CS 343 Fall 2013 – Assignment 3
Instructors: Bernard Wong and Peter Buhr
Due Date: Monday, October 21, 2013 at 22:00
Late Date: Wednesday, October 23, 2013 at 22:00

October 7, 2013

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.

1. MapReduce is a popular distributed computation framework that offers a simple yet flexible job interface. A MapReduce job consists of one or more Mapper and Reduce tasks. A Mapper task reads from an input source, performs some computation on the data, and outputs a stream of key-value pairs. The Reduce tasks partition the key-space (e.g. R_1 gets keys from A-M and R_2 gets keys from N-Z), with each Reduce task retrieving the key-value pairs in its partition from the output streams of every Mapper task.

   The canonical example for a MapReduce job is word count, which counts the number of times each word appears in a large document corpus. In this job, each Mapper task reads from one document and generates an output stream of “word : 1” for each word in the document, where the key is “word” and the value is 1. Each Reduce task retrieves the key-value pairs in its key-space partition, and sums up the values of all key-value pairs with the same key. Upon completion, each Reduce task prints out the frequency of each word in its partition. Figure 1 illustrates the word count workflow.

   A simple approach to partitioning the key-space is through hashing. Given N Reduce tasks with task IDs from 0 to $N-1$, a Reduce task is responsible for those keys where $hash(key)$ % N is equal to its task ID.

   Write a shared-memory *simulation* of the distributed word-count program as a MapReduce job and implement a simplified MapReduce framework. This program creates a Mapper task for each file in a specified directory (see the man pages for opendir, readdir and closedir). Do not recursively look for files inside directories within the specified directory. The Mapper task has the following interface (you may only add private members):

   ```
   _Task Mapper {
       void main();
     public:
       struct KeyValue {
           string key;
           int value;

           KeyValue(const string& key, int value) : key(key), value(value) {}
           KeyValue() {}
       };
       Mapper(const string& filename, int queue_len, uSemaphore* signal);
       virtual ~Mapper();
       uSemaphore* getSignal();
       const string& getFilename();
       KVQueue* q;
   };
   ```

   The Mapper task creates a KeyValue object for each word in the input file. Words can be fetched from an ifstream as follows (no strenuous parsing required!):

   ```
   datastream >> word;
   ```

   The value for each KeyValue object is 1. Each Mapper task has a KeyValue queue (bounded buffer) q. Each Mapper task pushes its KeyValue objects onto its own q. The size of the queue is determined by the parameter queue_len. The interface for the queue is as follows (you may only add private members):

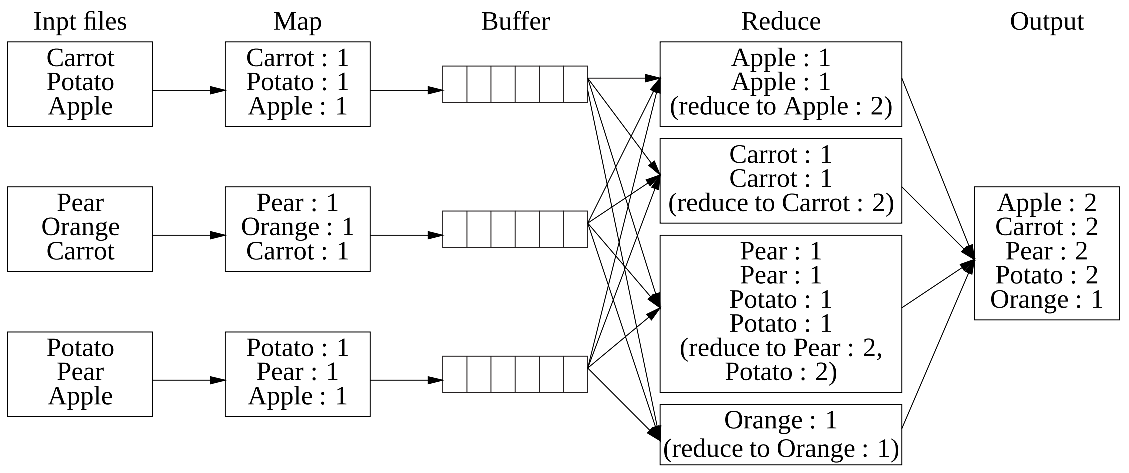

Figure 1: The MapReduce word count workflow.

```
class KVQueue {
  public:
    struct EmptyAndClosed {
        int num_thrown;
        EmptyAndClosed(int num_thrown) : num_thrown(num_thrown) {}
    };
    KVQueue(int size);
    void pushBack(const Mapper::KeyValue& item);
    Mapper::KeyValue popFront();
    // Copy the front value to "val" without removing it from the queue. Return 0 on success and -1
    // if the queue is empty. Throw an EmptyAndClosed exception if the queue is closed and empty.
    int peekFront(Mapper::KeyValue* val) throw(EmptyAndClosed);
    void close();
};
```

Calling popFront blocks if the queue is empty. Similarly, calling pushBack blocks if the queue is full. The mapper calls close on the queue after pushing its last KeyValue object. The function peekFront does not block and throws an EmptyAndClosed exception if the queue is closed and empty. The num_thrown parameter should indicate the number of times the EmptyAndClosed exception has been thrown for this queue. The behaviour of calls to popFront and pushBack is undefined after the queue is closed.

