CS 343 Winter 2024 – Assignment 4
Instructor: Colby Parsons
Due Date: Monday, March 11, 2024 at 22:00
Late Date: Wednesday, March 13, 2024 at 22:00

March 10, 2024

This assignment introduces complex locks in μC++ 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 100000000 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).
   - Start with the command-line argument 100000000 20 and adjust the times amount (if necessary) to get program execution into the range 1 to 100 seconds for the 3 versions of the program. (Timing results below 1 second are inaccurate.) Otherwise, increase/decrease the times amount as necessary and scale the difference in the answer.
   - Run 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 Dekker solution to the mutual exclusion problem. **If you run this program, do not compile with optimization.**

   (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(s) is unlikely to cause a failure even during a large test.

   (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) The covered (kissing) bridge in West Montrose Ontario is only one-lane wide. Hence, cars (and buggies) crossing the bridge from the north and south must take turns. As well, the bridge is old and made of wood so there is a limit of 3 cars simultaneously crossing the bridge in one direction.

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

int main( int argc, char * argv[] ) {
    intmax_t times = 1'000'000, size = 20;              // defaults
    char * nosummary = getenv( "NOSUMMARY" );           // print summary ?
    struct cmd_error {};                                // command-line errors

    try {
        switch ( argc ) {
          case 3: size = convert( argv[2] ); if ( size <= 0 ) { throw cmd_error(); }
          case 2: times = convert( argv[1] ); if ( times <= 0 ) { throw cmd_error(); }
          case 1: break;                                // use defaults
          default: throw cmd_error();
        } // switch
    } catch( ... ) {
        cerr << "Usage: " << argv[0] << " [ times (> 0) [ size (> 0) ] ]" << endl;
        exit( EXIT_FAILURE );
    } // 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 ( volatile int i = 0; i < size; i += 1 ) buf[i] = (S){ C - i, C + i, C | i, C ^ i };
        #elif defined( STRING )
        for ( volatile 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';
        strbuf.clear();                                 // reset string
        #elif defined( STRSTREAM )
        for ( volatile int i = 0; i < size; i += 1 ) ssbuf << (C - i) << '\t' << (C + i) << '\t'
                                                   << (C | i) << '\t' << (C ^ i) << '\t';
        ssbuf.seekp( 0 );                               // reset stream
        #else
            #error unknown buffering style
        #endif
    } // for

    if ( ! nosummary ) { malloc_stats(); }              // print heap statistics
}
```

Figure 1: Internal versus External Buffering

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

enum Intent { WantIn, DontWantIn } * Last;                      // shared
_Task Dekker {
    Intent & me, & you;
    void main() {
        for ( unsigned int i = 0; i < 10'000'000; i += 1 ) {
1           for ( ;; ) {                                        // entry protocol, high priority
2               me = WantIn;
                __asm__ __volatile__( "mfence" );               // prevent hardware reordering (x86)
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() {
    uProcessor p;
    Intent me = DontWantIn, you = DontWantIn;                   // shared
    ::Last = &me;                                               // arbitrary who starts as last
    Dekker t0( me, you ), t1( you, me );
}
```

Figure 2: Dekker 2-Thread Mutual Exclusion

Figure 3 shows the bridge interface. Implement any necessary **private** declarations/members and the **public** member Bridge::cross. A car calls the cross member with the direction of travel for crossing the bridge. Internally, member cross blocks the invoking car task until it is appropriate for it to cross the bridge.

When member cross returns, the car has crossed the bridge in the appropriate direction. Implement the bridge using:

i. a single uOwnerLock and a single uCondLocks to provide mutual exclusion and synchronization plus a signalling flag, and implement using ***barging avoidance***.

ii. 3 uSemaphores, used as binary not counting, to provide mutual exclusion and synchronization, and implement using ***barging prevention***

Your solution must **NOT** have starvation, staleness or freshness, and it **MUST** service arriving cars in temporal order. As well, only 3 cars can be simultaneously on the bridge. The private Bridge member routines startNorth, startSouth, endNorth, and endSouth are to be called in the member cross to enter and exit the bridge in a given direction.

