Contents

1 README

```
ransha
Ran Shaham (203781000)
EX: 3

FILES:
README            -- This file
MapReduceFramework.cpp    -- My implementation of the uthreads library.
Search.cpp        -- The test (Task 1) for the library.
Makefile        -- Creates the library and 'Search' when called with
                no arguments
my_pthread.h        -- Wrapper for some of the pthread functions to handle
                unsuccessful pthread calls
error_handle.h        -- A header that includes the function that handles
                errors in this exercise
RR.jpg            -- Gantt chart for question 6 - Round Robin
FCFS.jpg        -- " - First Come First Serve
SRTF.jpg        -- " - Shortest Remaining Time First
Priority.jpg        -- " - Priority Scheduling


REMARKS:
Framework Design:
Most of the design was driven directly from the exercise description, except
for the following:
* The ExecMap output data structures are vectors, which allow the shuffle to
  read from them while the map writes (push_back) to them.
* Synchronization between the map and the shuffle is done by signaling the
  shuffle thread when a new chunk is written.
* The shuffle keeps a vector of indices. Each entry specifies the last index
  in each ExecMap vector that has been written, and DOES NOT pop items from
  the map's vector. After reading, it updates the index for that container.

Search.cpp Design:
First I'll describe the k1-k3,v1-v3 types I chose-
* The <k1,v1> pairs are the directory names and the substring to search
  respectively. This way, on each call to the Map function, the given directory
  is opened (it is written in the forums that we can assume that each folder
  will appear only once in the arguments), iterated over and searched for file
  names that contain the substring.
* The k2 is the filename that contains the substring. That is, a class that
  has a string member that holds the filename. The v2's only purpose is to
  free allocated k2 pointers. It takes a k2* as an argument and when destructed
  it also deletes the k2 that it holds. The idea is that every v2 created has
  its own k2, and the shuffle can "ignore" (merge) several <k2,v2> pairs to:
  <k2, list<v2>> pairs - i.e. no v2 is ever ignored. Therefore when the v2s
  are destroyed, all their k2s can be safely destroyed, and that happens
  implicitly when calling to "delete v2" (destructor). This way all of the
  <k2,v2> allocated memory is freed.
* k3 is the same as k2 (filename), and v3 is simply the number of times the
  file name that is held in k3 appeared in the input directories. For example,
  if the directory structre is this:

  root    -> dir1    -> a, b, c, .vimrc
    -> dir2    -> d, a, e
    -> dir3    -> Makefile, README, test.o, a
    -> dir4    -> .ssh, .config, Games

  then calling "Search a dir1 dir2 dir3" will yield [after reduce] the <k3,v3>
  entry: <"a",3> (among the other entries - <"e",1>, <"b",1> etc.).
```

Next, the Map/Reduce operations:
* Map - Gets a <dirname, strToFind> pair. It checks whether the 'dirname' is a valid directory (and if it exists) - if not it returns. Otherwise, it opens the directory and for each entry (file/link/dir) checks if the 'strToFind' is a substring of that entry's name. If so, it creates the <filename, v2> pair using (using the 'new' operator) and sends it to Emit2.
* Reduce - Gets a <filename, list<v2>> pair. It copies the filename to the k3 entry, and sets the value of v3 (filename count) to the size of the list of v2s. This works because the shuffle merges all files with the same name while keeping the v2s in a list for that filename, so the lise size is exactly the number of occurunces of the filename.
  Then, it iterates over the input list and delete the v2* values (and k2, implicitly), since they are no longer needed for the framework.

