

CS 343 Winter 2023 – Assignment 4

Instructor: Caroline Kierstead

Due Date: Monday, March 13, 2023 at 22:00

Late Date: Wednesday, March 15, 2023 at 22:00

January 6, 2023

This assignment introduces complex locks in μ 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. Figure 1 is a C++ program comparing buffering using internal-data versus external-data format.
 - (a) Compare the three versions of the program with respect to performance by doing the following:
 - Compile the program with `u++` and run the program with preprocessor variables `-DARRAY`, `-DSTRING` and `-DSTRSTREAM`.
 - Time the executions using the `time` command:

```
$ /usr/bin/time -f "%Uu %Ss %Er %Mkb" ./a.out 1000000 20
3.21u 0.02s 0:03.32
```

(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 command-line arguments `1000000 20` and adjust the times amount (if necessary) to get 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 all 6 timing results to validate the experiments and the number of calls to `malloc`.
 - (b) State the performance and allocation difference (larger/smaller/by how much) between the three versions of the program.
 - (c) State the performance difference (larger/smaller/by how much) when compiler optimization is used.
 - (d) Very briefly (1-2 sentences) speculate on the cause of the performance difference between `ARRAY` and `STRING/STRSTREAM`.
2. Figure 2, p. 3 shows a solution to the mutual-exclusion problem by Harris Hyman that appeared in a letter (i.e., a non-refereed article) to the Communications of the ACM. Unfortunately it does not work.
 - (a) Explain which rule(s) of the critical-section game is broken and the steps resulting in failure.
 - (b) Compile with flags `-multi -nodebug` and run multiple times. Add flag `-O2` and run multiple times. Explain why the broken rule(s) can take a long time to cause a failure or no failure at all during a large test.
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 $C \in \{1 \dots G\}$ people, 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.:

```

#include <iostream>
#include <string>
#include <sstream>
using namespace std;
#include <malloc.h>

static intmax_t convert( const char * str ); // copy from https://student.cs.uwaterloo.ca/~cs343/examples/uIO.cc

int main( int argc, char * argv[] ) {
    intmax_t times = 1'000'000, size = 5; // defaults
    bool nosummary = getenv( "NOSUMMARY" ); // print summary ?

    try {
        switch ( argc ) {
            case 3: size = convert( argv[2] ); if ( size <= 0 ) throw 1;
            case 2: times = convert( 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

    enum { C = 1'234'567'890 }; // print multiple characters

    #if defined( ARRAY )
    struct S { long int i, j, k, l; };
    S buf[size]; // internal-data buffer
    #elif defined( STRING )
    string strbuf; // external-data buffer
    #elif defined( STRSTREAM )
    stringstream ssbuf; // external-data buffer
    #else
    #error unknown buffering style
    #endif

    for ( int i = 0; i < times; i += 1 ) {
        #if defined( ARRAY )
        for ( int i = 0; i < size; i += 1 ) buf[i] = (S){ C - i, C + i, C | i, C ^ i };
        #elif defined( STRING )
        for ( int i = 0; i < size; i += 1 ) strbuf += to_string(C - i) + '\t' + to_string(C + i) + '\t'
            + to_string(C | i) + '\t' + to_string(C ^ i) + '\t';
            // reset string
        strbuf.clear();
        #elif defined( STRSTREAM )
        for ( int i = 0; i < size; i += 1 ) ssbuf << (C - i) << '\t' << (C + i) << '\t'
            << (C | i) << '\t' << (C ^ i) << '\t';
            // reset stream
        ssbuf.seekp( 0 );
        #else
        #error unknown buffering style
        #endif
    } // for

    if ( ! nosummary ) { malloc_stats(); }
}

```

Figure 1: Internal versus External Buffering

```

void CriticalSection() {
    static volatile uBaseTask * curr;           // shared
    curr = &uThisTask();
    for ( volatile unsigned int i = 0; i < 100; i += 1 ) { // work
        if ( curr != &uThisTask() ) { abort( "interference" ); }
    } // for
}

volatile bool intent[2] = { false, false };    // shared between tasks
volatile int turn = 0;

_Task Hyman {
    int me, you;
    void main() {
        for ( unsigned int i = 0; i < 10000000; i += 1 ) {
1           intent[me] = true;                  // entry protocol
2           while ( turn != me ) {
3               while ( intent[you] ) {}
4               turn = me;
            } // while
5           CriticalSection();
6           intent[me] = false;                  // exit protocol
        } // for
    }
public:
    Hyman( int me, int you ) : me(me), you(you) {}
};
int main() {
    uProcessor p;                               // processor per task
    Hyman h0( 0, 1 ), h1( 1, 0 );
}

