CS 343 Fall 2012 – Assignment 3
Instructor: Martin Karsten
Due Date: Monday, Oct 22, 2012 at 22:00
Late Date: Wednesday, Oct 24, 2012 at 22:00

October 11, 2012

This assignment introduces locks in μC++ and examines 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 have *only* public constructors and/or destructors; no other public members are allowed.)**

1. (a) Implement a generalized FIFO bounded-buffer for a producer/consumer problem with the following interface (you may add only a public destructor and private members):

```
template<typename T> class BoundedBuffer {
  public:
    BoundedBuffer( const unsigned int size = 10 );
    void insert( T elem );
    T remove();
};
```

which creates a bounded buffer of size size, and supports multiple producers and consumers. You may *only* use uCondLock and uOwnerLock to implement the necessary synchronization and mutual exclusion needed by the bounded buffer.

Implement the BoundedBuffer in the following ways:

i. Use busy waiting when waiting for buffer entries to become free or empty. In this approach, new tasks may barge into the buffer taking free or empty entries from tasks that have been signalled to access these entries. This implementation uses two condition-locks, one for each of the waiting producer and consumer tasks. The reason there is barging in this solution is that uCondLock::wait re-acquires its argument owner-lock before returning. Now once the owner-lock is released by a task exiting insert or remove, there is a race to acquire the lock by a new task calling insert/remove and by a signalled task. If the calling task wins the race, it barges ahead of any signalled task. So the state of the buffer at the time of the signal is not the same as the time the signalled task re-acquires the argument owner-lock, because the barging task changes the buffer. Hence, the signalled task may have to wait again, and there is no bound on how many times this may happen (possible long-term starvation).

ii. Use no busy waiting when waiting for buffer entries to become free or empty. In this approach, new (barging) tasks must be prevented from taking free or empty entries if tasks have been unblocked to access these entries. This implementation uses three condition-locks, one for each of the waiting producer, consumer and barging tasks (*and has no looping*). To prevent barging, create a flag variable to indicate when signalling is occurring; tasks entering the monitor check the flag to determine if they are barging because another task has been signalled. Set the flag before signalling a producer/consumer task; the flag is reset only if there are no waiting tasks. If a task enters the buffer before a signal task has restarted (while the flag is set), it waits on the barging condition-lock. If a task needs to wait on either the producer/consumer condition-locks and the barging condition-lock is not empty, signal a task from it. Before exiting from the buffer and not signalling a producer/consumer, if the barging condition-lock is not empty, signal a task from it. Try to minimize the number of sets/resets of the signal flag.

Before inserting or removing an item to/from the buffer, perform an assert that checks if the buffer is not full or not empty, respectively. The buffer implementations are defined in a single file denoted in the following way:

```
#ifdef BUSY                          // busy waiting implementation
// implementation
#endif // BUSY

#ifdef NOBUSY                        // no busy waiting implementation
// implementation
#endif // NOBUSY
```

Test the bounded buffer with a number of producers and consumers. (Assume the buffer-element type, T, is **int** and the sentinel value is -1.) The producer interface is:

```
_Task Producer {
    void main();
  public:
    Producer( BoundedBuffer<int> &buffer, const int Produce, const int Delay );
};
```

The producer generates Produce integers, from 1 to Produce inclusive, and inserts them into buffer. Before producing an item, a producer randomly yields between 0 and Delay-1 times. Yielding is accomplished by calling yield(times) to give up a task's CPU time-slice a number of times. The consumer interface is:

```
_Task Consumer {
    void main();
  public:
    Consumer( BoundedBuffer<int> &buffer, const int Delay, const int Sentinel, int &sum );
};
```

The consumer removes items from buffer, and terminates when it removes a Sentinel value from the buffer. A consumer sums all the values it removes from buffer (excluding the Sentinel value) and returns this value through the reference variable sum. Before removing an item, a consumer randomly yields between 0 and Delay-1 times.

uMain::main creates the bounded buffer, the producer and consumer tasks. After all the producer tasks have terminated, uMain::main inserts an appropriate number of sentinel values into the buffer to terminate the consumers. The partial sums from each consumer are totalled to produce the sum of all values generated by the producers. Print this total in the following way:

```
total: ddddd
```

The sum must be the same regardless of the order or speed of execution of the producer and consumer tasks.

The shell interface for the boundedBuffer program is:

```
boundedBuffer [ Cons [ Prods [ Produce [ BufferSize [ Delays ] ] ] ] ]
```

(Square brackets indicate optional command line parameters, and do not appear on the actual command line.) Where the meaning of each parameter is:

Cons: positive number of consumers to create. The default value if unspecified is 5.

