CS 343 Fall 2013 – Assignment 4
Instructors: Bernard Wong and Peter Buhr
Due Date: Wednesday, November 6, 2013 at 22:00
Late Date: Friday, November 8, 2013 at 22:00

October 31, 2013

This assignment introduces monitors 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 have *only* public constructors and/or destructors; no other public members are allowed.)**

1. Run the following μC++ program with preprocessor variable PAD not defined and defined.

```
#include <iostream>
using namespace std;

volatile int counter1, counter2
#ifdef PAD
    __attribute__(( aligned (64) )) // align counters on 64-byte boundaries
#endif // PAD
    ;

unsigned int times = 100000000;

_Task Worker1 {
    void main() {
        for ( unsigned int i = 0; i < times; i += 1 ) {
            counter1 += 1;
        } // for
    } // Worker::main
}; // Worker1

_Task Worker2 {
    void main() {
        for ( unsigned int i = 0; i < times; i += 1 ) {
            counter2 += 1;
        } // for
    } // Worker::main
}; // Worker2

void uMain::main() {
    switch ( argc ) {
      case 2:
        times = atoi( argv[1] );
    } // switch

    cout << (void *)&counter1 << " " << (void *)&counter2 << endl;

    uProcessor p;          // add virtual processor
    Worker1 w1;            // create threads
    Worker2 w2;
} // uMain::main
```

(a) Compare the two 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), which is the CPU time consumed solely by the execution of user code (versus system and real time).

- Use the program command-line argument (if necessary) to adjust the number of times the experiment is performed to get execution times approximately in the range 0.1 to 100 seconds. (Timing results below 0.1 seconds are inaccurate.) Use the same command-line value for all experiments.
- Include both timing results to validate your experiments.

(b) Explain the relative differences in the timing results with respect to memory location of the counters.

(c) Explain why there is a **void** * cast before each counter when printing their address.

The material for this question will be covered later in the course so you have to research the answer to this question. Hint: the answer is in Section 10.3.2.2 of the course notes. Do not give the answer away on the course newsgroup.

2. The Santa Claus problem [Trono94] is an interesting locking problem. Santa Claus sleeps in his shop at the North Pole until he is woken. In order to wake him, one of two conditions must occur:

   (a) All five of the reindeer used to deliver toys have returned from their year-long holiday.

   (b) Three of the *E* elves have a problem that need Santa's help to solve. (One elf's problem is not serious enough to require Santa's attention; otherwise, he would never get any sleep.)

   If Santa is woken by all five reindeer, it must be Christmas, so he hitches them to the sleigh, delivers toys to all the girls and boys, and then unhitches them once they have returned. The reindeer go back on vacation until next year, while Santa goes back to sleep. If Santa is woken by three elves needing to consult him, he ushers them into his office where they consult on toy research and development, and then he ushers them out. The elves go back to work, and Santa goes back to sleep. Once Santa is awake, if he discovers that both conditions are true, priority is given to the reindeer since any problems the elves have can wait until after Christmas. Note that if Santa is consulting with one group of elves, any other elves that arrive desiring to consult must wait until Santa is ready to consult with the next group.

   To prevent the reindeer from perpetually preventing the elves from getting Santa's help, there is a bound of *N* on the number of times the reindeer can be served before a group of elves is served when there are elves waiting.

   All synchronization is performed by the monitor Workshop. The interface for Workshop is (you may add only a public destructor and private members):

```
_Monitor Workshop {
    // private members go here
  public:
    enum Status { Consulting, Delivery, Done };
    Workshop( Printer &prt, unsigned int N, unsigned int E, unsigned int D ); // printer, bound, elves, reindeer delivery
    Status sleep();                          // santa calls to nap; when Santa wakes status of next action
    void deliver( unsigned int id );         // reindeer call to deliver toys
    bool consult( unsigned int id );         // elves call to consult Santa,
                                             //    true => consultation successful, false => consultation failed
    void doneConsulting( unsigned int id );  // block Santa/elves until meeting over
    void doneDelivering( unsigned int id );  // block Santa/reindeer until all toys are delivered
    void termination( unsigned int id );     // elves call to indicate termination
};
```

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

```
_Task Santa {
    // private members go here
  public:
    Santa( Workshop &wrk, Printer &prt );
};
```

Santa executes until sleep returns status Done indicating the elves and reindeer are finished. The task main of the Santa task looks like:

- yield a random number of times between 0 and 10 inclusive so all tasks do not start simultaneously
- start message
- loop
  - yield a random number of times between 0 and 3 inclusive so that messages are not consecutive
  - napping message
  - block in workshop if nothing to do
  - awake message
  - if done consulting/deliveries, stop looping
  - if delivering toys, yield between 0 and 5 times inclusive to simulate delivery time, and then deliver-done message
  - if consulting, yield between 0 and 3 times inclusive to simulate consultation time, and then consultation-done message
- finished message

Yielding is accomplished by calling yield(times) to give up a task's CPU time-slice a number of times.

