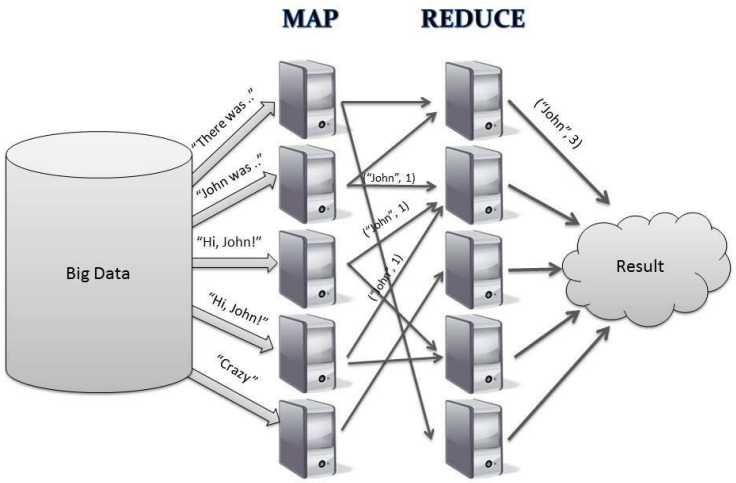
OS 2015/2016 – Targil 3
MapReduce Framework
Multi-threaded programming

Author and Supervisor – Netanel Zakay
Assisted by Ami Hollander and Or Keren

Background – MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster [A computer cluster consists of a set of loosely or tightly connected computers that work together so that, in many respects, they can be viewed as a single system].



A MapReduce program is composed of a **Map()** procedure (method) that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a **Reduce()** method that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The MapReduce framework orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.
Definition\clarification: A ***software framework*** is a universal, reusable software environment that provides particular functionality as part of a larger software platform to facilitate development of software applications.

The model is inspired by the "*map()*" & "*reduce()*" functions commonly used in functional programming. The key contributions of the MapReduce framework are not the actual map and reduce functions, but the scalability and fault-tolerance achieved for a variety of applications by optimizing the execution engine once. As such, a single-threaded implementation of MapReduce will usually not be faster than a traditional (non-MapReduce) implementation, any gains are usually only seen with multi-threaded implementations.

MapReduce libraries have been written in many programming languages, with different levels of optimization. A popular open-source implementation that has support for

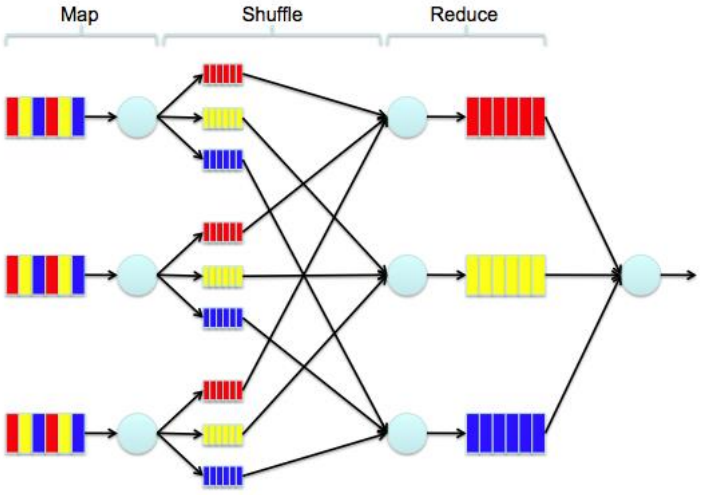
distributed shuffles is part of Apache Hadoop. The name MapReduce originally referred to the proprietary Google technology, but has since been genericized.

MapReduce details

MapReduce is a framework for processing parallelizable problems across huge datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes are on the same local network and use similar hardware) or a grid (if the nodes are shared across geographically and administratively distributed systems, and use more heterogeneous hardware). However, in this exercise, you will implement a MapReduce Framework for a single computer. This is done by creating a single thread for each task. Therefore, each *node* is a *thread* in your case. Our goal is to optimize the processing time of a program by taking advantage of the available cores in our multi-core computer. While our MapReduce will work on the lab servers, it will be much better in a server (with many cores).

