CMPUT 379 – Operating System Concepts **Fall 2019**
Assignment 2 – MapReduce Library

This assignment will be due **November 1, 2019 at 11:59pm** Mountain Time.

Objective

This programming assignment is intended to give you experience in using POSIX threads and synchronization primitives.

MapReduce Paradigm

MapReduce is a programming model and a distributed computing paradigm for large-scale data processing. It allows for applications to run various tasks in parallel, making them scalable and fault-tolerant. To use the MapReduce infrastructure, developers need to write just a little bit of code in addition to their program. They do not need to worry about how to parallelize their program; the MapReduce runtime will make this happen!

To use the MapReduce infrastructure, developers must implement two *callback* functions, namely `Map` and `Reduce`. The `Map` function takes a filename as input and generates intermediate key/value pairs from the data stored in that file. The `Reduce` function processes a subset of intermediate key/value pairs, *summarizing* all values associated with the same key. The summary operation typically reduces the list of values associated with a key to a single value. It could be as simple as summing up numerical values (e.g., in a distributed word count program) or concatenating string values (e.g., in a distributed grep program) associated with the same key.

The MapReduce infrastructure utilizes three types of computing resources as shown in Figure 1:

1. workers executing the `Map` function (*mappers*);
2. workers executing the `Reduce` function (*reducers*);
3. a master (*controller*) assigning tasks to the two types of workers mentioned above.

These computing resources can be threads running on a multicore system or nodes in a distributed system. The numbers of mappers and reducers must be specified by the developer. Depending on the workload, adding more workers could improve the performance or not.

Your Task

In this programming assignment you are going to build a MapReduce library in C or C++ utilizing POSIX threads and synchronization primitives. This library will support the execution of user-defined `Map` and `Reduce` functions on a multicore system. The original MapReduce paper[1] shows the programming paradigm to work in a distributed computing environment where each worker in the map phase has its own set of intermediate key/value pairs stored locally. These intermediate pairs are then distributed across the workers (in the shuffle phase) before the reduce phase starts.

You will implement a different version of MapReduce in this assignment. We assume individual workers are threads running on the same system. Hence, your MapReduce library should create a fixed number of mapper threads (kept in reserve in a thread pool) and a fixed number of reducer threads to run the computation in parallel.

[1]Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1), 107-113.