Having said that, the Search procedure is as follows:

1. Prepare the input - convert the input string arguments to the string to search and the directory names, and create a list of <k1*, v1*> pairs.
2. Run the framework and get the output <filename,count> list.
3. For each list entry of <file_i, count_i>, print file_i count_i times.
4. Iterate over the input and output lists and delete allocated memory

ANSWERS:

Q1:
The program will work with multiple processes (instead of threads).
Each process
* The process using select is the Shuffle, since it waits for data to be written

Q2:
Since the user's computer can run 8 threads concurrently - 1 on each core, multiThreadLevel should be at least 7 which will yield 8 threads running at the same time: 7 ExecMap threads + 1 shuffle thread.
Since the computer doesn't support hyper-threading, any additional thread will wait till a thread that was created earlier will finish running, therefore it is redundant. E.g., say multiThreadLevel is 10, then 8 ExecMap threads are created and being run and the remaining 2 WAIT until 2 of the earlier ones finish running. When they do, by definition there is no more mapping to be done so they are immidiately terminated. Then, after the mapping is finished can a shuffle be created and only start shuffling.
On the other hand, less than 7 won't utilze all system resources - after creating 4 (for example) ExecMap threads the main thread then creates one shuffle thread and waits for them to finish. So there will be 5 working thread untill the shuffle is done and 2 cores will remain unused this entire time (lower CPU utilization).
In conclusion, I'd suggest to set multiThreadLevel to 7.

Q3:
a. Nira's solution - Single thread
* Does not utilze multi-cores since only one thread of execution is running.
* No scheduler is needed since no concurrency is happening, the program flow is serial.
* No communication is needed since there's a single thread and process.
* When waiting for I/O operations or other time consuming actions, the program does not progress
* The overall speed is probably slower than the other methods, since the process can spend a lot of time waiting (for disk access when writing the log, for example) and cannot do anything in this time.

b. Moti - POSIX's library
* Utilizes multi-cores since the OS is aware of kernel-level threads and can run a thread for each core or processor concurrently.
* The library manages its own scheduling methods so besides block / signal there is no much scheduling that can be done.
* Communication time is relatively fast since the threads share heap and static data segments, and can read from them concurrently. Writing requires locking shared resources, but it is still better than multi-process communication.

* When one thread is blocked, another can proceed, although when accessing disk
  no concurrency can occur (that is, two threads cannot acess the disk in the
  same time).
* The overall speed depends on whether the machine running the framework has
  multiple cores or not. If not, there is no advantage for this method over
  user-level threads, and the overhead of context switching, thread creation
  and termination is bigger than in user-level threads.
  If the machine has multiple cores or processes, this method allows more than
  one thread running in the same time, thus is probably faster than the other
  methods.

c. Danny - User-level threads
* Since the OS is not aware of the multiple threads running, it does not
  run threads on multiple cores concurrently, so this method does not utilize
  multiple cores.
* This is THE method for creating scheduler based on internal data - the user
  can do whatever he/she wants, and has access to all data in the program, so
  this method is very flexible in terms of scheduling.
* Communication time is similar to kernel-level threads. If no built-in OS
  mutex types are used (which gurantee atomic operations) synchronization can
  be more tricky than in kernel-level.
* The OS is not aware of the fact multiple threads are running, so when the
  process makes a system call, the OS is likely to block it, therefore not
  allowing other user-level threads to run in the meantime. If the OS does
  not block the entire process, then the program can progress on another
  thread while another one waits.
* This is likely to be a fast implementation on single core machines, since
  context switches and thread actions require mainly function calls, and not
  system calls, therefore overhead time is reduced. On multiple cores, no
  concurrency is involved so this method is likely to be slower than methods
  that enable concurrency (multi-process or kernel-level threads).

d. Galit - Multi-processes
* This method utilizes multi-cores since modern OSs tend to do so with
  multiple processes.
* The scheduling is almost entirely up to the OS to decide - there are some
  hints that can affect the scheduler (nice values, for example) but are based
  on general process role, and not fine-grained internal data.
* Communication is trickier than in other methods - the processes need pipes
  or files to communicate and handling files require sys-calls, which are slow
  as we measured in EX1.
* Each process is more or less independent, so while one is blocked another can
  proceed (if it should, logically).
* In my opinion (i.e - based on my knowledge so far, not experience) this
  method will be slower than the others on single core machines, and perhaps
  a bit faster on multi-core than the single-thread or user-level threads
  methods, but slower than the kernel-level threads.

