer & printer );
};
```

Figure 4: Voter Interface

```
_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 5: Printer Interface

- 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 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 5 shows the interface for the printer (you may add only a public destructor and private members). (Monitors are 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 6.

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

| 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, $n$ waiting for signalled tasks to unblock (avoidance only, including self), current group number $gn$ being service by tally votes |
| D | block in done (BAR 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 |
| T | voter terminates (after call to done) |

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. 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 6, V0 has the value "S" in its buffer slot, V1 has value "S", and V2 is empty. When V1 attempts to print "V 0,2,1", which overwrites its current buffer value of "S", the buffer must be flushed generating line 4. V1's new value of "V 0,2,1" is then inserted into its buffer slot. When V1 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 V1 inserts value "C" and V0 inserts value "V 2,0,1" into the buffer. Assume V0 attempts to print "C", which overwrites its current buffer value of "V 2,0,1", 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 6, 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 (group 1), V0, V1 V2 vote together (group 2), and V2, V4, 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.

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 0,2,1
V 2,0,1 C s
C p     G s 1
G p 2   T       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
        V 1,0,2
        B 1                             V 2,0,1
V 2,0,1                                 B 2
C p     U 1
G p 1
S               S                       U 0
                V 2,0,1
                B 1                     G p 1
V 2,1,0                                 S
B 2     G p 1
        S
        V 1,2,0
        C p     U 1
U 0     G p 2                   S       b 1 2
        T       G p 2
G p 2           S       S       b 2 2   V 1,0,2
T               b 2 2           V 1,2,0 B 1
                V 1,0,2 b 2 2                           B 2
                C g                     U 1
                G g 3           U 0     G g 3
                        V 0,2,1         T
                        B 1     G g 3
                T       U 0     S
                        X
                        T       X
                                T
*****************
All tours started
```

Figure 6: 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 d 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.

**processors** is the number of processors ($> 0$) for parallelism. If d or no value for processors is specified, assume 1.

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.

Add the following declaration to the program main immediately after checking command-line arguments but before creating any tasks:

```
uProcessor p[processors - 1];  // number of kernel threads
```

to adjust the amount of parallelism for computation. The default value for processors is 1. Since the program starts with one kernel thread, only processors - 1 additional kernel threads are necessary.

(b) Recompile the program with preprocessor option -DNOOUTPUT to suppress output.

i. Compare the performance among the 3 kinds of locks by eliding all output (not even calls to the printer) and doing the following:

- 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 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 3 timing results to validate your experiments.
- Repeat the experiment using 2 processors and include the 3 timing results to validate your experiments.

ii. State the performance difference (larger/smaller/by how much) among the locks as the kernel threads increase.

Use the following to elide output:

```
#ifdef NOOUTPUT
#define PRINT( args... )
#else
#define PRINT( args... ) printer.print( args )
#endif // NOOUTPUT
```

## 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,doc} file, must be ASCII text and not exceed 500 lines in length, using the command fold -w120 *.doc | 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. q2*.txt – contains the information required by question 2, p. 2.
3. MPRNG.h – random number generator (provided)
4. q3tallyVotes.h, q3*.{h,cc,C,cpp}~~, printer.o~~ – code for question question 3a, p. 3. **Program documentation must be present in your submitted code. No user, system or test documentation is to be submitted for this question.**
5. q3*.txt – contains the information required by question 3b.
6. Modify the following Makefile to compile the programs for question 3a, p. 3 by inserting the object-file names matching your source-file names.

```
VIMPL:=MC
OUTPUT:=OUTPUT

CXX = u++                                       # compiler
CXXFLAGS = -g -multi -O2 -Wall -Wextra -MMD -D${VIMPL} -D${OUTPUT} # compiler flags
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}} # makefile name

OBJECTS = q3tallyVotes${VIMPL}.o # list of object files for question 3 prefixed with "q3"
EXEC = vote
```

```
DEPENDS = ${OBJECTS:.o=.d}                  # substitute ".o" with ".d"

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

.PHONY : all clean

all : ${EXEC}                               # build all executables

-include VoteImpl

ifeq (${LOCKVIMPL},${VIMPL})                # same implementation type as last time ?
${EXEC} : ${OBJECTS}
    ${CXX} ${CXXFLAGS} $^ -o $@
else                                        # implementation type has changed => rebuilt
.PHONY : ${EXEC}
${EXEC} :
    rm -f VoteImpl
    touch q3tallyVotes.h
    ${MAKE} ${EXEC} VIMPL="${VIMPL}"
endif

VoteImpl :
    echo "LOCKVIMPL=${VIMPL}" > VoteImpl
    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 ${OBJECTS} ${EXEC} VoteImpl
```

This makefile is invoked as follows:

```
$ make vote VIMPL=MC
$ vote ...
$ make vote VIMPL=SEM
$ vote ...
$ make vote VIMPL=BAR
```

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!**