```

Figure 2: Hyman Mutual Exclusion

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

During voting, a tourist blocks until all C votes of the current group 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 tours of size C_1, C_2, \dots, C_n may not evenly sum to the number of tourists, resulting in a *quorum* failure when the remaining tourists is less than G .

Implement a general vote-tallier as a class using *only*:

- a single `uOwnerLock` and two `uCondLocks`, plus a signalling flag, and implement using *barging avoidance*. Warning: solutions with only one `uCondLock` are highly unlikely to work. (Hint: `uCondLock::signal` returns true if a task is unblocked and false otherwise.)
- `uSemaphores`, used as binary not counting, to provide mutual exclusion and synchronization, and implement using *barging prevention*.
- 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 unbounded 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:

```

#if defined( MC ) // mutex/condition solution
#include "BargingCheckVote.h"
// includes for this kind of vote-tallier
class TallyVotes {
    // private declarations for this kind of vote-tallier
#elif defined( SEM ) // semaphore solution
#include "BargingCheckVote.h"
// 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 {};
        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( MC ) || defined( BAR )
                unsigned int id
            #endif
        );
        unsigned int getGroupSize(); // generates size of next tour group and prints in ' TG'
    };
};

```

Figure 3: Tally Vote Interfaces

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

At creation, a vote-tallier is passed the number of voters, maximum 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 all votes from the current group 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 current group-size changes dynamically for each tour in the range `prng(1, max tour group size)`. Before each group is formed, a *single* call to `getGroupSize` is made to get the current group-size. Calling `getGroupSize` more than once per group causes incorrect behaviour. 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 maximum group-size and these voters attempt to form a group, i.e., there are not enough voters to cover the worst-case random group-size from `getGroupSize`. (Note, the maximum group-size is used to determine quorum failure rather than the current group-size to simplify termination logic.) 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 example, with the mutex/condition and barrier lock, a voter calling `done` in the failure case may have to block (pretend to be a barger) to force waiting voters to unblock.

Figure 4 shows the macro placement that *must* be present only in the MC and SEM tally-votes implementa-

```

#include "BargingCheckVote.h"
class TallyVotes {
    ... // regular declarations
    BCHECK_DECL;
public:
    ... // regular declarations
    Tour vote( unsigned int id __attribute__(( unused )), Ballot ballot ) {
        // acquire mutual exclusion
        VOTER_ENTER;
        ... // voting code
        VOTER_LEAVE;
        // release mutual exclusion
        // return majority vote
    }
    unsigned int getGroupSize() {
        // generate group size
        NEW_GROUP( next_group_size );
        // return group size
    }
};

```

Figure 4: Barging Check Macros: MC and SEM

```

_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',
        NextTourSize = 'N', 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

tion to test for barging, and defining preprocessor variable BARGINGCHECK triggers barging testing (see 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.

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
- 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). (For now, treat **_Monitor** as a class and **_Cormonitor** as a coroutine 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

```

_Monitor / _Cormonitor Printer {    // chose one of the two kinds of type constructor
public:
    Printer( unsigned int voters );
    void print( Voter::States state, unsigned int nextGroupSize );
    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
N n	next tour group is size n (printed in the TG column)
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 in done (MC/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)

Figure 7: Voter Status Entries

flushed and storing starts again. Output must look like that in Figure 8.

The first column is assigned to the tour guide with the title "TG" and prints the size of the next group after the size is determined. Each column afterwards 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. 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 the left-hand example of Figure 8, there are 3 voters, a maximum of 3 voters in a group, and each voter attempts to vote once. At line 4, TG has the value "N 3" in its buffer slot, V0 has the value "S", and V2, and V1 are empty. When V0 attempts to print "V 2,1,0", which overwrites its current buffer value of "S", the buffer must be flushed generating line 4. V0's new value of "V 2,1,0" is then inserted into its buffer slot. When V0 attempts to print "B 1", which overwrites its current buffer value of "V 2,1,0", 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 "B 1" and V1 inserts value "S" into the buffer. When V1 attempts to print "V 2,1,0", which overwrites its current buffer value of "S", the buffer must be flushed generating line 6, and so on.