Prods: positive number of producers to create. The default value if unspecified is 3.

Produce: positive number of items generated by each producer. The default value if unspecified is 10.

BufferSize: positive number of elements in (size of) the bounder buffer. The default value if unspecified is 10.

Delays: positive number of times a producer/consumer yields *before* inserting/removing an item into/from the buffer. The default value if unspecified is Cons + Prods.

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. Use the following monitor to safely generate random values (monitors will be discussed shortly):

```
_Monitor MPRNG {
  public:
    MPRNG( unsigned int seed = 1009 ) { srand( seed ); }      // set seed
    void seed( unsigned int seed ) { srand( seed ); }         // set seed
    unsigned int operator()() { return rand(); }              // [0,UINT_MAX]
    unsigned int operator()( unsigned int u ) { return operator()() % (u + 1); } // [0,u]
    unsigned int operator()( unsigned int l, unsigned int u ) { return operator()( u - l ) + l; } // [l,u]
};
```

(b) i. Compare the busy and non-busy waiting versions of the program with respect to performance by doing the following:

- Time the execution using the time command:

```
% time ./a.out
3.21u 0.02s 0:03.32 100.0%
```

(Output from time differs depending on your shell, but all provide user, system and real time.) 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 50 55 10000 30 10 and adjust the Produce amount (if necessary) to get execution times in the range 0.1 to 100 seconds. (Timing results below 0.1 seconds are inaccurate.) Use the same command-line values for all experiments.
- Run both the experiments again after recompiling the programs with compiler optimization turned on (i.e., compiler flag -O2). Include all 4 timing results to validate your experiments.

ii. State the observed performance difference between busy and nobusy waiting execution, without and with optimization.

iii. Speculate as to the reason for the performance difference between busy and nobusy waiting execution.

iv. Add the following declaration to uMain::main after checking command-line arguments but before creating any tasks:

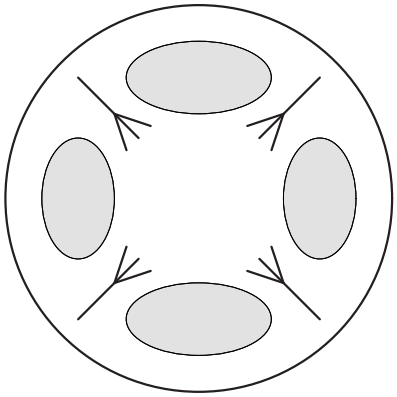
```
#ifdef __U_MULTI__
    uProcessor p[3] __attribute__(( unused )); // create 3 kernel thread for a total of 4
#endif // __U_MULTI__
```

to increase the number of kernel threads to access multiple processors. This declaration must be in the same scope as the declaration of the quicksort task. Compile the program with the -multi flag and no optimization on a multi-core computer with at least 4 CPUs (cores), and run the same experiment as above. Include timing results to validate your experiment.

v. State the observed performance difference between uniprocessor and multiprocessor execution.

vi. Speculate as to the reason for the performance difference between uniprocessor and multiprocessor execution.

2. A group of N (N $>$ 1) philosophers plans to spend an evening together eating and thinking (Hoare, page 55). Unfortunately, the host philosopher has only N forks and is serving a special spaghetti that requires 2 forks to eat. Luckily for the host, philosophers (like university students) are interested more in knowledge than food; after eating a few bites of spaghetti, a philosopher is apt to drop her forks and contemplate the oneness of the universe for a while, until her stomach begins to growl, when she again goes back to eating. So the table is set with N plates of spaghetti with one fork between adjacent plates. For example, the table would look like this for N=4:



Consequently there is one fork on either side of a plate. Before eating, a philosopher must obtain the two forks on either side of the plate; hence, two adjacent philosophers cannot eat simultaneously. The host reasons that, when a philosopher's stomach begins to growl, she can simply wait until the two philosophers on either side begin thinking, then *simultaneously pick up the two forks on either side of her plate* and begin eating. If a philosopher cannot get both forks immediately, then she must wait until both are free. (Imagine what would happen if all the philosophers simultaneously picked up their right forks and then waited until their left forks were available.)