Figure 4 shows the given macros BCHECK_DECL, BRIDGE_ENTRY_START and BRIDGE_ENTRY_END and their placement in class Bridge and members startNorth/startSouth. These macros must be present in the locking solutions MC and SEM to test for barging. If barging is detected, a macro prints a message and the program continues, possibly printing more barging messages. Barging checking can be toggled on or off during compilation using the preprocessor variable BARGINGCHECK **(see the Makefile)**. To inspect the program with gdb when barging is detected, set BARGINGCHECK=0 to abort the program.

```
#include "BargingCheckBridge.h"
#if defined( MC )                                   // external scheduling monitor solution
class Bridge {
    // YOU ADD DECLARATIONS AND MEMBERS
#elif defined( SEM )                                // internal scheduling monitor solution
class Bridge {
    // YOU ADD DECLARATIONS AND MEMBERS
#else
    #error unsupported bridge type
#endif
  private:
    BCHECK_DECL;
    void startNorth( unsigned int id );
    void startSouth( unsigned int id );
    void endNorth( unsigned int id );
    void endSouth( unsigned int id );
    // YOU ADD COMMON PRIVATE DECLARATIONS AND MEMBERS
  public:                                           // common public interface
    enum Direction { NORTH = 'N', SOUTH = 'S' };
    void cross( unsigned int id, Direction dir ) {
        // YOU WRITE THIS CODE
    } // cross
}; // Bridge
```

Figure 3: Bridge Interface

```
#include "BargingCheckBridge.h"
class Bridge {
    ...                                             // regular declarations
    BCHECK_DECL;
    ...                                             // regular declarations
    void startNorth( unsigned int id ) {        // also add macros to startSouth
        // acquire mutual exclusion
        BRIDGE_ENTRY_START;
        ...                                         // bridge code
        BRIDGE_ENTRY_END;
        // release mutual exclusion
    }
};
```

Figure 4: Barging Check Macros: MC and SEM

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

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

and then repeatedly performs the following steps crossings times:

- yield a random number of times, between 0 and 9 inclusive
- crosses in a random direction

After a car crosses the requisite number of times, it prints a done message and terminates. Crossing the bridge is accomplished by calling bridge member cross. 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 car 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. Figure Section 7, p. 6 shows the sample output for a run with 3 cars, each making 3 trips across the bridge. See additional output from the sample executable.

```
_Task Car {
    const int id;
    const int crossings;
    Bridge & bridge;
    Printer & printer;
    // YOU ADD DECLARATIONS AND MEMBERS
    void main();
  public:
    enum States : char { Start = 'S', Block = 'B', Unblock = 'U',
        Done = 'D', Crossed = 'C', Going = 'G' };
};
```

Figure 5: Car Interface

```
_Monitor / _Cormonitor Printer {        // chose one of the two kinds of type constructor
  public:
    Printer( unsigned int cars );
    void print( unsigned int id, Car::States state );
    void print( unsigned int id, Car::States state, Bridge::Direction dir, int datum );
};
```

Figure 6: Printer Interface

Each column is assigned to a car with the titles, "C_i", and Figure 8 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 bridge and/or a car task (you decide where to print).

As a car crosses the bridge in member cross it must do the following.

- print when a car is blocked, with the direction it is headed, and how many cars are currently blocked.
- print when a car is unblocked, with the direction it is headed, and how many cars remain blocked
- print which car is going across, in what direction, and the number of cars currently crossing.
- yield 3 times to simulate the crossing.
- assert there are no cars crossing in the opposite direction.
- assert there are 3 or fewer cars currently crossing.
- print which car is done crossing, in what direction, and the number of cars still crossing.

You may print any of the output from cross in the subroutines that it calls, such as startNorth, startSouth, endNorth, and endSouth.

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

```
bridge [ cars | 'd' [ crossings | 'd' [ seed | 'd' [ processors | 'd' ] ] ] ]
```

cars is the number of cars attempting to cross the bridge (> 0). If d or no value for cars is specified, assume 10.

crossings is the number of times each car crosses the bridge in a random direction (> 0). If d or no value for crossings is specified, assume 5.