The user who uses MapReduce framework supplies two functions – *Map* and *Reduce*. From the user perspective, the MapReduce framework is composed of three stages:

1. "Map" step: Each worker node (thread) applies the "map()" function to the local data, and writes the output to a temporary data structure.
2. "Shuffle" step: Worker nodes (thread) redistribute data based on the output keys (produced by the "map()" function), such that all data belonging to one key is located on the same worker node.
3. "Reduce" step: Worker nodes now process each group of output data, per key, in parallel.



This figure was taken from Wikipedia and is a slightly misleading. For example, the *shuffle* **doesn't** seem to be a process here. See our figure later for exact details of our MapReduce Framework.

MapReduce allows for distributed processing of the map and reduction operations. Provided that each mapping operation is independent of the others, all maps can be performed in parallel. Similarly, a set of 'reducers' can perform the reduction phase, provided that all outputs of the map operation that share the same key are presented to the same reducer at the same time.

In this exercise, you will create the MapReduce Framework (named *MapReduceFramework*) as a library. Using your library, any person will be able to parallel his work, by implementing the Map and Reduce functions, and use your interface to run them simultaneously. You will

also use your MapReduce Framework to create a simple Search program in order to test your framework and to understand it better.

MapReduce user

User flow – concept

We will start the description from the user side. The user is practically a test for your framework, and therefore it creates a Test Driven Development method. We hope that after understanding how the user uses the framework, you will understand the framework better.

A user uses the Map Reduce Framework in order to parallel single task execution. The Map and Reduce functions of MapReduce are both defined with respect to data structures in (key, value) pairs. Map takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

Map: (k1,v1) → list(k2,v2)

The Map function is applied in parallel to every pair in the input dataset. This produces a list of pairs for each call. The MapReduce framework then collects all pairs with the same key from all lists and groups them together, creating one group for each key. This step is called *Shuffle*, and practically does the following: **shuffle (list<k2,v2>) -> list (k2, list<v2>)**. Note that the *Shuffle* is part of the framework, in contrast to *map* and *reduce* which are implemented by the user.

The Reduce function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:

Reduce: (k2, list (v2)) → list(k3,v3)

Each Reduce call typically produces either one pair of (k3, v3) or an empty return, though one call is allowed to produce multiple pairs of (k3, v3). The produced pairs of all Reduce calls are collected as the desired result list.

Thus the MapReduce framework transforms a list of (key1, value1) pairs into a list of (key3, value3).

For example, assuming that a user wants a program that counts the number of appearances of each word in a few documents. A possible implementation is as following.

1. Create the *Map* function. The function receives a key & value, where the key is the file name and the value is null. It opens the file, goes over all the words in the document, and for each word w, it adds the value <w,1> to a list.
2. In this stage, the *map* function finished creating a list of <w,1> for each occurrence of each word. Then the framework runs the *shuffle* function, which merges them into a list of <w, list <1 > >.

   Now we create the *Reduce* function. The function receives a key & list<values>, where the key is a word, and the value is 1. The function counts the number of ones, and returns a pair of <word, count>

3. Now, we use the framework with the following steps:
    a. Create a list of the documents names with null value.
    b. Use the MapReduce Framework with the Map and Reduce functions described above and list of documents. This produces a list of <word, count>, as expected.

As you can see from the steps above, the user doesn't need to program in a multi-threaded environment or even to know what a thread is. This multi-thread programing is done (and hidden by) the framework. The user only divides his (big) task into Map and Reduce functions, and their execution is done efficiently by the framework.

User in our project

To use the library, a user must link his file with the compiled library (*MapReduceFramework.a*) and include two header files. The first is MapReduceClient.h. This header contains the data structures and functions that the user needs to implement in order to use the library. The other file is MapReduceFramework.h. This file contains the Framework functions that the user should use.

The user needs to do the following steps to use the MapReduceFrameWork:

1. Create the keys and values types as described above. This is done by inheriting from the k1Base, v1Base, k2Base, v2Base, v3Base, k3Base and creating their own key and values types.
   For example, if the *k1Base* is a file name, he may create a class that inherits from k1Base named *FileNameKey*, which has a string field (*fileName)*. Pay attention that the keys must implement "***operator<"*** to allow sorting by keys.
2. Create a class which inherits from *MapReduceBase* and implements the *Map* and *Reduce* functions. The problem here is that the framework is generic, and therefore all functions receive the base class for keys and values. Therefore, the expected implementation of the functions is to have a (ugly) down-casting from the base type, to the user types created in step 1. Then you will be able to use the fields and functions that you defined in your classes.
    - The Map function should use the library function Emit2 to add a new pair of < k2Base, v2Base >
    - The Reduce function should use the library function Emit3 to add a new value of < k3Base, v3Base >
3. Create a list of <k1Base, v1Base> pairs. This is the input of the Map function.
4. Execute the function *runMapReduceFramework*, with an instance of the class you created in step 2 and the input created in step 3. The output of the function is a list of pairs <k3Base, v3Base>. Use down-casting to cast them to the custom types created at step 1.