The table manages the forks and must be written as a class using μC++ semaphores to provide mutual exclusion and synchronization. The table implementation has the following interface (you may add only a public destructor and private members):

```
class Table {
    // private declarations for this kind of table
  public:
    Table( unsigned int noOfPhil, Printer &prt );
    void pickup( unsigned int id );
    void putdown( unsigned int id );
};
```

Member routines pickup and putdown are called by each philosopher, passing the philosopher's identifier (value between 0 and $N-1$), to pick up and put down both forks, respectively. Member routine pickup does not return until both forks can be picked up. To simultaneously pick up and put down both forks may require locking the entire table for short periods of time. No busy waiting is allowed; use cooperation among philosophers putting down forks and philosophers waiting to pick up forks. As well your solution must preclude starvation; i.e., two or more philosophers cannot conspire so that another philosopher never gets an opportunity to eat.

A philosopher eating at the table is simulated by a task, which has the following interface (you may add only a public destructor and private members):

```
_Task Philosopher {
  public:
    enum States { Thinking = 'T', Hungry = 'H', Eating ='E', Waiting = 'W', Finished = 'F' };
    Philosopher( unsigned int id, unsigned int noodles, Table &table, Printer &prt );
};
```

Each philosopher loops performing the following actions:

- hungry message
- yield a random number of times, between 0 and 4 inclusive, to simulate the time to get hungry
- pickup forks
- pretend to eat a random number of noodles, between 1 and 5 inclusive.
- eating message
- yield a random number of times, between 0 and 4 inclusive, to simulate the time to eat the noodles.

- put down forks
- if eaten all the noodles, stop looping
- thinking message
- yield a random number of times, between 0 and 19 inclusive, to simulate the time to think

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):

```
_Monitor / _Cormonitor Printer {      // choose one of the two kinds of type constructor
  public:
    Printer( unsigned int NoOfPhil );
    void print( unsigned int id, Philosopher::States state );
    void print( unsigned int id, Philosopher::States state, unsigned int bite, unsigned int noodles );
};
```

A philosopher calls the print member when it enters states: thinking, hungry, eating, waiting, finished. The table calls the print member *before* it blocks a philosopher that must wait for its forks to become available. The printer attempts to reduce output by storing information for each philosopher 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 1.

Each column is assigned to a philosopher with an appropriate title, e.g., "Phil0", and a column entry indicates its current status:

State	Meaning
H	hungry
T	thinking
W *l*,*r*	waiting for the left fork *l* and the right fork *r* to become free
E *n*,*r*	eating *n* noodles, leaving *r* noodles on plate
F	finished eating all noodles

Identify the left fork of philosopher$_i$ with number i and the right fork with number $i+1$. For example, W2,3 means forks 2 and 3 are unavailable, so philosopher 2 must wait, and E3,29 means philosopher 1 is eating 3 noodles, leaving 29 noodles on their plate to eat later. When a philosopher finishes, the state for that philosopher is marked with F and all other philosophers are marked with "...".

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. When a task finishes, the buffer is flushed immediately, the state for that object is marked with F, and all other objects are marked with "...". 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 internal representation; **do not build and store strings of text for output**.

In addition, you are to devise and include a way to test your program for erroneous behaviour. This testing should be similar to the check performed in the routine CriticalSection, from software solutions for mutual exclusion, in that it should, with high probability, detect errors. Use an assert or tests that call μC++'s uAbort routine to halt the program with an appropriate message. (HINT: once a philosopher has a pair of forks, what must be true about the philosophers on either side?)

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

```
phil [ P [ N [ S ] ] ]
```

P is the number of philosophers and must be greater than 1; if P is not present, assume a value of 5. N is the number of noodles per plate and must be greater than 0; if N is not present, assume a value of 30. S is the seed for the random-number generator and must be greater than 0. If the seed is unspecified, use a random value like the process identifier (getpid) or current time (time), so each run of the program generates different output. Use the following monitor to safely generate random values (monitors will be discussed shortly):

```
% phil 5 20 56083
Phil0  Phil1  Phil2  Phil3  Phil4
****** ****** ****** ****** ******
H      H
       E2,18  H
              W2,3   H
E1,19  T              E3,17  H
T             E2,18  T
              T      H      E4,16
              H      W3,4   T
                     E2,15
              E5,13  T
       H      T      H
H      E1,17  H      E2,13
       T             T
E1,18      H  E1,12  H
              T      E4,9
T      E5,12  H      T
              W2,3          H
       T      E2,10         E3,13
       H      T      H      T
       E1,11         E2,7   H
H                    T      W4,0
W0,1   T      H             E5,8
E3,15         E4,6          T
T
H             T      H
E2,13  H             E2,5   H
T      W1,2
       E3,8          T      W4,0
H                    H      E2,6
W0,1   T      H      W3,4   T
E5,8                 E1,4
T             E5,1   T
              T      H
                     E3,1
       H             T
H      E5,3   H             H
       T      E1,0
...    ...    F      ...    ...
                            E4,2
E2,6                        T
T
H
E2,4
T                    H      H
                     E1,0
...    ...    ...    F      ...
       H                    E2,0
...    ...    ...    ...    F
       E3,0
...    F      ...    ...    ...
H
E2,2
T
H
E2,0
F      ...    ...    ...    ...
**********************
Philosophers terminated
```