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.

```
$ bridge 3 3
C0      C1      C2
*******  *******  *******
S       S
G S 1
C S 0   G N 1   S
        C N 0
        G S 1
B N 1   C S 0   G S 1
U N 0           C S 0
G N 1   G N 2   B S 1
C N 1   C N 0
        D       U S 0
                G S 1
                C S 0
                G N 1
                C N 0
G N 1           D
C N 0
D
*****************
All crossings ended
```

Figure 7: Example Output. See sample executables output for better alignment.

State	Meaning
S	start
G dir, n	going across the bridge in dir direction (N/S) and with n cars on the bridge (including self)
C dir, n	completed crossing in dir direction (N/S) and with n cars still on the bridge (not including self)
B dir, n	block before crossing, n cars waiting (including self)
U dir, n	unblock once ready to cross, n cars still waiting (not including self)
D	car is done all crossings and terminates

Figure 8: Car Status Entries

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
#definCarNT( stmt ) stmt
#endif // NOOUTPUT
PRINT( printer.print( id, direction ) ); // elide printer call
```

i. Compare the performance between the 2 kinds of locks:

- Time the executions using the time command with output and barging checking turned off:

```
$ /usr/bin/time -f "%Uu %Ss %Er %Mkb" bridge 200 4000 1003 2
20.94u 0.03s 0:10.47r 7304kb
```

Output from time differs depending on the shell, so use the system time command. Compare the *user* (20.94u) and *real* (0:10.47r) 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 cars and then crossings to get real time in range 1 to 100

seconds. (Timing results below 1 second are inaccurate.) Use the same number of cars for all experiments.
- Include both MC and SEM timing results to validate your experiments.
- Repeat the experiment using 2 processors and include the MC and SEM timing results to validate your experiments.

ii. State the performance difference (larger/smaller/by how much) between the locks.

iii. Very briefly speculate on the performance difference between the MC and SEM implementations when using 2 processors.

iv. 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. q2*.txt – contains the information required by question 2, p. 1.
3. BargingCheckBridge.h – barging checker (provided)
4. q3bridge.h, q3*.{h,cc,C,cpp} – code for question 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.

```
BIMPL:=MC
OUTPUT:=OUTPUT
BCHECK:=BARGINGCHECK

CXX = u++                                       # compiler
CXXFLAGS = -quiet -g -nodebug -multi -O2 -Wall -Wextra -MMD \
        -D"${BIMPL}" -D"${OUTPUT}" -D"${BCHECK}" # compiler flags
MAKEFLAGS = --no-print-directory
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}} # makefile name

OBJECTS = ... q3bridge${BIMPL}.o                # list of object files for question 3 prefixed with "q3"
DEPENDS = ${OBJECTS:.o=.d}                      # substitute ".o" with ".d"
EXEC = bridge

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

.ONESHELL :
.PHONY : all clean

all : ${EXEC}                                   # build all executables

-include LockImpl

# same implementation concurrency/type as last time ?
ifeq (${shell if [ "${LOCKBIMPL}" = "${BIMPL}" -a "${OUTPUTTYPE}" = "${OUTPUT}" \
        -a "${BCHECKIMPL}" = "${BCHECK}" ] ; \
        then echo true ; fi },true)
${EXEC} : ${OBJECTS}
    ${CXX} ${CXXFLAGS} $^ -o $@
else                                            # implementation type has changed => rebuilt
```

```
.PHONY : ${EXEC}
${EXEC} :
    rm -f LockImpl
    touch q3bridge.h
    ${MAKE} ${EXEC} BIMPL="${BIMPL}" OUTPUT="${OUTPUT}" BCHECK="${BCHECK}"
endif

LockImpl :
    echo "LOCKBIMPL=${BIMPL}\nOUTPUTTYPE=${OUTPUT}\nBCHECKIMPL=${BCHECK}" > LockImpl
    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 *.o ${EXEC} LockImpl *.out
```

This makefile is invoked as follows:

```
$ make bridge BIMPL=MC BCHECK=BARGINGCHECK
$ bridge ...
$ make bridge BIMPL=SEM OUTPUT=OUTPUT
$ bridge ...
```

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!