CMPUT 379 — Operating System Concepts **Fall 2025**

Assignment 2 — Multithreaded MapReduce Library

This assignment will be due **November 5, 2025 at 5:59 PM** Mountain Time.

Objective

In this assignment, you will learn how to use POSIX threads and synchronization primitives.

MapReduce

MapReduce is a programming model and a distributed computing paradigm for large-scale data processing. It allows applications to run various tasks in parallel, making them scalable and fault tolerant.

To use MapReduce, developers have to write just a bit of code in addition to their program. They do not need to worry about parallelizing their program; the MapReduce runtime library will make this happen! Specifically, developers must implement two *callback* functions, namely `Map` and `Reduce`. The `Map` function takes a file-name 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, reducing all the values associated with a key to a single value typically. The reduction operation could be as simple as summing numerical values (e.g., in a distributed word count program) or concatenating strings associated with the same key (e.g., in a distributed grep program).

The MapReduce library utilizes three types of computational resources as illustrated in Figure 1:

1. workers executing the `Map` function, referred to as *mappers*;
2. workers executing the `Reduce` function, referred to as *reducers*;
3. a controller assigning tasks to both types of workers.

The numbers of workers is usually specified by the user and adding more workers improves performance in general.

Your Task

In this programming assignment you are going to build a MapReduce library in C utilizing POSIX threads and POSIX synchronization primitives, in particular mutex locks and condition variables. This library supports the execution of user-defined `Map` and `Reduce` functions on a multicore system.

The original MapReduce paper[1] presents the paradigm 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 map-outputs are then distributed among the workers in a shuffle phase before the reduce phase starts. You will implement a different version of MapReduce in this assignment. We assume that individual workers are threads running on the same system. Hence, the MapReduce library should create a fixed number of threads, kept in reserve in a thread pool to first run map tasks in parallel and then reduce tasks in parallel. Implementing the thread pool using synchronization primitives is the 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, referred to as *partitions*, the MapReduce library must use synchronization primitives to access these partitions too. Otherwise, the data may be overwritten.

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

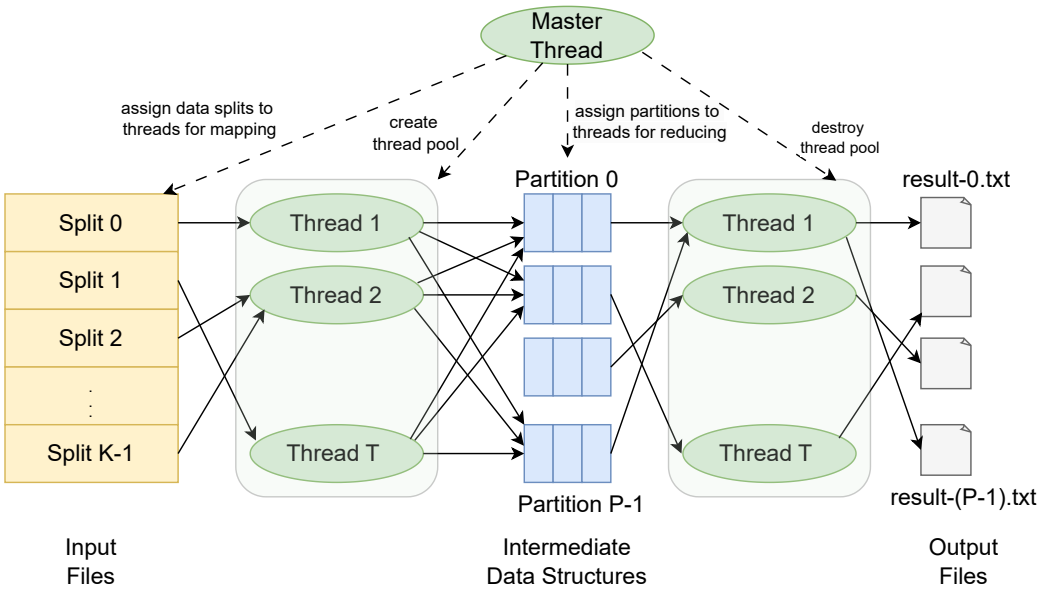

Figure 1: Overview of the MapReduce framework

MapReduce Execution

The MapReduce execution is divided into 7 steps that are completed sequentially. These steps are explained below:

