CS 343 Winter 2024 – Assignment 5
Instructor: Colby Parsons
Due Date: Monday, March 25, 2024 at 22:00
Late Date: Wednesday, March 27, 2024 at 22:00

March 22, 2024

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. Watch the video clip http://www.youtube.com/watch?v=ByPrDPbdRhc from the Dr. Who episode "Blink". **Warning: it is scary but not violent.** 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.)

2. (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.

   Implement the bridge using:

   i. μC++ monitor using external scheduling,
   ii. μC++ monitor using internal scheduling with 1 uCondition,
   iii. μC++ nested monitors and internal scheduling, with 1 uCondition,
   iv. 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. Figure 1 shows a μC++ simulation of Java barging by replacing the normal calls to condition-variable wait and signal with these calls. This code randomly accepts calls to the interface routines, if a caller exists. Only condition variable bench may be used and it may only be accessed via member routines wait() and signalAll(). Hint: to control barging tasks, use a ticket counter.
   v. μC++ monitor that simulates a general automatic-signal monitor. Hint: to control barging tasks, use a ticket counter (this solution and the Java monitor solution will be very similar). 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 EXIT() ...
```

```
void Bridge::wait() {
        bench.wait();                                          // wait until signalled
        while ( rand() % 2 == 0 ) {                            // multiple bargers allowed
                try {
                        _Accept( startNorth || endNorth || startSouth || endSouth ) {          // accept barging callers
                        } _Else {                                                  // do not wait if no callers
                        } // _Accept
                } catch( uMutexFailure::RendezvousFailure & ) {}
        } // while
}

void Bridge::signalAll() {                          // also useful
        while ( ! bench.empty() ) bench.signal();  // drain the condition
}
```

Figure 1: Java Simulation

```
_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;
        EXIT();
    }

    int remove() {
        WAITUNTIL( count > 0, , );         // empty before/after
        int elem = Elements[front];
        front = ( front + 1 ) % 20;
        count -= 1;
        EXIT();
        return elem;                       // return value
    }
};
```

Figure 2: Automatic signal monitor

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, the expression before is executed before the wait and the expression after is executed after the wait. Macro EXIT must be called on return from a public routine of an automatic-signal monitor. Figure 2 shows a bounded buffer implemented as an automatic-signal monitor.

Make absolutely sure to *always* execute the EXIT() macro at the end of each mutex member, either normal or exceptional return. As well, the macros must be abstract (hidden), 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.

vi. μC++ server task performing the maximum amount of work on behalf of the client (i.e., very little code in members startNorth/South and endNorth/South). The output for this implementation differs from the monitor output because all cars print blocking and unblocking messages, and the server task prints all Car statuses except for start and done (S/D). You will likely create a solution using a list of futures. 5 bonus marks will be awarded to solutions that do not use a list and instead use the tricker approach of only using 2 futures to avoid dynamic allocation. If you attempt to use only two futures be careful, since it is very easy to inadvertently introduce temporal barging with this approach.

Figure 3 shows the bridge interface. Implement any necessary **private** declarations/members and the **public** members that are provided. Hint: many solutions will have repeated code that can be cleanly abstracted into private helper functions. 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. No unbounded busy-waiting is allowed in any solution, and barging tasks and must be avoided/prevented.

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

```
#pragma once
#include "BargingCheckBridge.h"
#if defined( INT ) // internal scheduling monitor solution
_Monitor Bridge {
  private: // YOU ADD DECLARATIONS AND MEMBERS
#elif defined( EXT ) // external scheduling monitor solution
_Monitor Bridge {
  private: // YOU ADD DECLARATIONS AND MEMBERS
#elif defined( AUTO ) // automatic-signal monitor solution
#include "AutomaticSignal.h"
_Monitor Bridge {
  private: // YOU ADD DECLARATIONS AND MEMBERS
#elif defined( NEST ) // nested monitor solution
_Monitor Bridge {
  public:
    _Monitor BridgeNested {
      private: // YOU ADD DECLARATIONS AND MEMBERS
      public:
        void startNorth( unsigned int id __attribute__(( unused )) );
        void startSouth( unsigned int id __attribute__(( unused )) );
        void endNorth( unsigned int id __attribute__(( unused )) );
        void endSouth( unsigned int id __attribute__(( unused )) );
    };
  private: // YOU ADD DECLARATIONS AND MEMBERS
#elif defined( INTB ) // internal scheduling monitor solution with barging
_Monitor Bridge {
  private: // YOU ADD DECLARATIONS AND MEMBERS
    uCondition bench; // only bench for tasks to block on
    void wait(); // barging version of wait
    void signalAll();
#elif defined( TASK ) // task solution
_Task Bridge {
  private:
    void main();
    // YOU ADD DECLARATIONS AND MEMBERS
#else
    #error unsupported bridge type
#endif
#if defined( NEST )
    _Nomutex void endNorth( unsigned int id __attribute__(( unused )) );
    _Nomutex void endSouth( unsigned int id __attribute__(( unused )) );
#else
    _Mutex void endNorth( unsigned int id __attribute__(( unused )) );
    _Mutex void endSouth( unsigned int id __attribute__(( unused )) );
#endif
#if defined( TASK )
    _Mutex FTicket & startNorth( unsigned int id __attribute__(( unused )));
    _Mutex FTicket & startSouth( unsigned int id __attribute__(( unused )) );
  public:
    enum Direction { NORTH = 'N', SOUTH = 'S' };
    _Nomutex void cross( unsigned int id __attribute__(( unused )), Direction dir ) {
        // YOU WRITE THIS CODE
    } // cross
#else
    _Mutex void startNorth( unsigned int id __attribute__(( unused )) );
    _Mutex void startSouth( unsigned int id __attribute__(( unused )) );
  public:                                                    // common interface
    enum Direction { NORTH = 'N', SOUTH = 'S' };
    _Nomutex void cross( unsigned int id __attribute__(( unused )), Direction dir ) {
        // YOU WRITE THIS CODE
    } // cross
#endif
}; // 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

```
_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

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 all solutions except for TASK 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.

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 / _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

```
$ 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

_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 shows the sample output for a run with 3 cars, each making 3 trips across the bridge. See additional output from the sample executable.

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.

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

i. Compare the performance between the 6 implementations:

- 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 1
  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 all 6 timing results to validate your experiments.
- Repeat the experiment using 2 processors and include all 6 timing results to validate your experiments.

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

iii. Very briefly speculate on the performance difference between the 6 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. BargingCheckBridge.h – barging checker (provided)
3. AutomaticSignal.h, q2bridge.h, q2*.{h,cc,C,cpp} – code for question question 2a, p. 1. **Program documentation must be present in your submitted code. No user, system or test documentation is to be submitted for this question.**
4. q2*.txt – contains the information required by question 2b.
5. Makefile – construct a makefile **similar to those presented in the course** to compile the program for question 2a, p. 1. **This makefile must NOT contain hand-coded dependencies.** The makefile is invoked as follows:

```
$ make bridge BIMPL=EXT
$ bridge ...
$ make bridge BIMPL=INT
$ bridge ...
$ make bridge BIMPL=INTB
$ bridge ...
$ make bridge BIMPL=AUTO OUTPUT=OUTPUT
$ bridge ...
$ make bridge BIMPL=TASK OUTPUT=NOOUTPUT
$ bridge ...
$ make bridge BIMPL=NEST
$ 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!