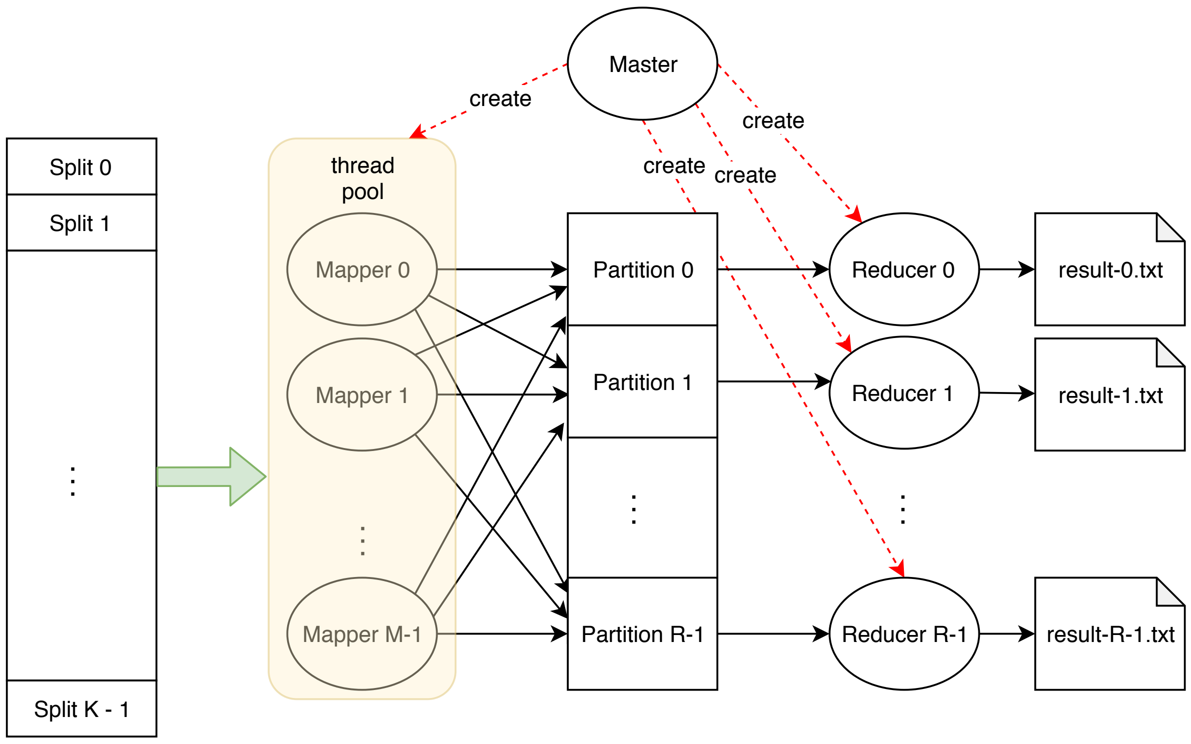

Figure 1: MapReduce execution overview

Implementing the thread pool using synchronization primitives is a central challenge of this assignment. You are not allowed to use an existing thread pool library.

Since the intermediate key/value pairs are stored in shared data structures, your MapReduce library must also use synchronization primitives to access this data. Otherwise, the data will be overwritten and lost. Designing and implementing this data structure is another challenge of the assignment.

MapReduce Execution

MapReduce execution can be divided into 7 steps that must be completed one after the other. These steps are explained below:

1. **Creating mappers**: the master thread creates M mapper threads in the `MR_Run` library function where M is an argument of this function. The mapper threads are in a thread pool implemented using synchronization primitives, such as mutex locks and condition variables. By maintaining a fixed number of threads, the thread pool helps to minimize the thread creation and destruction overhead. Note that the master thread will wait for the mapper threads to process all splits in the map phase before moving on to the reduce phase.
2. **Assigning data splits to mappers**: in the `MR_Run` function, the master thread iterates over K input data splits (represented by K files) and submits a job for each split to the thread pool. If there is an idle mapper thread in the thread pool, it starts processing the job right away by invoking the user-defined `Map` function with the passed argument (i.e., the filename). Otherwise, the job is added to a queue to be processed later by an idle mapper thread. The master ensures that each split is processed exactly once. Note that there are several different ways that jobs can be assigned to idle mapper threads; your implementation should use the *longest job first* policy. This policy assigns the longest job (i.e., the largest input file) in the queue to the next idle mapper thread. To check the size of each input file, you can use the `stat` system call.

3. **Running the user-defined Map function**: each mapper runs the user-defined `Map` function on a given split to generate key/value pairs. Every time a key/value pair is generated, the `Map` function invokes the `MR_Emit` library function, to write the intermediate key/value pair to a particular shared data structure. Since multiple mapper threads may attempt to concurrently update a shared data structure, this library function must use proper synchronization primitives, such as mutex locks.
4. **Partitioning the map output**: when the `MR_Emit` function is called by a mapper thread, it first determines where to write the given key/value pair by calling the `MR_Partition` function. The `MR_Partition` function takes a key and maps it to an integer between 0 and $R-1$, which indicates the data structure the key/value pair must be written to. This way `MR_Partition` allows the MapReduce library to create R separate data structures (i.e., *partitions*), each containing a subset of keys and the value list associated with each of them. Once the partition that a key/value pair must be written to is identified, `MR_Emit` inserts the pair in a certain position in that partition, keeping the partition sorted in **ascending key order** at all times. This is necessary for the `MR_GetNext` library function to easily distinguish when a new reduce task should start, which is exactly when the next key in the sorted partition is different from the previous key.
5. **Creating reducers**: the master thread terminates the mapper threads when all K splits are processed in the map phase. It then creates R reducer threads in the `MR_Run` library function where R is an argument of this function. Each reducer thread is responsible for processing data in a particular partition; hence, the ith partition ($0 \leq i < R$) is assigned to the ith reducer. Each reducer thread runs the `MR_ProcessPartition` library function which takes as input the index of the partition assigned to it.
6. **Running the user-defined Reduce function**: the `MR_ProcessPartition` library function invokes the user-defined `Reduce` function on the next unprocessed key in the given partition in a loop. Thus, `Reduce` is invoked only once for each key. To perform the summary operation, the `Reduce` function calls the `MR_GetNext` library function to iterate over all values that have the same key in its partition. The `MR_GetNext` function returns the next value associated with the given key in the sorted partition or `NULL` when the key's values have been processed completely.
7. **Producing the final output**: each reducer thread writes the result of the summary operation performed on the value list associated with the passed key to a file named `result-<partition_number>.txt`. These output files are created initially by the reducer threads. The master thread terminates a reducer thread as soon as all keys in its corresponding partition are processed. The master thread frees up its resources and returns from the `MR_Emit` function once all keys in all R partitions are processed.