1. **Creating a thread pool**: The master thread, in a library function called `MR_Run`, creates a thread pool that holds T threads. These threads will be used to perform map tasks in the map phase and reduce tasks in the reduce phase of MapReduce. By keeping a fixed number of threads in reserve, the thread pool helps minimize the overhead of thread creation and destruction.
2. **Assigning data splits to threads in the map phase**: The master thread, in `MR_Run`, submits a map job for each of the K input data splits (represented by K distinct files) to the thread pool. If there is an idle thread in the thread pool, that thread starts processing the job right away by invoking the user-defined `Map` function, passing the corresponding filename as an argument. Otherwise, jobs are added to a job queue to be processed later when a thread becomes idle. It is important to ensure that each split is processed exactly once in the map phase.

   Note that jobs in the queue can be assigned to idle threads in different ways. Your implementation must use the *shortest job first* policy, which assigns the shortest job (i.e., the smallest input data split) in the queue to the next idle thread.[2] To achieve this, the master thread submits jobs to the thread pool in sorted order.
3. **Running the Map function to produce the map outputs**: The user-defined `Map` function generates intermediate key-value pairs given a data split. Once each key-value pair is generated, it invokes the `MR_Emit` library function to write that pair to a partition. Since multiple mapper threads may attempt to update a partition concurrently, `MR_Emit` must use synchronization primitives properly.
4. **Partitioning the map outputs**: The `MR_Emit` function finds the partition that should hold each key-value pair by calling `MR_Partitioner`. The `MR_Partitioner` library function uses a hash function to map the key of the respective pair to an integer between 0 and $P - 1$, indicating the index of the partition that the key-value pair must be written to. By doing that, it allows the MapReduce library to create P separate partitions, each containing a subset of keys and the values associated with them. Once the partition that will

[2]To check the size of a file, you can use the `stat` system call.

hold a key-value pair is determined, `MR_Emit` inserts the pair in that partition, keeping the partition sorted in **ascending key order** at all times. This is necessary for the `MR_GetNext` library function to be able to decide when a new reduce task should start, which is exactly when the next key in the partition differs from the current key.

5. **Assigning partitions to threads in the reduce phase**: The master thread, in `MR_Run`, must wait until all input data splits are mapped and all threads are idle before starting the reduce phase. Once these conditions are met, it submits a reduce job for each of the P partitions to the thread pool. If there is an idle thread in the thread pool, it starts processing the job right away by invoking the `MR_Reduce` library function. Otherwise, jobs are added to the job queue to be processed later when a thread becomes idle. As stated earlier, the job queue is managed using the *shortest job first* policy so the master thread submits jobs in sorted order. The size of a reduce job depends on the size of the corresponding partition.[3] It is important to ensure that each partition is processed exactly once in the reduce phase.

   Note that `MR_Reduce` takes as input the index of the corresponding partition and the user-defined `Reduce` function.

6. **Running the user-defined `Reduce` function to produce the final outputs**: The `MR_Reduce` library function invokes the user-defined `Reduce` function on each key that is found in its partition in an iterative manner. Thus, `Reduce` is invoked exactly once for each key. To perform the reduce task, `Reduce` calls the `MR_GetNext` library function to iterate over the values that are associated with the key and reduce these values to a single value. Specifically, `MR_GetNext` returns the next value associated with the given key in the partition or `NULL` when all values associated with the key are reduced.

7. **Writing the final outputs and destroying the thread pool**: Each thread writes the result of the reduce task performed on the list of values associated with the given key to a file named `result-X.txt` with `X` being the partition number between 0 and $P-1$. The master thread, in `MR_Run`, destroys the thread pool when all partitions are reduced and all threads are idle.

