This assignment introduces complex locks in \( \mu \text{C++} \) 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. (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, e.g.:

\[
\begin{align*}
\text{P S G} \\
\text{voter1} & \ 0 \ 1 \ 2 \\
\text{voter2} & \ 2 \ 1 \ 0 \\
\text{tally} & \ 2 \ 2 \ 2 \\
\end{align*}
\]

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.

The number of tourists should be evenly divisible by the tour-group size so all groups contain the same number of tourists. 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 quickly while another executes slowly, so the quick 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.

Implement a vote-tallier for \( G \)-way voting as a class using only:

i. an \texttt{uOwnerLock} and \texttt{uCondLocks} to provide mutual exclusion and synchronization.

ii. \texttt{uSemaphores} used as binary rather than counting to provide mutual exclusion and synchronization.

iii. a single \texttt{uBarrier} to provide mutual exclusion and synchronization. Note, a \texttt{uBarrier} has implicit mutual exclusion so it is only necessary to manage the synchronization. As well, only the basic aspects of the \texttt{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 prevented/avoided.

Figure 1 shows the different forms for each \( \mu \text{C++} \) 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:

\[
\texttt{u++ -DSEM -c TallyVotesSEM.cc}
\]

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 \texttt{vote} method with their id and a ranked vote, indicating their desire for a picture, statue, or gift-shop tour. The \texttt{vote} routine does not return until \texttt{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 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 calls `done()
TallyVotes detects a quorum failure 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 raise exception `Failed`, and any waiting voters are unblocked and raise exception `Failed`. For the owner/condition lock, a voter calling `done` in the failure case is 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, must block on the barrier to force waiting voters to unblock.

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

```cpp
_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', Terminated = 'T' };
        Voter( unsigned int id, unsigned int nvotes, TallyVotes & voteTallier, Printer & printer );
};
```

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 finish 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. The interface for
the printer is (you may add only a public destructor and private members).

```cpp
class _Monitor / _Cormonitor

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, unsigned int numBlocked );

```

(Monitors are discussed shortly, and are classes with public methods that implicitly provide mutual exclu-
sion.) 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 2.

Each column is assigned to a voter with an appropriate title, “\(V_i\),” and a column entry indicates its current
status:

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>starting</td>
</tr>
<tr>
<td>V p, s, g</td>
<td>voting with ballot containing 3 rankings</td>
</tr>
<tr>
<td>B n</td>
<td>blocking during voting, (n) voters waiting (including self)</td>
</tr>
<tr>
<td>U n</td>
<td>unlocking after group reached, (n) voters still waiting (not including self)</td>
</tr>
<tr>
<td>b</td>
<td>barging into voter and having to wait for signalled tasks</td>
</tr>
<tr>
<td>C</td>
<td>group is complete and voting result is computed</td>
</tr>
</tbody>
</table>

Figure 2: Voters: Example Output
State | Meaning
--- | ---
F t | finished voting and selected tour is t (p/s/g)
X | failed to form a group
T | voter terminates (task main ends)

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 2, 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 2, 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, V1 V2 vote together, and V2, V4, V5 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 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’ ] ] ] ]
```

- **voters** is the size of a tour (> 0), i.e., the number of voters (tasks) to be started (multiple of group). 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, short 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) i. Compare the performance among the three 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
```

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 and as kernel threads increase.

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. MPRNG.h – random number generator (provided)
2. q1tallyVotes.h, q1*.h,*,.cc,.c,.cpp – code for question question 1a, p. 1. Program documentation must be present in your submitted code. No user, system or test documentation is to be submitted for this question.
3. q1*.txt – contains the information required by question 1b.
4. Modify the following `Makefile` to compile the programs for question 1a, p. 1 by inserting the object-file names matching your source-file names.

```
LIMPL:=MC
OUTPUT:=OUTPUT
CXX = u++  # compiler
CXXFLAGS = -g -mult -O2 -std=c++11 -Wall -Wextra -MMD -D$(LIMPL) -D$(OUTPUT) # compiler flags
MAKEFILE_NAME = $(firstword $(MAKEFILE_LIST)) # makefile name
OBJECTS = q1tallyVotes$(LIMPL).o # list of object files for question 1 prefixed with “q1”
EXEC = vote
DEPENDS = $(OBJECTS:.o=.d) # substitute “.o” with “.d”

.PHONY : all clean
all : $(EXEC) # build all executables
    -include LockImpl
ifeq ($(LOCKIMPL),$(LIMPL)) # same implementation type as last time ?
    $(EXEC) : $(OBJECTS)
        $(CXX) $(CXXFLAGS) $^ -o @$
else
ifeq ($(LIMPL),) # no implementation type specified ?
    # set type to previous type
    LIMPL=$(LOCKIMPL)
    $(EXEC) : $(OBJECTS)
        $(CXX) $(CXXFLAGS) $^ -o @$
```
else
  .PHONY : $(EXEC)
  # implementation type has changed
  $(EXEC) :
    rm -f LockImpl
    touch q1tallyVotes.h
    sleep 1
    $(MAKE) $(EXEC) LIMPL="${LIMPL}"
endif
endif

LockImpl :
  echo "LOCKIMPL=${LIMPL}" > LockImpl
  sleep 1

###################################################
# OPTIONAL : changes to this file => recompile
$(OBJECTS) : $(MAKEFILE_NAME)

-include $(DEPENDS)

# include *.d files containing program dependences
clean :
  rm -f *.d *.o $(EXEC) LockImpl

# remove files that can be regenerated

This makefile is invoked as follows:

$ make vote LIMPL=MC
$ vote ...
$ make vote LIMPL=SEM
$ vote ...
$ make vote LIMPL=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!