Q4:
Processes:
a. Stack is not shared, each process gets its own stack.
b. Same for the heap.
c. Same for global variables.
When using fork() sys call, the above data segments are copied to the child
process, not shared.

Kernel-level threads:
a. Each thread gets its own stack
b. Threads share heap data, since they reside in the same process
c. Same as heap, the process' static data segment is the same for all threads

User-level threads:
Exactly the same as ernel-level threads.
A stack can't be shared between threads since they run different functions and
execute a different set of commands (perhaps), so each thread should have its
own stack for local variables and adresses.
The heap and static data segments are shared, since the threads are run in a
one process.

Q5:
A deadlock is a situation in which two or more concurrent running threads of execution (processes/ threads) try to access some shared resources; each of them blocks another and is blocked by another, thus neither of them progresses with their execution.
For example, threads A,B both need resources x,y to perform an operation. Consider the following scenerio: A runs, reserves x; then, a context switch occurs and B reserves y. If it tries to reserve x it will be blocked. If A tries to reserve y it will also be blocked. Therefore they both block each other and neither progresses.

A livelock is also a situation in which two or more running threads don't proceed with their execution. In this case, all threads are taking active actions in order to let the other threads finish their work, and then proceed. If each thread lets the other threads proceed and no thread actually proceeds, a livelock occurs.
For example, say threads A,B need resources x,y as before. Again, A reserves x and B reserves y. Now B tries to reserve x and fails so it lets go of y and waits 1 second. Before it does, A gets the control and tries to access y, which is still in B's control, so it lets go of x and waits for 1 second. Then, they try again using the same method. This creates the above scenerio, where 2 threads aren't progressing but both are taking active actions to prevent locks.

Q6:

# NAME	Avg. wait time (format: (P1:a+b+... + P2:c+d+... + ...) / 5 = result
1. RR	(1+2+3+2 + 1 + 2 + 2+3+2 + 3) / 5 = 4.2
2. FCFS	(0 + (10-1) + (11-3) + (13-7) + (25-8)) / 5 = 8
3. SRTF	(1+2+1 + 0 + 0 + 7 + 0) / 5 = 2.2
4. PRIO	(0 + (10-1) + (23-3) + (11-7) + (25-8)) / 5 = 10

# NAME	Turnaround time
1. RR	(18 + (3-1) + (7-3) + (26-7) + (12-8)) / 5 = 9.4
2. FCFS	(10 + (11-1) + (13-3) + (25-7) + (26-8)) / 5 = 13.2
3. SRTF	(14 + (2-1) + (5-3) + (26-7) + (9-8)) / 5 = 7.4
4. PRIO	(10 + (11-1) + (25-3) + (23-7) + (26-8)) / 5 = 15.2

* (All time calculations are ignoring context switching)

2 FCFS.jpg

3 Makefile

```
CFLAGS=-std=c++11 -Wall -Wextra -g -pthread $(INCS)

# cpp to object files rule
%.o: %.cpp
    $(CXX) $(CFLAGS) -c $<

# Library Stuff
INCS=-I.
LOADLIBES=-L.

LIBSRC=MapReduceFramework.cpp
LIBH=MapReduceFramework.h
LIBOBJ=MapReduceFramework.o
ARFLAGS=rvs
RANLIB=ranlib

LIBTARGET=MapReduceFramework.a

# test rules
TEST_SRC=Search.cpp
TEST_FILE=Search
VALGRIND_FLAGS = --leak-check=full --show-possibly-lost=yes \
         --show-reachable=yes --undef-value-errors=yes

$(TEST_FILE): $(TEST_SRC) $(LIBTARGET)
    $(CXX) $(CFLAGS) $(LOADLIBES) $^ -o $@

ValgrindTest: $(TEST_OBJ)
    valgrind $(VALGRIND_FLAGS) ./$<


# library rules
$(LIBTARGET): $(LIBOBJ)
    $(AR) $(ARFLAGS) $@ $^
    $(RANLIB) $@

$(LIBOBJ): $(LIBSRC)
    $(CXX) $(CFLAGS) -c $< -o $(LIBOBJ)

# cleaning and such
RM=rm -fv
LOG_FILE=.MapReduceFramework.log
clean:
    $(RM) $(TEST_FILE) $(LIBTARGET) *.o $(LOG_FILE)

depend:
    makedepend -- $(CFLAGS) -- $(SRC) $(LIBSRC)

# tar rule
TAR=tar
TARFLAGS=-cvf
TARNAME=ex3.tar
EXTRA_HEADERS=my_pthread.h error_handle.h
GANTT_PICS=RR.jpg FCFS.jpg SRTF.jpg Priority.jpg
TARSRCS=$(LIBSRC) $(TEST_SRC) Makefile README $(EXTRA_HEADERS) $(GANTT_PICS)

tar:
    $(TAR) $(TARFLAGS) $(TARNAME) $(TARSRCS)