The MapReduce Library

We now describe the functions in the `mapreduce.h` header file (included in the starter code) to help you understand what they are supposed to do and how they should be implemented:

```
// typedefs
typedef void (*Mapper)(char *file_name);
typedef void (*Reducer)(char *key, unsigned int partition_idx);

/**
* Run the MapReduce framework
* Parameters:
*     file_count   - Number of files (i.e. input splits)
*     file_names   - Array of filenames
*     mapper       - Function pointer to the map function
*     reducer      - Function pointer to the reduce function
*     num_workers  - Number of threads in the thread pool
*     num_parts    - Number of partitions to be created
*/
void MR_Run(unsigned int file_count, char *file_names[],
```

[3] Since partitions are kept in memory, unlike the input files, you cannot use the `stat` system call to get their size. Instead, you can store the size of each partition in a variable and update it every time that `MR_Emit` is called, based on the total size of all key-value pairs in that partition.

```
            Mapper mapper, Reducer reducer,
            unsigned int num_workers, unsigned int num_parts);
```

The MapReduce execution starts with a call to the `MR_Run` library function in the user program. This function is passed an array of filenames containing K input splits, that is filenames$[0], \cdots,$ filenames$[K-1]$. Additionally, the `MR_Run` function is passed two callback functions for the `Map` and `Reduce` tasks, the number of threads that will be used in the map and reduce phase denoted T, and the number of intermediate data structures that will hold map outputs denoted P. The master thread runs `MR_Run` to create a thread pool of size T threads and uses these threads to perform the map and reduce tasks as depicted in Figure 1.

```
/**
* Write a specific map output, a <key, value> pair, to a partition
* Parameters:
*     key           - Key of the output
*     value         - Value of the output
*/
void MR_Emit(char *key, char *value);

/**
* Hash a mapper's output to determine the partition that will hold it
* that will hold this output
* Parameters:
*     key           - Key of a specific map output
*     num_partitions- Total number of partitions
* Return:
*     unsigned int  - Index of the partition
*/
unsigned int MR_Partitioner(char *key, unsigned int num_partitions);
```

The `MR_Emit` library function takes a key-value pair produced by a mapper and writes it to a specific partition. This partition is determined by passing the key of this pair to the `MR_Partitioner` library function. This function can be any "good" hash function, such as the following algorithm known as DJB2:

```
unsigned int MR_Partitioner(char *key, unsigned int num_partitions) {
    unsigned long hash = 5381;
    int c;
    while ((c = *key++) != '\0')
        hash = ((hash << 5) + hash) + c;    // hash * 33 + c

    return hash % num_partitions;
}
```

To simplify your task, use the hash function above in your implementation.

```
/**
* Run the reducer callback function for each <key, (list of values)>
* retrieved from a partition
* Parameters:
*     threadarg     - Pointer to a hidden args object
*/
void MR_Reduce(void *threadarg);
```

```c
/**
* Get the next value of the given key in the partition
* Parameters:
*     key           - Key of the values being reduced
*     partition_idx - Index of the partition containing this key
* Return:
*     char *        - Value of the next pair if its key is the current key
*     NULL          - Otherwise
*/
char *MR_GetNext(char *key, unsigned int partition_idx);
```

The `MR_Reduce` library function takes the index of the partition assigned to the thread that is running. It invokes the user-defined `Reduce` function in a loop, each time passing it the next unprocessed key in that partition. This continues until all keys in the partition are processed. For each key, the user-defined `Reduce` function reduces the list of values to a single value and writes the result to a file.

The `MR_GetNext` library function is called from the user-defined `Reduce` function to get each value that is associated with a key. Specifically, `MR_GetNext` takes a key and a partition number, and returns a value associated with the key that exists in that partition. The ith call to this function should return the ith value associated with the key in the 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 that is useful for testing your MapReduce runtime 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 an application, we describe a simple **distributed word count** program that relies on the MapReduce library. This program is included in the starter code (see `distwc.c`).

A word count program counts the number of times each word appears in a given set of files. We will assume that filenames are valid and the corresponding files exist in the file system. The `main` function takes some text files as arguments and passes them to `MR_Run` along with two function pointers that will be used in the map and reduce phase, the numbers of threads in the thread pool, and the number of intermediate data structures that will hold the result of the map operation. All the developer has to do is to include the MapReduce library and implement the `Map` and `Reduce` functions. The distributed word count program is described below and illustrated in Figure 2.

