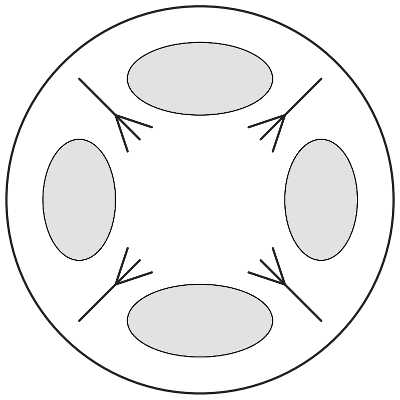
CS 343 Fall 2013 – Assignment 5
Instructor: Bernard Wong and Peter Buhr
Due Date: Monday, November 18, 2013 at 22:00
Late Date: Wednesday, November 20, 2013 at 22:00

November 11, 2013

This assignment has advanced usage of monitors and introduces 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=TcaducVVdE0 from the Dr. Who episode "Blink". **Warning: do not watch it alone!** 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 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.)

Implement a table to manage the forks as a:

(a) μC++ monitor using only internal scheduling.

(b) 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. To simulate barging use the following routine in place of normal calls to a condition-variable wait:

```
#if defined( TABLETYPE_INT )          // internal scheduling monitor solution
// includes for this kind of table
_Monitor Table {
    // private declarations for this kind of table
#elif defined( TABLETYPE_INTB )       // internal scheduling monitor solution with barging
// includes for this kind of table
_Monitor Table {
    // private declarations for this kind of table
    uCondition waiting;               // only one condition variable (you may change the variable name)
    void wait();                      // barging version of wait
#elif defined( TABLETYPE_AUTO )       // automatic-signal monitor solution
// includes for this kind of table
_Monitor Table {
    // private declarations for this kind of table
#elif defined( TABLETYPE_TASK )       // internal/external scheduling task solution
_Task Table {
    // private declarations for this kind of table
#else
    #error unsupported table
#endif
    // common declarations
  public:                             // common interface
    Table( const unsigned int NoOfPhil, Printer &prt );
    void pickup( unsigned int id );
    void putdown( unsigned int id );
};
```

Figure 1: Table Interfaces

```
void Table::wait() {
    waiting.wait();                       // wait until signalled
    while ( rand() % 5 == 0 ) {           // multiple bargers allowed
        _Accept( pickup, putdown ) {      // accept barging callers
        } _Else {                         // do not wait if no callers
        } // _Accept
    } // if
} // Table::wait
```

This code randomly accepts any of the interface routines but only if there are philosophers running outside the monitor.

(c) μC++ monitor that simulates a general automatic-signal monitor.

(d) μC++ server task performing the maximum amount of work on behalf of the client (i.e., very little code in members pickup and putdown).

(e) Explain why a μC++ monitor using only external scheduling cannot be implemented.

Figure 1 shows the different forms for each μC++ table 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 by using the preprocessor to conditionally compile a specific interface. 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. The automatic-signal and Java-simulation implementations should be different from the first implementation because there is no (or little) cooperation, and barging must be prevented for the Java-simulation implementation. An appropriate preprocessor variable is defined on the compilation command using the following syntax:

```
u++ -DTABLETYPE_INT -c TableINT.cc
```

*μ*C++ 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 RETURN( expr. . . ) . . .     // gcc variable number of parameters
```

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 waiting, the expression before is executed before the wait and the expression after is executed after the wait. Macro RETURN is used to return from a public routine of an automatic-signal monitor, where expr is optionally used for returning a value. For example, a bounded buffer implemented as an automatic-signal monitor looks like:

```
_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;
        RETURN();
    }

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

Make absolutely sure to *always* have a RETURN() macro at the end of each mutex member. As well, the macros must be self-contained, i.e., no direct manipulation of variables created in AUTOMATIC_SIGNAL is allowed from within the monitor.

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', Barging = 'B', 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, 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 2.

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
B	barging into the monitor and having to wait for signalled tasks
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/or 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. 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

```
% 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
W0,1   T      E1,0          E1,5
...    ...    F      ...    ...
E4,4                        T
T      H             H
                     E1,0
...    ...    ...    F      ...
       E1,2
H      T                    H
                            E5,0
...    ...    ...    ...    F
E2,2   H
T      E1,1
H      T
E2,0
F      ...    ...    ...    ...
       H
       E1,0
...    F      ...    ...    ...
***********************
Philosophers terminated
```

Figure 2: Output for Internal-Scheduling Monitor, 5 Philosophers, each with 20 Noodles

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.)

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.

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. q1*.txt – contains the information required by question 1, p. 1.
2. MPRNG.h,AutomaticSignal.h, q2table.h, q2*.{h,cc,C,cpp} – code for question question 2, p. 1. **Program documentation must be present in your submitted code. No user or system documentation is to be submitted for this question.**
3. q2phil.txt – contains the information required by question 2e, p. 2.
4. q2phil.testtxt – test documentation for question 2, p. 1, 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 for question 2, p. 1 to allow the different implementation types to be compiled:

```
TYPE:=INT

CXX = u++                                          # compiler
CXXFLAGS = -g -Wall -Wno-unused-label -MMD -DTABLETYPE_${TYPE}
MAKEFILE_NAME = ${firstword ${MAKEFILE_LIST}} # makefile name

OBJECTS = q2table${TYPE}.o # list of object files for question 2 prefixed with “q2”
DEPENDS = ${OBJECTS:.o=.d}                         # substitute “.o” with “.d”
EXEC = phil

.PHONY : clean

all : ${EXEC}                                      # build all executables

-include ImplType

ifeq (${IMPLTYPE},${TYPE})                         # same implementation type as last time ?
${EXEC} : ${OBJECTS}
	${CXX} ${CXXFLAGS} $^ -o $@
else
ifeq (${TYPE},)                                    # no implementation type specified ?
# set type to previous type
TYPE=${IMPLTYPE}
${EXEC} : ${OBJECTS}
	${CXX} ${CXXFLAGS} $^ -o $@
else                                               # implementation type has changed
.PHONY : ${EXEC}
${EXEC} :
	rm -f ImplType
	touch q2table.h
	${MAKE} ${EXEC} TYPE="${TYPE}"
endif
endif
```

```
ImplType :
    echo "IMPLTYPE=${TYPE}" > ImplType
    sleep 2

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

-include ${DEPENDS}

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

This makefile will be invoked as follows:

```
make phil TYPE=INT
phil ...
make phil TYPE=INTB
phil ...
make phil TYPE=AUTO
phil ...
make phil TYPE=TASK
phil ...
```

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!