```

```
60  all: $(TEST_FILE)
61
62  .PHONY: all clean tar ValgrindTest
```

4 MapReduceFramework.cpp

```cpp
/* == Includes == */
#include "MapReduceFramework.h"

#include "my_pthread.h"        /* inlcudes pthread.h + error handling */
#include <fstream>
#include <vector>
#include <queue>
#include <map>

#include <cmath>          /* for fmin */
#include <sys/time.h>         /* gettimeofday */
#include <ctime>         /* for strftime */
#include <chrono>         /* time measurements */
#include <algorithm>         /* for std::move */

/* To reduce line-length and increase readability */
using std::vector;
using std::queue;
using std::map;
using std::multimap;
using std::list;
using std::pair;

/* == Some constants == */
#define CHUNK_SIZE 10
#define USEC_TO_NSEC(x) ((x) * 1000)
#define NANO_IN_SEC 1000000000
#define TIME_TO_WAIT 1000000

#define TIME_FORMAT "[%d.%m.%Y %T]"

#define LOG_FILENAME ".MapReduceFramework.log"
#define LOG_OPEN_MODE std::ofstream::out | std::ofstream::app

#define THREAD_EXECMAP "ExecMap"
#define THREAD_SHUFFLE "Shuffle"
#define THREAD_EXECREDUCE "ExecReduce"

#define MSG_FRAMEWORK_START(t) "runMapReduceFramework started with " << t \
    << " threads"
#define MSG_FRAMEWORK_END "runMapReduceFramework finished"
#define MSG_THREAD_CREATED(th) "Thread " th " created "
#define MSG_THREAD_TERMINATED(th) "Thread " th " terminated "
#define MSG_MAP_SHUFFLE_TIME(t) "Map and Shuffle took " << t << " ns"
#define MSG_REDUCE_TIME(t) "Reduce took " << t << " ns"

/* Struct for the map procedure. nextToRead is the only variable that *
 * will be mutexed, as specified in the instructions. */
typedef struct MapData
{
    MapReduceBase &mapReduce;
    vector<IN_ITEM> &itemsList;
    size_t numOfItems;
    size_t nextToRead;
    std::ofstream &logofs;
} MapData;

/* Same for reduce, the items list is static and therefore not in here. */
typedef struct ReduceData
```

```
{
    MapReduceBase &mapReduce;
    size_t threadNum;
    size_t numOfItems;
    size_t nextToRead;
    std::ofstream &logofs;
} ReduceData;

/* Comparator to sort keys by values, instead of adresses */
template<class T> struct ptr_less
{
    bool operator()(T* lhs, T* rhs)
    {
    return *lhs < *rhs;
    }
};

/* Threads variables */
vector<pthread_t> threadList; /* A list of existing ExecMap/Reduce threads */
pthread_t shuffleThread;      /* The shuffle thread */

/* ExecMap variables */
typedef vector<pair<k2Base*, v2Base*>> K2_V2_LIST;
/* a list of pairs in which the first element is the pthread_t (id) and *
 * the second is that thread's list (pointer) of the map-function-outputs */
vector<pair<pthread_t, K2_V2_LIST*>> mapResultLists;

/* Shuffle Variables */
/* An array of indices that specify till what index in the map-lists *
 * the shuffle thread has already read */
vector<size_t> shuffleCurrIndex;
/* Note that the map compares keys using ptr_less defined above */
map<k2Base*, V2_LIST*, ptr_less<k2Base>> shuffleOut;

/* Reduce Variables */
vector<pair<k2Base*, V2_LIST*>> reduceIn;
typedef queue<pair<k3Base*, v3Base*>> K3_V3_LIST;
/* Same as before, but for the reduce-function-outputs */
vector    <pair<pthread_t, K3_V3_LIST*>> reduceResultLists;
multimap<k3Base*, v3Base*, ptr_less<k3Base>> finalOutputMap;

/* Mutexes */
pthread_mutex_t mutex_map_read;        /* protect input list */
pthread_mutex_t mutex_em_map;        /* barrier to init maps */
pthread_mutex_t mutex_more_to_shuffle;    /* condition of shuffle loop */
pthread_mutex_t mutex_reduce_read;    /* protect reduce input */
pthread_mutex_t mutex_log_write;    /* protect writing to log */
/* Condition Variables */
pthread_cond_t cv_more_to_shuffle;
bool more_to_shuffle;


/**
 * Gets the current time as a string in the exercise format. i.e
 * "[DD.MM.YYYY HH:MM:SS]"
 */
string getTimeStr()
{
    size_t MAX_CHARS = 64;
    time_t rawtime;
    struct tm * timeinfo;

    time(&rawtime);
    timeinfo = localtime(&rawtime);
    char buff[MAX_CHARS];

    strftime(buff, MAX_CHARS, TIME_FORMAT, timeinfo);
    return buff;
```

