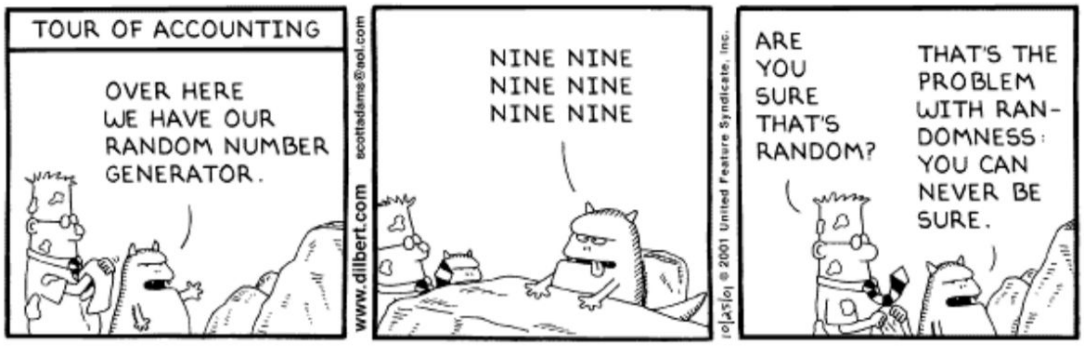
CPSC 4420 PROJECT 2: SPRING 2016
Producer/Consumer Randomness Tester

Due: _____, at the beginning of class, through Canvas (source code). *Also bring a paper printout of your source code to class.*

*** If your program does not compile, you will receive no credit!! Double check
*** that your program will compile before submitting it.



Overview:
Write a program to use multiple threads to generate random numbers and perform a basic randomness check on these numbers. Three producer threads will generate random numbers using 3 different methods. These random numbers will be placed in shared buffers. Three consumer threads will process the random numbers, performing a basic randomness check. The parent process will report the results after all threads have exited.

The basic randomness check that we will use will examine the only the lowest place digit (the rightmost digit of the integer generated), and maintain a count of how many times that digit occurs in the numbers generated. NOTE: all higher digit information will be discarded – we are only looking at the lowest place digit. For example, if the number 1804289383 is generated, we will only examine the 3, and ignore the other digits. (Note: a thorough test of randomness is a deep and complex subject. You can learn more about it here: http://www.random.org/analysis/)

Pseudorandom Number Generators:
Most random number generators used in computing today are known as pseudorandom number generators. Each generator will create a sequence of numbers that are random, based on a starting "seed". In practical use, it is very important to give the generators a high-quality seed that is full of "entropy" (or randomness). The reason the seed is important is this: *the generator will ALWAYS give the same sequence of random numbers for a given seed*. This is known as pseudorandomness. In order to make testing of the code easy, we will all use the same seed.

We will test 3 pseudorandom number generators to ensure that our code is working.

1. The **random()** pseudorandom number generator included in the gcc libraries. We will use the starting seed of 1, so that the first 3 random numbers generated should be: 1804289383, 846930886, and 1681692777.
2. An extremely **poor** random number generator. This will be a function that you write that always returns the same number. (Not very random!)
3. The **RANDU** pseudorandom number generator which was created in 1960 and was widely used until its defects were discovered. (See http://en.wikipedia.org/wiki/RANDU which states that it was "most ill-conceived random number generators ever designed.") It uses a recurrence relation to generate new numbers based on an initial seed (we will again use a starting seed of 1, so that the first three random numbers are 1, 65539, and 393225). The relation is:

$$V_{j+1} = 65539 * V_j \bmod 2^{31}$$

where V_j is the last random number generated (or the initial seed, for the first number), and V_{j+1} is the newest random number. Our basic randomness test will show what the problem with the RANDU method is, when using *int* variables in C to store the results.

Specifics:

1. Your program should not crash. Ever. Errors should be handled gracefully and meaningful error messages produced.
2. Use a #define statement to define these two values:
   a) a DEBUG flag to print (or not) debugging statements from the producers and consumers.
   b) a NUM_TO_TEST to specify how many random numbers to generate. Begin by testing just 10, and work up to 1,000,000 or so.
3. The main() method will create/destroy the mutexes and condition variables. You will most likely need a mutex and a condition variable for each producer/consumer pair. The condition variable will synchronize access to the shared buffer where the random numbers are stored. The size of this buffer must be at least 10.
4. main() will create/join the producer and consumer threads.
5. main() will analyze the results from the consumer threads, and display a table of the results from the basic randomness test.
6. The producer thread will generate NUM_TO_TEST random numbers, storing them in the shared buffer as quickly as possible. A condition variable will be used to cause the producer to block when the buffer has no empty slots.
   a) The RANDU function must use *int* integers to perform the calculations (not *long int* variables) to make the problem clear.
7. The consumer thread will process NUM_TO_TEST values from the shared buffer as quickly as possible, examining the lowest order digit (the rightmost digit), and maintaining a count of the number of times that digit is generated. The same condition variable can be used to cause the consumer to block when the buffer has no data.
8. Exactly how your implement the shared buffer is your choice, but it is recommended that you use a circular queue. We will review the circular queue in class.

Style:

As always, follow good programming style. Comment your code as discussed in class (all variables commented, purpose of all functions described, sufficient comments in the code to make it clear the purpose of each code segment). In addition, only use global variables when absolutely necessary. Mutexes, condition variables, and threads should be created, initialized, destroyed, and joined properly.

Hints:

1. I strongly recommend writing this project in stages, where you first have just two threads (one producer, one consumer) and get one of the pseudorandom number generators to work correctly successfully report on the basic randomness test. Then add a second pseudorandom number generator (and two more threads), and get it to work correctly...and so on.
2. You will find that you need quite a few global variables. They are required for threads, but you want to ask yourself, for each one, is it really necessary? If more than one thread needs access to that variable, it needs to be global. If not, find a way to make the variable local.
3. You will probably want each thread to have quite a few if (DEBUG) statements, so that you can see which thread is doing what, and how the OS is forcing them to take turns.
4. In order to calculate the RANDU formula, you will likely want to use the *pow(x, y)* function to calculate 2^{31}: *(int)pow(2, 31)*. The cast to int is needed because *pow()* returns a double, and you can only calculate modulus operations on integers. (Type *man pow* at the linux command line for more information.)

Sample Output:

(debugging messages turned off, 1,000,000 random numbers generated for each method)

Results for process A:

Mod i	Number of Occurrences
0	99697
1	100027
2	99935
3	99818
4	100440
5	99771
6	100010
7	99969
8	100073
9	100260

Results for process B:

Mod i	Number of Occurrences
0	0
1	0
2	1000000
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Results for process C:

Mod i	Number of Occurrences
0	3319
1	98878
2	1622
3	98319
4	1627
5	98139
6	1657
7	98471
8	1653
9	98288

You can also execute:

~coyj/cpsc4420/project/project2/10debugon as well as **20debugon**, **100debugoff,** and **million** to see additional sample output