Task 1 – use MapReduceFramework to create Search program

You are required to submit a program named *Search.cpp*. The *Search* program receives the following arguments:

- Substring to search (String).
- Folders path - any number of folders to search in.

For example:
Search ments myDocument\ c:\pictures

The program searches all the given folders, prints to the screen all the files that include the given substring in their name. For example, if we run: "*Search os os2015/exercise blabla",* and the folders are:

- *"os2015/exercise"*, which contains the following files: *TA*, *BLA*, *osTargil*, *sos*, and *targilOs*.
- *"blabla"*, which is not a directory.

The output will be:
osTargil sos

Remarks:

- Your task is to implement this program by using the framework (that you will implement later).
  The first step is to divide the work correctly between the Map and the Reduce functions. You are required to explain your design in the README file.
- If the program doesn't receive an argument, print to the standard error: "Usage: <substring to search> <folders, separated by space>".
  For any other number of arguments, the program should work. If there are no folders, no output will be printed.
- As demonstrated above, if you receive an invalid path, it will be skipped and no errors will be printed.
- There is no need to search recursively. This means that you search the files only in the base folders.
- While we used the name "file", we actually mean to each type of file, including directory, link, etc.

Task 2 - MapReduceFramework

MapReduceFramework flow – concept

The purpose of the *MapReduceFramework* is to run the *Map* and *Reduce* functions of the user in a multi-threaded environment in order to have fast and parallel execution of a complex program. To have a real concurrency we will also minimize the shared variables between the different flows. This way the threads won't block each other from running.

From the Framework's perspective, a common MapReduce has five parallel and distributed steps:

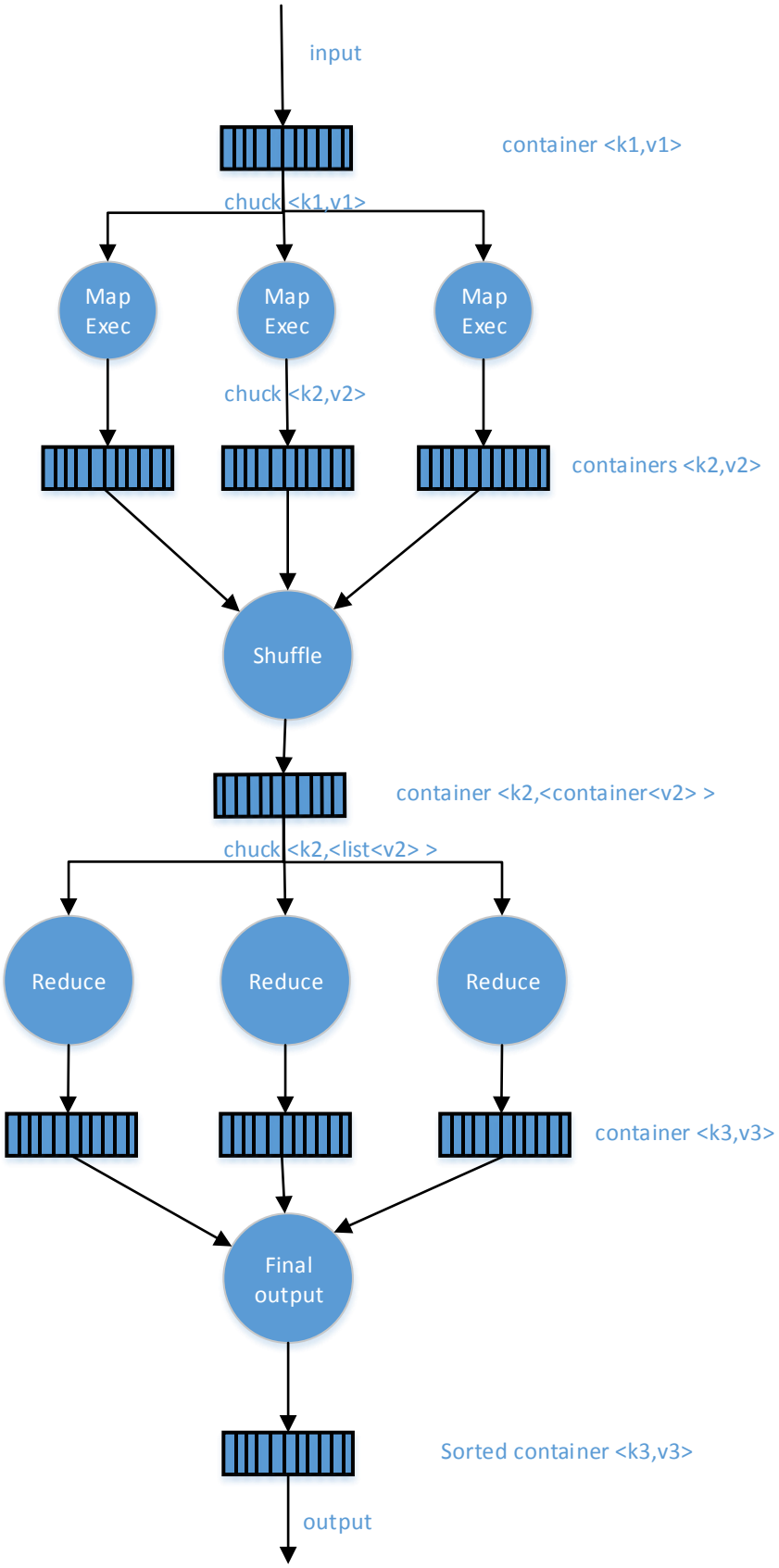
1. **Prepare the Map() input** – in the general case, this step includes searching and preparing the data for the map function. In our case, the data is sent to the *runMapReduceFramework* as a parameter.

