CS342 Fall 2015 – Project 3
Synchronization: Mutex and Condition Variables

Assigned: November 02, 2015, Monday
Due date: November 16, 2015, Monday, 23:55

In this project you will implement two C libraries, a DP (dining philosophers) library and a thread-safe hash-table data structure library. You will use POSIX pthreads library and its mutex and condition variables (synchronization primitives provided by pthreads library) in your implementation. You libraries will be used by sample/test applications. You will develop your project in Linux. The project will be done individually.

Part A. Develop a DP (dining philosophers) library (libdp.so) that will implement getforks and putforks operations of the DP problem (that we have seen in the class) so that an application can use the library to simulate the DP problem by use of N threads. You will use POSIX pthreads *mutex* and *condition* variables in your library implementation.

A DP application will use your library as follows (pseudo-code given).

```
#include <dp.h>
main (argc, argv)
{
        int N = atoi (argv[1]);
        dp_init (N);

        for (i=0; i<N; ++i)
                create_thread (philosopher, i);

}
philosopher (i)  // thread function acting as a philosopher
{
        //think – in THINKING state
        sleep (a random amount);
        dp_get_forks (i);
        //eat – in EATING state
        sleep (random amount);
        dp_put_forks (i);
}
```

Your library will implement 3 functions:

- void **dp_init (int N).** Initializes the environment so that N philosophers (threads) can start running.
- void **dp_get_forks (int i).** Tries to acquire two forks (left and right). If forks are available returns immediately, otherwise causes the calling thread to wait until forks become available.
- void **dp_put_forks (int i).** Releases the left and right forks. May cause a waiting neighbor to return from waiting state and start eating.

You can define mutex and condition variables in your library. Initially all philosophers are in THINKING state and all forks are available. A philosopher needs two forks (left and right) to eat. The number of philosophers, N, will be specified by the application (taken as a command line parameter). N can be greater than or equal to 5 and smaller than or equal to 20.

A DP application app.c will be compiled and linked with your library as follow:

```
gcc –Wall –o app –llibdp.so app.c
```

We can run the application as follows, for example:

```
./app 5
```

PART B. Develop a library (libhash.so) that will implement the hash table data structure in a thread-safe manner. A hash table has N buckets (buckets 0 through N-1) and a key i will be inserted into bucket j = hash(i), where j is in range [0, N-1]. In this project, the hash function will be a simple hash function, i.e., hash (i) = i mod N. N can be at least 10 and at most 100. Multiple keys mapping to the same bucket will be added to a linked list (chaining). In this way collisions will be resolved. Hence, for each bucket of the hash table we will have a linked list, initially empty. We will insert keys into the hash table. The keys could have associated values as well, but not in this project (for simplicity). A key can be student ID, for example. The key type is integer. Valid keys are positive.

You library will implement the following functions:

- **void hash_init (int N).** Creates a hash table of N buckets. Each bucket will have an associated linked list (chain) initialized to empty list.
- **int hash_insert (int k).** Inserts key k into the table. If success returns 0, otherwise returns -1.
- **int hash_delete (int k).** Removes key k from the table. If success returns 0, otherwise returns -1.
- **int hash_get (int k).** If k exists in the table, returns k, otherwise returns -1.

Your library should implement these operations in a thread-safe manner. For that you will use mutex variables (locks). You should use one mutex per bucket to increase concurrency. If you lock the whole hash table, then the performance will be low.

A multi-threaded application, app.c, that will use your library will first include the header file “hash.h” corresponding to your library. The application may create many threads and each thread may do a lot of table operations. We assume the application will use exactly one hash table. The current library specification does not allow the creation and use of multiple hash tables. An application will be compiled and linked with your library as follows:

```
gcc –Wall –o app –llibhash.so app.c
```

We will develop test applications to test and stress your library implementation.

Report. Do some timing experiments and write a report about the results. You can measure, for example, the time to finish an application that does a lot hash table operations with some number of threads. You can repeat the experiments for various numbers of threads for various sequences of hash table operations. You also change the table size (number of buckets) and repeat the experiments. Put tables and/or figures into your report showing the results. Try to interpret the results.

Clarifications:

- Your library will be a shared library (.so), not a static library (.a). You need to learn how to create a shared library.
- You need to learn how to use mutex and condition variables. There are links to some resources in the References section of the course webpage. You can find additional resources from Internet.