The MapReduce Library

We now describe the functions in the `mapreduce.h` header file (included in the starter code) to help you understand how to write your MapReduce library:

```
// function pointer types used by library functions
typedef void (*Mapper)(char *file_name);
typedef void (*Reducer)(char *key, int partition_number);

// library functions you must define
void MR_Run(int num_files, char *filenames[],
            Mapper map, int num_mappers,
            Reducer concate, int num_reducers);

void MR_Emit(char *key, char *value);

unsigned long MR_Partition(char *key, int num_partitions);
```

```
void MR_ProcessPartition(int partition_number);

char *MR_GetNext(char *key, int partition_number);
```

The MapReduce execution is started off by a call to MR_Run in the user program. This function is passed an array of filenames containing K data splits, that is filenames$[0], \cdots$, filenames$[K-1]$ ($K =$ num_files). Additionally, the MR_Run function is passed the number of mappers M, the number of reducers R, and two function pointers for Map and Reduce functions. The main thread that runs the MR_Run function is the master thread. It creates M mappers and R reducers as depicted in Figure 1.

The MR_Emit library function takes a key and a value associated with it, and writes this pair to a specific partition which is determined by passing the key to the MR_Partition library function. This function can be any "good" hash function such as CRC32, MurmurHash, or the following algorithm known as DJB2:

```
unsigned long MR_Partition(char *key, int num_partitions) {
    unsigned long hash = 5381;
    int c;
    while ((c = *key++) != '\0')
        hash = hash * 33 + c;
    return hash % num_partitions;
}
```

You can use the above hash function or the open-source implementation of a well-known hash function. Make sure to cite your sources in the readme file.

The MR_ProcessPartition library function takes the index of the partition assigned to the thread that runs it. It invokes the user-defined Reduce function in a loop, each time passing it the next unprocessed key. This continues until all keys in the partition are processed.

The MR_GetNext library function takes a key and a partition number, and returns a value associated with the key that exists in that partition. In particular, the ith call to this function should return the ith value associated with the key in the sorted partition or NULL if i is greater than the number of values associated with the key.

In the following section, we provide an example user program which is useful for testing your MapReduce library. Your task is to implement the library assuming that there exists a parallelizable user program that includes your library and calls the MR_Run function.