The Reducer task has the following interface (you may only add private members):

```
_Task Reducer {
    void main();
  public:
    Reducer(int id, int num_reducers, uSemaphore* signal, const vector<Mapper*>& mappers);
    virtual ~Reducer();
    // DJB2 hash. Include as part of the class interface.
    unsigned long hash(const string& str) {
        unsigned long hash = 5381;
        for (unsigned int i = 0; i < str.size(); ++i) {
            hash = ((hash << 5) + hash) + str[i];
        }
        return hash;
    }
    int getID();
    int getNumReducers();
    uSemaphore* getSignal();
    vector<Mapper*>& getMappers();
};
```

Reducer tasks read from the Mapper queues and retrieve objects where:

```
hash(key) % num_reducers == id
```

Each Reducer task sums the values for each key, and upon completion, prints out the count for each key (one key per line). Ensure the proper stream locks are used to avoid overlapping output.

A semaphore is used by the Mapper tasks to signal to the Reducer tasks that data is available in one or more queues. It is important that the Reducer tasks are blocked on this semaphore when data is unavailable. At the end of the program, print out the counter for the signalling semaphore in the following format:

```
Finished! Semaphore counter: 0
```

A problem with this overly simplified MapReduce framework is that Reducer tasks have to keep track of the count of every key in memory until completion. For extremely large corpuses, even with each Reducer task running on a separate machine, the memory requirement of each Reducer task might exceed the available memory. Most MapReduce frameworks address this problem by sorting the Mapper task's output stream, which may require spilling intermediate state to disk in an efficient way while sorting. Also while sorting, each Mapper task can sum the values with the same key to reduce the amount of data it needs to send to the reducers (this is often described as a *combiner* stage). Therefore, Reducer tasks only need to sum the values with the same key from different Mapper tasks. Given sorted output streams, the Reducer tasks can output the count of a word once it has processed all words that are lexicographically smaller from every Mapper task. This optimization significantly reduces the per machine memory requirement.

Therefore, in addition to the basic approach, you must also implement a Mapper task that outputs a sorted stream, and a Reducer task that can output word counts prior to retrieving all of the key-value pairs from the queues. The interfaces are as follows (you may only add private members):

```
_Task SortMapper : public Mapper {
    void main();
  public:
    // Writes items to the queue in sorted order.
    SortMapper(const string& filename, int queue_len, int buffer_size, uSemaphore* signal);
    virtual ~SortMapper();
};

_Task SortReducer : public Reducer {
    void main();
  public:
    SortReducer(int id, int num_reducers, uSemaphore* signal, const vector<Mapper*>& mappers);
    virtual ~SortReducer();
};
```

A simple approach to sorting the output stream is as follows. Keep only the smallest buffer_size keys in a buffer (or a std::map) in the first pass of the input; combine the values for these keys during the scan. After the first scan of the input data, the sorted and combined key-value pairs are written to the queue. In the second pass of the input, skip all keys that have already been written to the queue and keep the smallest buffer_size keys from the remaining keys. Keep performing passes over the input data until all key-value pairs have been sent to the queue.

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

```
wordcount input-directory [ num-reducers ( > 0 ) [ queue-length ( > 0 ) [ sort-buffer-size ( >= 0 ) ] ] ]
```

The default values are: num-reducers = 4, queue-length = 16, sort-buffer-size = 0. If the sort-buffer-size is greater than 0, then SortMapper and SortReducer are used in place of Mapper and Reducer.

For the input file:

```
cs343 queue semaphore
queue list map reduce
assignment list queue
```

The program should generate the following output:

```
cs343 : 1
list : 2
reduce : 1
semaphore : 1
map : 1
assignment : 1
queue : 3
Finished! Semaphore counter: 0
```

Note that the ordering of the words is arbitrary.

You may notice the program's performance generally degrades with additional reducers. Try adding more uProcessors to see if it improves performance. The MapReduce framework has been significantly simplified to make the assignment tractable, and certain design decisions were made to exercise specific synchronization concepts. Describe some of the synchronization-related problems that limit the performance of this MapReduce design, and propose changes to the framework to improve its performance in a single machine, multi-core environment.

Submission Guidelines

Please follow these guidelines very 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 a maximum of 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*.{h,cc,C,cpp} – code for question 1. **Program documentation must be present in your submitted code. No 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. q1*.testtxt – test documentation for question 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.**

3. q1improvements.txt – contains the information required by question 1.

4. Construct a Makefile to compile the program for question 1, p. 1. Put this Makefile in the directory with the program files, name the source files as specified above, and when make wordcount is entered it compiles the program and generates the executable wordcount. 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.**

   Put this Makefile in the directory with the programs, name the source files as specified above, and then type make wordcount 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!