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

```
_Task Elf {
    // private members go here
  public:
    enum { CONSULTING_GROUP_SIZE = 3 };          // number of elves for a consultation with Santa
    Elf( unsigned int id, Workshop &wrk, Printer &prt, unsigned int numConsultations );
};
```

Note, numConsultations is either C or 3. The task main of the Elf task looks like:

- yield a random number of times between 0 and 10 inclusive so all tasks do not start simultaneously
- start message
- loop until done number of consultations
  - yield a random number of times between 0 and 3 inclusive so that messages are not consecutive
  - working message
  - yield a random number of times between 0 and 5 times inclusive to simulate time working
  - help message
  - wait for consultation
  - if consultation failed, consulting-failed message, stop looping
  - consulting-succeeded message
  - yield a random number of times between 0 and 3 times inclusive to simulate consultation time
  - indicate consultation done
  - consultation-done message
- indicate termination
- finished message

A consultation with Santa can fail because the number of elves may not be a multiple of the group size (3). So when elves finish, there comes a point when there are not enough elves to form a group and a consultation cannot occur. Interestingly, this problem can occur even when the number of elves is a multiple of the group size. For example, if there are 6 elves, numbered 1..6, and each does 2 consultations, there are 4 consultations. However, if elf 6 is delayed, i.e., does not participate in the initial consultations, then one possible failure scenario is:

	consultations		
	1	2	3
elves	3*	4	5
	1*	2*	4*
	5	6	

Elves 1, 2, 3, and 4 perform their 2 consultations and terminate, leaving elf 5, who has done 1 consultation and elf 6 who has done 0 consultations; since there are only two elves remaining, the final consultation cannot occur.

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

```
_Task Reindeer {
    // private members go here
  public:
    enum { MAX_NUM_REINDEER = 5 };                // number of reindeer in system for delivering toys
    Reindeer( unsigned int id, Workshop &wrk, Printer &prt, unsigned int numDeliveries );
};
```

Note that numDeliveries is either D or 3. The task main of the Reindeer task looks like:

- yield a random number of times between 0 and 10 inclusive so all tasks do not start simultaneously
- start message
- loop until done number of deliveries
  - yield a random number of times between 0 and 3 inclusive so that messages are not consecutive
  - vacation message
  - yield a random number of times between 0 and 5 inclusive to simulate vacation time
  - checking-in message
  - wait for delivery
  - delivering-toys message
  - yield a random number of times between 0 and 5 inclusive to simulate toy-delivery time
  - indicate delivery done
  - delivery-done message