```
128 }
129
130
131 /**
132  * Initializes all mutexes, cv and relevant static variables for the library.
133  */
134 void init(int multiThreadLevel)
135 {
136     // Pthread types
137     my_pthread_mutex_init(&mutex_map_read, nullptr);
138     my_pthread_mutex_init(&mutex_em_map, nullptr);
139     my_pthread_mutex_init(&mutex_more_to_shuffle, nullptr);
140     my_pthread_cond_init(&cv_more_to_shuffle, nullptr);
141     my_pthread_mutex_init(&mutex_reduce_read, nullptr);
142     my_pthread_mutex_init(&mutex_log_write, nullptr);
143     // Variables and data structures
144     more_to_shuffle = true;
145     threadList.resize(multiThreadLevel);
146     mapResultLists.resize(multiThreadLevel);
147     shuffleCurrIndex.resize(multiThreadLevel);
148     reduceResultLists.resize(multiThreadLevel);
149 }
150
151 /**
152  * Destroys mutexes, cv and clear the static data structures in use.
153  */
154 void cleanup()
155 {
156     // Pthread types
157     my_pthread_mutex_destroy(&mutex_map_read);
158     my_pthread_mutex_destroy(&mutex_em_map);
159     my_pthread_mutex_destroy(&mutex_more_to_shuffle);
160     my_pthread_cond_destroy(&cv_more_to_shuffle);
161     my_pthread_mutex_destroy(&mutex_reduce_read);
162     my_pthread_mutex_destroy(&mutex_log_write);
163     // heap-allocated data delete
164     for (size_t i = 0; i < mapResultLists.size(); ++i)
165     {
166         delete mapResultLists[i].second;
167         delete reduceResultLists[i].second;
168         mapResultLists[i].second = nullptr;
169         reduceResultLists[i].second = nullptr;
170     }
171     for (auto it = reduceIn.begin(); it != reduceIn.end(); ++it)
172     {
173         delete it->second;
174         it->second = nullptr;
175     }
176     // Variable & data structures reset
177     shuffleThread = 0;
178     threadList.clear();
179     mapResultLists.clear();
180     reduceResultLists.clear();
181     shuffleCurrIndex.clear();
182     shuffleOut.clear();
183     reduceIn.clear();
184     finalOutputMap.clear();
185 }
186
187 /**
188  * The ExecMap procedure: Reads data from the input <k1,v1> list and uses
189  * the user provided Map function to create a list of <k2,v2> values (for
190  * each thread running this function).
191  * Takes a MapData struct pointer as an argument.
192  */
193 void* ExecMap(void *arg)
194 {
195     /* A barrier, opens when all ExecMap data structures are *
```

```
196         * initialized */
197       my_pthread_mutex_lock(&mutex_em_map);
198       my_pthread_mutex_unlock(&mutex_em_map);
199       MapData* mapdata = (MapData*) arg;
200       size_t myItems; // The start index for this thread's data
201       // While there's more data to read, read a chunk and map.
202       while ((myItems = mapdata->nextToRead) < mapdata->numOfItems)
203       {
204           // Lock the index counter
205           my_pthread_mutex_lock(&mutex_map_read);
206           // Reserve items to map and get the start index of them.
207           myItems = (mapdata->nextToRead += CHUNK_SIZE) - CHUNK_SIZE;
208           // Unlock
209           my_pthread_mutex_unlock(&mutex_map_read);
210           // Map this thread's data
211           int end = fmin(myItems + CHUNK_SIZE, mapdata->numOfItems);
212           for (int i=myItems; i < end; ++i)
213           {
214               mapdata->mapReduce.Map(mapdata->itemsList.at(i).first,
215                           mapdata->itemsList.at(i).second);
216           }
217           // Notify the shuffle thread
218           my_pthread_cond_signal(&cv_more_to_shuffle);
219       }
220       my_pthread_mutex_lock(&mutex_log_write);
221       mapdata->logofs << MSG_THREAD_TERMINATED(THREAD_EXECMAP)
222           << getTimeStr() << endl;
223       my_pthread_mutex_unlock(&mutex_log_write);
224       return nullptr;
225   }
226
227
228   /**
229    * The Shuffle procedure: Merge <k2,v2> pairs from the ExecMap containers
230    * to <k2, queue<v2>> containers.
231    */
232   void* Shuffle(void *)
233   {
234       struct timespec timeToWake;