Figure 1: Output for 5 Philosophers each with 20 Noodles

```
_Monitor MPRNG {
  public:
    MPRNG( unsigned int seed = 1009 ) { srand( seed ); }      // set seed
    void seed( unsigned int seed ) { srand( seed ); }         // set seed
    unsigned int operator()() { return rand(); }              // [0,UINT_MAX]
    unsigned int operator()( unsigned int u ) { return operator()() % (u + 1); } // [0,u]
    unsigned int operator()( unsigned int l, unsigned int u ) { return operator()( u - l ) + l; } // [l,u]
};
```

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. The driver must handle an arbitrary number of philosophers and noodles, but only tests with values less than 100 for the two parameters will be made, so the output columns line up correctly.

(WARNING: in GNU C, `-1 % 5 != 4`, and on UNIX, the name fork is reserved.)

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 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. q1buffer.h, q1*.{h,cc,C,cpp} – code for question question 1a, p. 1. **Program documentation must be present in your submitted code. No test, user or system documentation is to be submitted for this question. Output for this question is checked via a marking program, so it must match exactly with the given program.**
2. q1buffer.txt – contains the information required by question 1b, p. 3.
3. q2*.{h,cc,C,cpp} – code for question question 2, p. 3. **Program documentation must be present in your submitted code. No user or system documentation is to be submitted for this question.**
4. q2phil.testtxt – test documentation for question 2, p. 3, which includes the input and output of your tests, documented by the use of script and after being formatted by scriptfix. **Poor documentation of how and/or what is tested can results in a loss of all marks allocated to testing.**
5. Use the following Makefile to compile the programs for question 1a, p. 1 and 2, p. 3:

```
KIND:=NOBUSY
OPT:=

CXX = u++                                       # compiler
CXXFLAGS = -g -Wall -Wno-unused-label -MMD ${OPT} -D"${KIND}"
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}} # makefile name

OBJECTS1 = # list of object files for question 1 prefixed with "q1"
EXEC1 = boundedBuffer

OBJECTS2 = # list of object files for question 2 prefixed with "q2"
EXEC2 = phil

OBJECTS = ${OBJECTS1} ${OBJECTS2}               # all object files
DEPENDS = ${OBJECTS:.o=.d}                      # substitute ".o" with ".d"
EXECS = ${EXEC1} ${EXEC2}                       # all executables

.PHONY : all clean

all : ${EXECS}                                  # build all executables

#############################################################
```

```
-include ImplKind

ifeq (${IMPLKIND},${KIND})                    # same implementation type as last time ?
${EXEC1} : ${OBJECTS1}
    ${CXX} ${CXXFLAGS} $^ -o $@
else
ifeq (${KIND},)                               # no implementation type specified ?
# set type to previous type
KIND=${IMPLKIND}
${EXEC1} : ${OBJECTS1}
    ${CXX} ${CXXFLAGS} $^ -o $@
else                                          # implementation type has changed
.PHONY : ${EXEC1}

${EXEC1} :
    rm -f ImplKind
    touch q1buffer.h
    ${MAKE} ${EXEC1} KIND=${KIND}
endif
endif

ImplKind :
    echo "IMPLKIND=${KIND}" > ImplKind

#################################################################
${EXEC2} : ${OBJECTS2}
    ${CXX} ${CXXFLAGS} $^ -o $@

#################################################################
${OBJECTS} : ${MAKEFILE_NAME}                 # OPTIONAL : changes to this file => recompile

-include ${DEPENDS}

clean :                                       # remove files that can be regenerated
    rm -f *.d *.o ${EXECS} ImplKind ImplType
```

This makefile is used as follows:

```
$ make boundedBuffer KIND=BUSY
$ boundedBuffer ...
$ make boundedBuffer KIND=NOBUSY OPT='-multi -O2'
$ boundedBuffer ...
$
$ make phil
$ phil ...
```

Put this Makefile in the directory with the programs, name the source files as specified above, and then type make buffer or make phil in the directory to compile 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. **If the makefile fails or does not produce correctly named executables, or if a program does not compile, you receive zero for all "Testing" marks.**

Follow these guidelines. Your grade depends on it!