2. **Run the user-provided Map() code** – run the Map() function, generating output in pairs of <k2,v2>.
3. **Shuffle the Reduce () input** – the MapReduce system designates Reduce function, assigns a k2 key value each function should work on, and provides that function with all the Map-generated data associated with that key value.
4. **Run the user-provided Reduce() code** – Reduce() runs exactly once for each k2 key value produced by the Map step.
5. **Produce the final output** – the MapReduce system collects all the Reduce output, and sorts it by k3 to produce the outcome.

MapReduceFramework implementation

In this stage, we assume that you already understand the concept and the basic flow of the MapReduceFramework. Here we will walk through the requirements for your MapReduceFramework implementation. In the flow above, the steps were sequential. This means that they are easy to understand (and therefore good for intuition), but extremely inefficient.

Before continuing on, please look at the supplied header files.


input
container <k1,v1>
chuck <k1,v1>
Map Exec
Map Exec
Map Exec
chuck <k2,v2>
containers <k2,v2>
Shuffle
container <k2,<container<v2> >
chuck <k2,<list<v2> >
Reduce
Reduce
Reduce
container <k3,v3>
Final output
Sorted container <k3,v3>
output

1. *ExecMap* threads

Our goal is to execute the Map function in multi-threading. The easiest approach is to create a single thread for each task. However, this will lead to extremely inefficient code due to both – the time to create the threads and the context switching between so many threads. Instead, we create *multiThreadLevel* threads [*multiThreadLevel* is a parameter of the *runMapReduceFramework*, see the header file for more details], each one of these threads runs the Map function with a single pair of <key, value> in a loop. These threads are called *ExecMap*.

The next task is to decide which keys each *ExecMap* will handle. The basic solution would be to divide the keys equally between the threads. However, this may lead to extremely unbalanced work-division between the different threads. This may happen when the run time of the *Map* function depends on < keys1, values1>, and therefore certain values can consume much more time than others.

A simple solution is to create a queue-like data structure, which contains pairs of <keys1, values1>. All the *ExecMap* threads pop pairs from it and send them to the *Map* function as long as the queue is not empty.
This solution is based on the same principles as the common Thread Pool pattern (explained in class) to balance the work between the *ExecMap* threads. In this pattern, there is a need to have a shared-mutex that protects this queue. However, in our case the tasks done by *ExecMap* threads are usually short. Therefore, *ExecMap* will lock the queue often, and this will limit the concurrency of the program.