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

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");
```

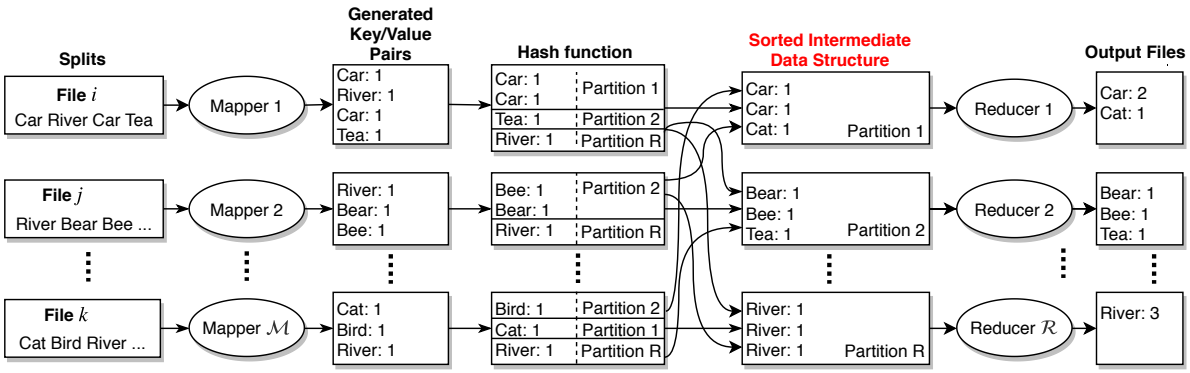

Figure 2: Illustration of the distributed word count application written using the MapReduce library. For simplicity, the use of a thread pool with a fixed number of threads is not shown here.

```
        }
    }
    free(line);
    fclose(fp);
}

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

int main(int argc, char *argv[]) {
    // using 5 threads within the thread pool and 10 intermediate data structures
    MR_Run(argc - 1, &(argv[1]), Map, Reduce, 5, 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.

Intermediate Data Structures

The partitions are used as intermediate data structures to hold key-value pairs that are accessed by multiple threads. Hence, they must be global variables in the MapReduce library and support concurrent access for correctness. Each thread must obtain a lock, specifically a `pthread_mutex`, before accessing a shared data structure.

The implementation of the shared data structures is part of your task. These data structures must support efficient implementation of `MR_Emit` and `MR_GetNext` functions. Since you cannot use the Standard Template Library (STL) in C, implementing each partition as key-value pairs that are linked together using pointers would be acceptable.

The Thread Pool Library

The MapReduce library utilizes a thread pool library to create a fixed number of threads and have them run the callback functions in the map and reduce phase.
In this assignment, you must write your own thread pool library using POSIX mutex locks and condition variables. Below, we describe the function and struct declarations in the `threadpool.h` header file to help you understand what this library is supposed to do.

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

typedef struct ThreadPool_job_t {
    thread_func_t func;              // function pointer
    void *arg;                       // arguments for that function
    struct ThreadPool_job_t *next;   // pointer to the next job in the queue
    // add other members if needed
} ThreadPool_job_t;

typedef struct {
    unsigned int size;               // no. jobs in the queue
    ThreadPool_job_t *head;          // pointer to the first (shortest) job
    // add other members if needed
} ThreadPool_job_queue_t;

typedef struct {
    pthread_t *threads;              // pointer to the array of thread handles
    ThreadPool_job_queue_t jobs;     // queue of jobs waiting for a thread to run
    // add other members if needed
} ThreadPool_t;

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

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

```
/**
* Add a job to the ThreadPool's job queue
* Parameters:
*     tp   - Pointer to the ThreadPool object
*     func - Pointer to the function that will be called by the serving thread
*     arg  - Arguments for that function
* Return:
*     true  - On success
*     false - Otherwise
*/
bool ThreadPool_add_job(ThreadPool_t *tp, thread_func_t func, void *arg);

/**
* Get a job from the job queue of the ThreadPool object
* Parameters:
*     tp - Pointer to the ThreadPool object
* Return:
*     ThreadPool_job_t* - Next job to run
*/
ThreadPool_job_t *ThreadPool_get_job(ThreadPool_t *tp);

/**
* Start routine of each thread in the ThreadPool Object
* In a loop, check the job queue, get a job (if any) and run it
* Parameters:
*     tp - Pointer to the ThreadPool object containing this thread
*/
void *Thread_run(ThreadPool_t *tp);

/**
* Ensure that all threads are idle and the job queue is empty before returning
* Parameters:
*     tp - Pointer to the ThreadPool object
*/
void ThreadPool_check(ThreadPool_t *tp);
```

The `ThreadPool_create` and `ThreadPool_destroy` library functions create and destroy the ThreadPool object, respectively. Upon creation, each thread will run the thread start function, called `Thread_run`, to get a job from the job queue and run it. This must be done in a loop as jobs may be added to the job queue at different times. Before destroying the ThreadPool object, you must ensure that (a) there is no job remaining in the job queue and (b) all threads are idle. The `ThreadPool_check` function waits for these conditions to be satisfied before it returns.