  finished message

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 monitor or cormonitor
  public:
    enum States { Starting = 'S', Blocked = 'B', Unblocked = 'U', Finished = 'F', // general
          Napping = 'N', Awake = 'A',                     // Santa
          Working = 'W', NeedHelp = 'H',                  // elf
          OnVacation = 'V', CheckingIn = 'I',             // reindeer
          DeliveringToys = 'D', DoneDelivering = 'd',     // Santa, reindeer
          Consulting = 'C', DoneConsulting = 'c',         // Santa, elf
          ConsultingFailed = 'X',                         // elf
    };
    Printer( const unsigned int MAX_NUM_ELVES );
    void print( unsigned int id, States state );
    void print( unsigned int id, States state, unsigned int numBlocked );
};
```

The printer attempts to reduce output by storing information for each task 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 should look similar to that in Figure 2, p. 6. Each column is assigned to a task with an appropriate title, e.g., "E1", and a column entry indicates its current state (see Figure 1): 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**. Calls to perform printing may be performed from the tasks and/or the workshop (you decide where to print).

For example, in line 9 of Figure 2, p. 6, tasks R7 and R10 have the values "B 1" and "V" in their buffer slots and the other buffer slots are empty. When R10 attempts to print "I", which overwrites its current buffer value of "V", the buffer must be flushed generating line 9. R10's new value of "I" is then inserted into its buffer slot. When R10 attempts to print "B 2", which overwrites its current buffer value of "I", the buffer must be flushed generating line 10 and no other values are printed on the line because the print is consecutive (i.e., no intervening call from another object).

State	Meaning
S	Santa/Elf/Reindeer is starting
N	Santa is about to try and nap
A	Santa is awake from his nap
W	Elf is working
H	Elf needs to consult with Santa
V	Reindeer is on vacation
I	Reindeer is back from vacation and checking in
D	Santa/Reindeer is delivering toys
d	Santa/Reindeer is done delivering toys
C	Santa/Elf is consulting
X	Elf failed to consult
c	Santa/Elf is done consulting
B	Santa blocks going to sleep[a]
B *n*	Elves blocked (including self) for consultation
B *n*	Reindeers blocked (including self) for Christmas
B *n*	Elves + Santa blocked at end of consultation
B *n*	Reindeer + Santa blocked at end of delivery
U [*n*]	Santa/Elf/Reindeer unblock, *n* value see B
F	Santa/Elf/Reindeer is finished

[a] Santa does not specify *n* when blocking/unblocking in sleep because no choice has been made between working with the elves or reindeer.

Figure 1: Task States

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

northpole [N [E [Seed [C [D]]]]]

N is the bound on the number of times the reindeer get served ahead of the elves and must be greater than 0. If unspecified, use a default value of 3. E is the number of elves and must be greater than 0. If unspecified, use a default value of 3. Seed 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:

```
_Monitor PRNG {
  public:
    PRNG( 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. C is the number of times each elf wants to consult with Santa and must be greater than or equal to 0. If unspecified, use a default value of 3. (All elves use the same value.) D is the number of times each reindeer wants to deliver toys with Santa and must be greater than or equal to 0. If unspecified, use a default value of 3. (All reindeer use the same value.) 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.

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**

```
% northpole 1 5 1003 1 1
Sa      E1      E2      E3      E4      E5      R6      R7      R8      R9      R10
--      --      --      --      --      --      --      --      --      --      --
S                                                       S
N                                                       V
B               S               S
        S               W
        W       W                               S       I                       S
                                                        B 1                     V
                                                                                I
                H                                                               B 2
                B 1             H
        H               B 2                                     S
        B 3             S
U                       W                       V
A
C               U 3     H                       I               V
                C               U 2     S                               S
        U 1             C                                               V
B 1     C       B 2     B 1     B 3     W       B 3             I       I
U 3     c                               H                       B 4     B 5
c               U 2
N               c               U 1
                                c       B 2
...     F       ...     ...     ...     ...     ...     ...     ...     ...     ...
...     ...     F       ...     ...     ...     ...     ...     ...     ...     ...
A
D                                                       U 5
                                                        D                       U 4
                                                                                D
                                                        U 3
                                                        D               U 2
                                                                        D       U 1
                                                                                D
...     ...     ...     ...     F       ...     ...     ...     ...     ...     ...
                        U 2
                        X               U 1
B 3                             X               B 1     B 2     B 4     B 5
...     ...     ...     F       ...     ...     ...     ...     ...     ...     ...
                                                d
...     ...     ...     ...     ...     ...     F       ...     ...     ...     ...
                                                        U 5
                                                        d
...     ...     ...     ...     ...     ...     ...     F       ...     ...     ...
                                                                U 4
                                                                d
...     ...     ...     ...     ...     ...     ...     ...     F       ...     ...
U 3
d                                                                       U 2
                                                                        d
...     ...     ...     ...     ...     ...     ...     ...     ...     F       ...
N                                                                               U 1
                                                                                d
...     ...     ...     ...     ...     ...     ...     ...     ...     ...     F
...     ...     ...     ...     ...     F       ...     ...     ...     ...     ...
A
F       ...     ...     ...     ...     ...     ...     ...     ...     ...     ...
Workshop closed
```

Figure 2: Northpole: Example Output

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,q2*.{h,cc,C,cpp} – code for question question 2, p. 2. **Program documentation must be present in your submitted code. No user or system documentation is to be submitted for this question.**
3. q2northpole.testtxt – test documentation for question 2, p. 2, 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.**
4. Construct a Makefile to compile the program for question 2, p. 2. Put this Makefile in the directory with the programs, name the source files as specified above, and when make northpole is entered it compiles the program and generates the executable northpole. 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!