In order to maximize the parallelism we will do the following. First, *ExecMap* will not pop from the queue a single value, but a chunk of values. For this exercise, it's enough to use a constant chunk (e.g. ten), but you are allowed to use other method to choose the chunk size. In this way, each *ExecMap* will access the queue less often. As a result, much less locks will be done by *ExecMap,* and therefore they will not block each other.

Second improvement, we will minimize the time that each *ExecMap* locks the mutex. Instead of popping from a queue a chunk of elements, which might be time consuming, we will do the following. First we will use a vector/array instead of a queue and the *ExecMap* will only read from this data structure. Therefore, there is no need to protect it. The only shared variable that will be changed by each *ExecMap* is an integer that represents the next value to read. All the threads update this index and therefore it should be protected. However, the mutex can (and should) be locked for as short as a few commands (one line of code is enough).

Using these two improvements, the different *ExecMap* threads may run simultaneously with rarely locking each other. This leads to a highly efficient flow.

3. *Emit2* function

This function is called by the *Map* function (implemented by the user) in order to add a new pair of <k2,v2> to the framework's internal data structures. This function is part of the MapReduceFramework's API.

Pay attention that *ExecMap* threads use the Map function often. In order to maximize the concurrency between the different threads, you aren't supposed to lock a shared mutex between the different *ExecMap* threads in each call to *emit2* function. This can be easily avoided if the *Emit2* of different *ExecMap threads* writes to different locations (e.g. different

queues). This means that you should create a data structure per thread rather than a single shared data structure.

Tip: the function *pthread_self()* may be needed here.

4. *Shuffle* thread

The *Map* functions create many pairs of <k2, v2> (usually orders of magnitude more than the pairs of <k1,v1> that *Map* receives). The goal of the shuffle function is to merge all the pairs with the same key. The *Shuffle* converts a list of <k2, v2> to a list of <k2, list<v2> >, where each element in this list has a unique key.

The simplest implementation would be to do the *Shuffle* after the end of the *Map* functions (means that the *ExecMap* terminated). However, the *Shuffle* may take a long time, and parallelizing this task is a complicated mission. Therefore, this solution may cause the *MapReduceFramework* to run in a single threaded mode for a long time, which is inefficient. We want to avoid this situation.

Our solution is to create a thread for the *Shuffle* simultaneously to the creation of the *ExecMap* threads. The *Shuffle* waits as long as there is no data to sort. The *ExecMap* thread should wake it up when data is written. Note, it's similar to the consumer-producer problem, but with multiple producers (*ExecMap* threads) and a single consumer (the *shuffle* thread).

Some technical tips to implement it efficiently. It is clear that this thread should use a conditional variable to wait, and should be awaken using *pthread_cond_signal* after writing a chunk (and not after writing each element). However, this approach has a hidden bug - the *Shuffle* will not receive any signal as long as it doesn't wait. Therefore, when it sorts data, several signals may be ignored. Then, when it will reach the "wait" line, it may not receive another signal, and a deadlock will occur. To avoid this, we highly recommend you to use *pthread_cond_timedwait* (for example, you may use timeout of 0.01 seconds).

Finally, pay attention that you are not required to lock a mutex in order to check if a container is empty. The *shuffle* is the only function that pulls elements from this container. Therefore, if a thread already wrote data to the container and sent a signal to wake the *Shuffle*, it will not be empty. On the other hand, note that the *shuffle* may “think” (by using the empty function or the size) that a data structure is empty, when it's practically not (why?). But this is not a problem in our design (again, why?).

5. *ExecReduce* threads

When you reach this stage, you know that all the previous stages have been completed. This means that you are back in a single threaded mode, and you have a data structure that contains pairs of <k2, list<v2> > (created by the *Shuffle* thread).

Here you execute the *Reduce* function efficiently by using ExecReduce. This is done exactly in the same manner as you executed the *map* function efficiently using *ExecMap*. With a bit more details – you are supposed to create *multiThreadLevel* threads named ExecReduce that call the *Reduce* function with a single key in a loop. These functions consume data in chunks (with a fixed size of 10 for example) from a shared data structure.

6. *Emit3*

The *Emit3* function is used by the *Reduce* function in order to add a pair of <k3, v3> to the final output. The rest of the details of *Emit3* are similar to the *Emit2*. However, here the

situation is more simple, because all the data that *Emit3* creates is used only after the termination of the *ExecReduce* functions. This makes our life much easier.