The `ThreadPool_add_job` function is used to submit a job (i.e., execution of a function with some arguments) to the thread pool. The `ThreadPool_get_job` function is used to get a job from the queue and have it processed by an idle thread. Note that, for full marks, the job queue must be managed using the Shortest Job First (SJF) policy. We give partial marks for using other policies, such as First Come, First Served (FCFS), instead of SJF.

Note that you must use synchronization primitives, in particular `pthread_mutex` and `pthread_cond` to properly support concurrency in this library. To this end, you may add new data members to `ThreadPool_job_t`,

`ThreadPool_job_queue_t`, and `ThreadPool_t` structs in `threadpool.h`. For full marks, the same ThreadPool object must be used in the map and reduce phase.

Starter Code

When you accept the assignment and clone the repository, you will find two header files: `mapreduce.h` and `threadpool.h`. In addition to these header files, there will be 20 text files (in the `testcase` directory) and one example program (`distwc.c`) that utilizes the MapReduce library and can be used to test your submission. To do this, pass the text files as command line arguments to the distributed word count program. In total, there are 21 words in these files and the occurrence count of each word is exactly 5000 across the 20 text files.

Submission

Submit all the required files via GitHub Classroom. Do not use compressed files or archives.

The following files must be included in your repository.

1. Two C files, namely `mapreduce.c` and `threadpool.c`, to implement the MapReduce library and the ThreadPool library.
2. A custom `Makefile` (use this exact name) with at least these five targets: (a) target `threadpool.o` that compiles `threadpool.c` and creates an object file, (b) target `mapreduce.o` that compiles `mapreduce.c` and creates an object file, (c) target `distwc.o` that compiles `distwc.c` and creates an object file, (d) target `wordcount` that links all object files to produce an executable called `wordcount`, (e) target `clean` that removes the object files and the executable. You may add more targets to your Makefile.
3. A plain text document, called `README.md` (use this exact name), that contains a header (see below), lists the synchronization primitives that you used in each library, describes how partitions are implemented, and elaborates on how your implementation was tested, e.g. using different numbers of input files, partitions, and worker threads. **In particular, measure the running time of the `wordcount` program as the number of worker threads increases (for a fixed number of input files and partitions) and discuss why the reduction in running time becomes limited (if any) beyond a certain number of threads.**

   Make sure you cite all sources that you used in this file. You may use a markup language, such as Markdown, to format this plain text file.

   At the very top of the readme file before any other lines, include **a header** with the following information: **your name, student ID (SID), and CCID**. For example:

   ```
   # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   # Name : Jane Doe
   # SID : 1234567
   # CCID : jdoe
   # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   ```

4. All files required to build your project, including the header files `threadpool.h` and `mapreduce.h`. As mentioned earlier, you can modify `threadpool.h` to add data members to the structs.

Note that the distributed program `distwc.c` and sample input files that were provided as part of the starter code should also be included in the repository. You may add additional .c and .h files to the repository if necessary.

Misc. Notes

- This assignment must be completed individually without consultation with anyone. Questions can be asked from TAs in the labs.
- The libraries must be implemented in C (**not C++**) and compiled using `gcc` with `-Wall -pthread` flags. Do not use `-O` flags for compiler optimization. Your code should compile cleanly with these flags, i.e. producing no warning or error.
- We encourage you to write your own parallelizable programs that utilize the MapReduce library and test your implementation with a different number of files, partitions, and threads.
- You are not allowed to use an existing thread pool library in this assignment. You must implement the thread pool yourself using POSIX 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 also use the helgrind tool (`--tool=helgrind`) in Valgrind to detect some synchronization errors.
- You may use your personal computer for programming, but you must make sure that your code compiles and runs correctly on a Linux lab machine. The list of these machines can be found on Canvas.