Example Map and Reduce Functions

To illustrate how the library functions are invoked from the user program, we describe a simple **distributed word count** program, distwc.c, which uses the MapReduce library. A word count program counts the number of times each word appears in a given set of files. We will assume that the filenames are valid and the files exist in the file system. This main function takes a list of text files and passes it to MR_Run along with the numbers of mappers and reducers, and two function pointers. All the developer has to do is to include the MapReduce library and implement Map and Reduce functions. The distributed word count program is described below.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mapreduce.h"
```

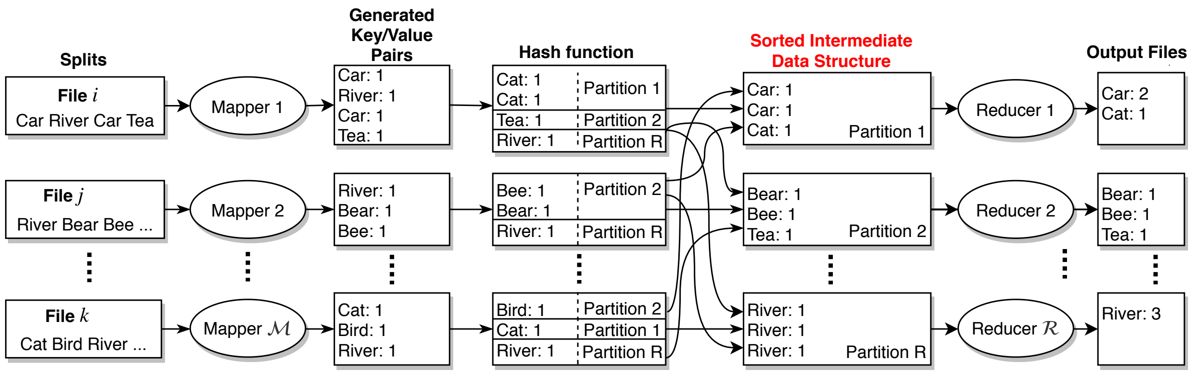

Figure 2: Example MapReduce data flow

```
void Map(char *file_name) {
    FILE *fp = fopen(file_name, "r");
    assert(fp != NULL);

    char *line = NULL;
    size_t size = 0;
    while (getline(&line, &size, fp) != -1) {
        char *token, *dummy = line;
        while ((token = strsep(&dummy, " \t\n\r")) != NULL) {
            MR_Emit(token, "1");
        }
    }
    free(line);
    fclose(fp);
}

void Reduce(char *key, int partition_number) {
    int count = 0;
    char *value, name[100];
    while ((value = MR_GetNext(key, partition_number)) != NULL)
        count++;
    sprintf(name, "result-%d.txt", partition_number);
    FILE *fp = fopen(name, "a");
    fprintf(fp, "%s: %d\n", key, count);
    fclose(fp);
}

int main(int argc, char *argv[]) {
    MR_Run(argc - 1, &(argv[1]), Map, 10, Reduce, 10);
}
```

Using Pthreads

To implement the MapReduce library, you must use the POSIX threads library (pthreads) for thread management. See the man page of pthreads for more information. We are going to assume that the intermediate key/value pairs are fixed during the reduce phase. Thus, you must wait until all mapper threads finish execution, before starting the reduce phase.

Intermediate Data Structures

The data structures being used for the intermediate key/value pairs are shared among multiple threads. Hence, they must be global variables in the MapReduce library and support concurrency for correctness. Each thread must obtain the lock on a shared data structure before updating it. For locking and unlocking the data structure you will use `pthread_mutex`.

The implementation of the shared data structures is up to you. You might use data structures from C++ STL or implement the data structures you need (if you write code in C, for example). While there are many ways to implement this data structure, it must be thread-safe, and support efficient implementation of `MR_Emit` and `MR_GetNext` functions.

Thread Pool

The MapReduce library adopts a thread pool library to create a fixed number of mapper threads and assign the task of running the `Map` callback function on input files to them.
In this assignment, you will write your own thread pool library using POSIX mutex locks and condition variables. Below we describe the functions and struct declarations in the `threadpool.h` header file to help you understand what this library is supposed to do.

```
typedef void (*thread_func_t)(void *arg);

typedef struct ThreadPool_work_t {
    thread_func_t func;              // The function pointer
    void *arg;                       // The arguments for the function
    // TODO: add other members here if needed
} ThreadPool_work_t;

typedef struct {
    // TODO: add members here
} ThreadPool_work_queue_t;

typedef struct {
    // TODO: add members here
} ThreadPool_t;

/**
* A C style constructor for creating a new ThreadPool object
* Parameters:
*     num - The number of threads to create
* Return:
*     ThreadPool_t* - The pointer to the newly created ThreadPool object
*/
ThreadPool_t *ThreadPool_create(int num);