235       struct timeval timeNow;
236       pair<k2Base*, v2Base*> currPair;
237       k2Base* currKey;
238       v2Base* currVal;
239       auto foundList = shuffleOut.end();
240       V2_LIST* newList = nullptr;
241       K2_V2_LIST* currList = nullptr;
242       size_t currIdx = 0;
243
244       my_pthread_mutex_lock(&mutex_more_to_shuffle);
245       while (more_to_shuffle)
246       {
247           if (gettimeofday(&timeNow, nullptr) < 0)
248           {
249               handleError("gettimeofday");
250           }
251           timeToWake.tv_sec = timeNow.tv_sec +
252               (USEC_TO_NSEC(timeNow.tv_usec) + TIME_TO_WAIT) /
253                           NANO_IN_SEC;
254           timeToWake.tv_nsec = fmod(USEC_TO_NSEC(timeNow.tv_usec) +
255               TIME_TO_WAIT, NANO_IN_SEC);
256           // Wait for data (or time passing)
257           my_pthread_cond_timedwait(&cv_more_to_shuffle,
258                   &mutex_more_to_shuffle, &timeToWake);
259           // Check every ExecMap container for items to shuffle
260           for (size_t i = 0; i < mapResultLists.size(); ++i)
261           {
262               currList = mapResultLists[i].second;
263               // Read all unread data and shuffle it.
```

```
264             for (currIdx = shuffleCurrIndex[i];
265                  currList != nullptr && currIdx < currList->size();
266                  ++currIdx)
267             {
268                 currPair = currList->at(currIdx);
269                 currKey = currPair.first;
270                 currVal = currPair.second;
271                 foundList = shuffleOut.find(currKey);
272                 // If no k2 entry exists, create it.
273                 if (foundList == shuffleOut.end())
274                 {
275                     newList = new(std::nothrow) V2_LIST;
276                     // If new operator has failed
277                     if (newList == nullptr)
278                     {
279                         handleError("new operator");
280                     }
281                     newList->push_back(currVal);
282                     shuffleOut.insert(std::make_pair(
283                                 currKey,
284                                 newList));
285                     newList = nullptr;
286                 }
287                 else
288                 {
289                     foundList->second->push_back(currVal);
290                 }
291             }
292             // 'Mark' all read data as read
293             shuffleCurrIndex[i] = currIdx;
294         }
295     }
296     my_pthread_mutex_unlock(&mutex_more_to_shuffle);
297     return nullptr;
298 }
299
300
301 /**
302  * The Reduce procedure: Read input from a data structure and run the reduce
303  * function on it to produce the output data
304  */
305 void* ExecReduce(void *arg)
306 {
307     ReduceData* reducedata = (ReduceData*) arg;
308     // Create this thread's data structure (list of k3,v3)
309     size_t threadNum = reducedata->threadNum;
310     pthread_t tid = pthread_self();
311     if (!pthread_equal(reduceResultLists[threadNum].first, tid))
312     {
313         reduceResultLists[threadNum] = std::make_pair(
314          tid, new(std::nothrow) K3_V3_LIST);
315         if (reduceResultLists[threadNum].second == nullptr)
316         {
317             handleError("new operator");
318         }
319     }
320     size_t myItems; // The start index for this thread's data
321     // While there's more data to read, read a chunk
322     while ((myItems = reducedata->nextToRead) < reducedata->numOfItems)
323     {
324         // Lock the index counter
325         my_pthread_mutex_lock(&mutex_reduce_read);
326         // Reserve items to map and get the start index of them.
327         myItems = (reducedata->nextToRead += CHUNK_SIZE) - CHUNK_SIZE;
328         // Unlock
329         my_pthread_mutex_unlock(&mutex_reduce_read);
330         int end = fmin(myItems + CHUNK_SIZE, reducedata->numOfItems);
331         for (int i = myItems; i < end; ++i)
```