7. Producing the final output

You reach this stage in a single thread mode, after the termination of the ExecReduce functions. Each ExecReduce function produces a container of <k3, v3>. The last stage is to merge all the containers that have been created, sort them by k3, and return them to the user. In contrast to the shuffle step that usually handle many values, the list of outputs (<k3, v3>) is usually much shorter, and therefore there is no need to parallelize this step.

Producing the log file

Like any big framework, your MapReduceFramework must produce a log file. The log will be saved in the current working directory, and will be called: *.MapReduceFramework.log* (notice the "." prefix, meaning it is a hidden file). If the log already exists, it will be opened and appended to. Otherwise, it will be created. The following information should be printed to the log with exactly the specified syntax.

- "runMapReduceFramework started with MULTI_THREAD_LEVEL threads\n" is printed in the beginning of the runMapReduceFramework, where MULTI_THREAD_LEVEL is the value of multiThreadLevel parameter.
- "Thread THREAD_NAME created [DD.MM.YYYY HH:MM:SS]\n" is created by each thread when it has been created, where THREAD_NAME is the name of the function as described above. The possible thread names are: *ExecMap, Shuffle*, and *ExecReduce*. *For* example:
  "Thread ExecReduce created [18.02.2016 14:01:49]\n".
- "Thread THREAD_NAME terminated [DD.MM.YYYY HH:MM:SS]\n" is created when a thread terminates, where THREAD_NAME is the name of the function as described above.
  "Thread ExecReduce terminated [18.02.2016 13:05:00]\n".
  Note: this message may be printed from the thread itself or from a function that terminates the thread, depending on your design.
- [Updated, 11/4/15] "Map and Shuffle took ELASPED_TIME ns**\n**" prints the time (in nano-seconds) that was spent during the Map and Shuffle functions. This is practically the time from the beginning of the *runMapReduceFramework* execution and until the beginning of the Reduce function. The elapsed time can be measured in the same manner as you did in EX1. This line should be added to the log after all the creation and termination of the function has been added (immediacy before the reduce time, see next bullet).
- [Updated, 11/4/15] "Reduce took ELASPED_TIME ns**\n**" prints the time (in nano-seconds) that was spent during Reduce function and the final output production. This is practically the time from beginning of the Reduce execution and until the end of the *runMapReduceFramework* function. The time can be measured in the same manner as you did in EX1.
  This line must be the last line that is written to the log in each call to *runMapReduceFramework*.
- "runMapReduceFramework finished\n" is printed before the end of the runMapReduceFramework.

Notes relating to the log file

- While in general we tried to minimize locking mutexes that are shared between the *ExecMaps**ExecReduce* in order to increase the parallelism, here we will use the simplest approach --- you may protect the log using a single, shared mutex. This is fine because we write to the log only twice for each thread.
- In CLion, the working directory is an internal folder. For example: /tmp/.clion-login/system/cmake/generated/3da687e1/3da687e1/Debug/v3. You can get the working directory using ***getcwd***. I suggest you use a full path while working with CLion, **but don't forget to change it before submitting your exercise.**

Error Handling

While in a real Framework all the inputs must be checked, in this exercise you may assume that all the inputs are valid. For example, the *multiThreadLevel* is a positive number, the list of <k1,v1> is not null, you don't receive a null key in the *emit2*,*emit3* function, etc.

However, you must check if there was a failure in each system call that you use. Actually, almost each library function of C, C++ may fail, and you must check that the function is successful before continuing. For example, new, read, open, pthread_create, etc.

Handling a failure in multi-threaded environment is not easy. Assuming that a thread failed, how could it notify the main thread that a system call failed?
Anyway, we will use the simplest solution to handle a failure. You should print to the standard error:
"MapReduceFramework Failure: FUNCTION_NAME failed.", where FUNCTION_NAME is the name of the library call that was failed [e.g. "new"].
After each failure the program should be terminated (use *exit(1)*).

Requirements and tips

- You must use the pthread library, including *pthread_mutex_t,* and *pthread_cond_t* as we did in the class. It is forbidden to use (advanced) c++'s locks. You are also not allowed to use pipes, user level threads nor forks.
- Don't change the supplied header files, you don't submit them.
- We didn't intentionally force you to use specific data structures, except those that were written explicitly in the header files (in order to have the same interface to all the exercises). For example, we said "list" to describe a container. You may create array, vector, queue, or any other container that you like.
- To run multi-threads in CLion, you will need to add the following line to the CMake file:
  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -pthread")
- In order to create a correct comparison between keys, the user must implement its own *operator<* for each key, and the MapReduce framework should use these functions to compare between keys. Pay attention that STL's containers use the *operator<* to sort objects.
- The *Search* program is the most basic & simple test for your framework. However, it's not enough to test the framework properly. You are expected to create additional (more complicated) clients.