For example, in the right-hand example of Figure 8, there are 6 voters, a maximum of 3 voters in a group, and each voter attempts to vote twice. Voters V2 and V4 are delayed (lines 17 and 23), e.g., they went to Tom's for a coffee and donut. The TG entries show V3 and V5 vote together (group 1, lines 13 and 15), V0 and V1 vote together (group 2, lines 11 and 16), V5 votes alone (group 3, line 19), and V0, V2, V3 vote together (group 4, lines 22, 25, 27), and V1/V2 vote alone (groups 5, line 28, and 6, line 26). Hence, all voters have voted twice and terminated, except V4, which has not voted at all, so it fails with X (line 29).

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

1	\$ vote 3 3 1
2	TG V0 V1 V2
3	*****
4	N 3 S
5	V 2,1,0
6	B 1 S
7	V 2,1,0
8	B 2 S
9	V 0,2,1
10	C p
11	G p 1
12	T
13	G p 1
14	T
15	*****
16	All tours ended

1	\$ vote 6 3 2
2	TG V0 V1 V2 V3 V4 V5
3	*****
4	N 2
5	S
6	S
7	S
8	N 2 V 1,2,0
9	B 1 V 1,2,0
10	N 1 U 0 C s
11	G s 2
12	S
13	
14	
15	N 3
16	G s 2
17	V 1,0,2 S
18	B 1
19	S
20	V 2,0,1 B 2
21	N 1 U 1 G g 4
22	C g
23	U 0 V 2,0,1 S
24	N 1 C p V 1,2,0
25	N 2 G g 4 C s
26	T
27	G p 5 T
28	G g 4 T
29	X
30	T
31	*****
32	All tours ended

Figure 8: Voters: Example Output

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

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 maximum 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 repeatable results, all random numbers are generated using the `μC++` prng functions, which requires including `uPRNG.h` (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 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 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 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.
- iii. As the kernel threads increase, very briefly speculate on any performance difference.

Use the following to elide output:

```
#ifndef NOOUTPUT
#define PRINT( stmt )
#else
#define PRINT( stmt ) stmt
#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, 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. q2.txt – contains the information required by question [2, p. 1](#).
3. BargingCheckVote.h – barging checker (provided)
4. q3tallyVotes.h, q3*.h, cc, C, cpp – code for question [3a, p. 1](#). **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. 1](#) by inserting the object-file names matching your source-file names.

```
VIMPL:=MC
OUTPUT:=OUTPUT
BCHECK:=NOBARGINGCHECK

CXX = u++                                # compiler
CXXFLAGS = -g -multi -O2 -Wall -Wextra -MMD -D"${VIMPL}" -D"${OUTPUT}" \
          -D"${BCHECK}" # 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 VotImpl

ifeq ($(shell if [ "${LOCKVIMPL}" = "${VIMPL}" -a "${OUTPUTTYPE}" = "${OUTPUT}" -a \
    "${BCHECKIMPL}" = "${BCHECK}" ] ; then echo true ; fi ),true)
${EXEC} : ${OBJECTS}
    ${CXX} ${CXXFLAGS} $^ -o $@
else
.PHONY : ${EXEC}
${EXEC} :
    rm -f VotImpl
    touch q3tallyVotes.h
    ${MAKE} ${EXEC} VIMPL="${VIMPL}" OUTPUT="${OUTPUT}" BCHECK="${BCHECK}"
endif

VotImpl :
    echo "LOCKVIMPL=${VIMPL}\nOUTPUTTYPE=${OUTPUT}\nBCHECKIMPL=${BCHECK}" > VotImpl
    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} VotImpl

```

This makefile is invoked as follows:

```

$ make vote VIMPL=MC BCHECK=BARGINGCHECK
$ vote ...
$ make vote VIMPL=SEM OUTPUT=OUTPUT
$ vote ...
$ make vote VIMPL=BAR 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!