/**
* A C style destructor to destroy a ThreadPool object
* Parameters:
*     tp - The pointer to the ThreadPool object to be destroyed
*/
void ThreadPool_destroy(ThreadPool_t *tp);

/**
* Add a task to the ThreadPool's task queue
* Parameters:
*     tp   - The ThreadPool object to add the task to
*     func - The function pointer that will be called in the thread
*     arg  - The arguments for the function
* Return:
*     true  - If successful
*     false - Otherwise
*/
bool ThreadPool_add_work(ThreadPool_t *tp, thread_func_t func, void *arg);

/**
* Get a task from the given ThreadPool object
* Parameters:
*     tp - The ThreadPool object being passed
* Return:
*     ThreadPool_work_t* - The next task to run
*/
ThreadPool_work_t *ThreadPool_get_work(ThreadPool_t *tp);

/**
* Run the next task from the task queue
* Parameters:
*     tp - The ThreadPool Object this thread belongs to
*/
void *Thread_run(ThreadPool_t *tp);
```

The `ThreadPool_create` and `ThreadPool_destroy` functions create and destroy the ThreadPool object, respectively. Each thread created by `ThreadPool_create` runs the `Thread_run` function which gets a task from the task queue and executes it (this is done in a loop). The `ThreadPool_destroy` function should wait until all tasks are executed before destroying the ThreadPool.

The `ThreadPool_add_work` function is used to submit a task (i.e., the execution of a function) to the thread pool. You must create your own task queue and have a mechanism to support concurrency. The `ThreadPool_get_work` function is used to get a task from the queue and have it processed by an idle thread.

You will have to define `ThreadPool_work_t`, `ThreadPool_work_queue_t`, and `ThreadPool_t` structs in `threadpool.h`.

Deliverables

The starter code includes two header files, `mapreduce.h` and `threadpool.h`, and one example program, `distwc.c`, that utilizes the MapReduce library. Your task is to implement the MapReduce and ThreadPool libraries. You might modify `mapreduce.h` and `threadpool.h`.

Submit your assignment as a single compressed archive file (`mapreduce.zip` or `mapreduce.tar.gz`) containing:

1. A custom Makefile with at least these targets: (a) the main target `wc` which simply links all object files and produces an executable file called `wordcount`, (b) the target `compile` which compiles your code and the provided distributed word count program, and produces the object file(s), (c) the target `clean` which removes the objects and executable files, and (d) the target `compress` which creates the compressed archive for submission.
2. A plain text document, called `readme.md`, which explains how to store the intermediate key/value pairs, the time complexity of `MR_Emit` and `MR_GetNext` functions, the data structure used to implement the task queue in the thread pool library, and your implementation of the thread pool library. Furthermore, you need to explain what synchronization primitives you used and how you tested the correctness of your code. Make sure you cite all sources that contributed to your assignment in this file. You may use a markup language, such as Markdown, to format this plain text file.
3. All files required to build your project including `mapreduce.h` and `threadpool.h`.

Misc. Notes

- This assignment must be completed **individually** without consultation with anyone besides the instructor and TAs.
- You can write code in C or C++, and compile it with `gcc` or `g++` passing `-Wall -Werror -pthread` flags. No warnings should be returned when your code is compiled these flags. Make sure that your code does not print anything extra for debugging.
- We encourage you to write more parallelizable programs that utilize the MapReduce library. Test your MapReduce library using these programs with different numbers of mapper threads and reducer threads besides the distributed word count program that we provided.
- You are not allowed to use a standard thread pool library in this assignment. You must implement a thread pool yourself using synchronization primitives.
- The use of implicit threading (e.g., OpenMP, Intel TBB, etc.) is not permitted in this assignment.
- Check your code for memory leaks. You may use the Valgrind tool suite:

  ```
  valgrind --tool=memcheck --leak-check=yes ./wordcount
  ```
- You can use your own machine to write the code, but you must make sure that it compiles and runs on the Linux lab machines (e.g., ugXX.cs.ualberta.ca where XX is a number between 00 and 34).
- When developing and testing your program, make sure that you clean up all processes before you logout of a workstation.