Theoretical part – 15 points

This section contains theoretical questions that should be answered in the README file. Pay attention that this equals to 1.5 points of your final grade (more than a complete exercise in many courses with theoretical exercises). This is an integral part of the exercise and therefore these questions (with variants) might be asked in the interviews.

1. In the MapReduceFramework, you used conditional variable and c++'s containers to synchronize between the *ExecMap* and the *Shuffle.* An alternative approach is to use *pipe* and *select.* How would you implement it? You are required to propose a design, which includes at least answers to the following questions:
    - Which threads use *select*, which threads *write* to the pipe and which *read* from the pipe
    - What data is written to the pipe (writing objects may not be easy)
    - Which data-structures\containers would you use?
    - What would happen at the end (e.g. how does the consumer of the pipe know that there is no more data).

    Tip: pay attention that the file descriptors table is per process, not per thread.
2. Assuming that a user wants to run the MapReduceFramework on his personal computer, which contains octa-cores (8 core) but without Hyper-threading support. What *multiThreadLevel* would you use to optimize the performance?
    A good solution contains detailed explanation.
    An excellent solution might be based on measurements.
3. Nira, Moti, Danny and Galit decided to implement the requirements of the MapReduceFramework as described in this exercise.
    However, each one of them implemented the concurrency slightly different:
    a. Nira used a single thread and a single process (single flow like “regular” program, no concurrency).
    b. Moti used Posix's library as required
    c. Danny used the user level threads that he implemented in exercise 2
    d. Galit didn’t understand the concept of threads, and therefore decided to use multi-processes rather than multi-threads (this means that instead of creating a thread, she created a process).

    You are required to compare the following attributes for each one of them:

    a. Utilizing multi-cores
    b. The ability to create a sophisticated scheduler, based on internal data.
    c. Communication time (between different threads/processes)
    d. Ability to progress while a certain thread/process is blocked (e.g. is waiting for disk)
    e. Overall speed
4. For a kernel level thread, a user-level thread and a process, what is shared for a parent and its child from the following:
    a. Stack
    b. Heap
    c. Global variables.
5. What is the difference between livelock and deadlock?
    Give an example for each one of them.

6. Assuming we have the following processes:

Process	Arrival time	running Time	Priority (higher number- higher priority)
P1	0	10	1
P2	1	1	3
P3	3	2	2
P4	7	12	3
P5	8	1	1

You are required to calculate the **Gantt chart**, **turnaround time** and the **average wait time** for the following schedulers:

1. Round Robin (RR) with quantum=2
2. First Come First Serve (FCFS)
3. Shortest Remaining Time First (SRTF)
4. Priority Scheduling

Submission

- You are required to submit the following files:
  - README, contains short explanations of your design, and answers for the theoretical questions.
  - [Updated, 5/4/15] Four images of the Gantt chart (theoretical part, question 6)**.**
    The images' type must be JPEG and the suffix of their name must be ".jpg".
    For example: "*RoundRobinGanttChart.jpg*"
  - MapReduceFramework.cpp, contains your implementation of the MapReduceFramework.
  - Search.cpp, contains your implementation of the search program.
  - Makefile. A "make" command will produce an executable file named **Search**, as well as **MapReduceFramework.a** which is a library that contains your implementation for the MapReduceFramework.
    - Tip – check that MapReduceFramework.a works before submitting it!

  You are allowed to submit additional headers and source files. You are not allowed to submit folders. Note that you should not submit the headers that we supplied (MapReduceFramework.h, MapReduceClient.h). Your program should work with our headers.

- [Updated, 5/4/15] Deadline policy: the deadline is 8-May (Sunday), 23:59. Besides the Pesach vacation, you have three weeks to work on that. Start Early and enjoy your vacation!
  The delays-penalty policy is as following:
  - Until 9-May-23:59: -5 points
  - Until 10-May-12:00: -10 points
  - Until 13-May-12:00: -25 points

- Until 15-May-23:59: -40 points
- After that: -100. Course failure.

and