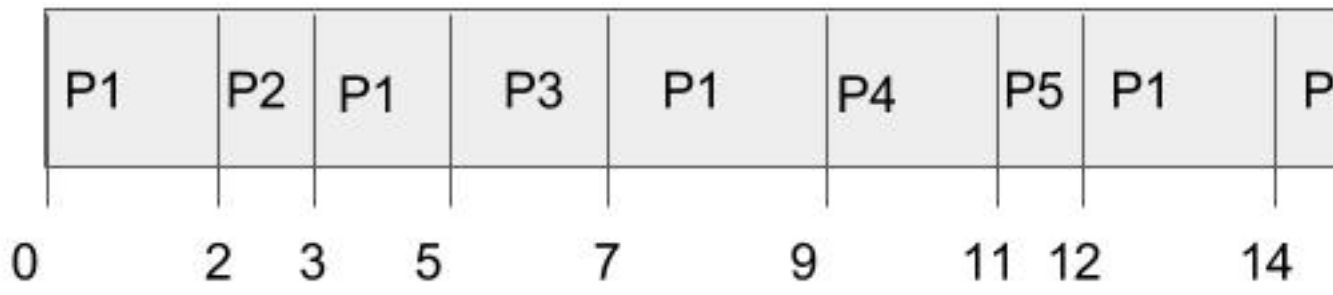
```
332         {
333             reducedata->mapReduce.Reduce( reduceIn.at(i).first,
334                             *reduceIn.at(i).second);
335         }
336     }
337     my_pthread_mutex_lock(&mutex_log_write);
338     reducedata->logofs << MSG_THREAD_TERMINATED(THREAD_EXECREDUCE)
339         << getTimeStr() << endl;
340     my_pthread_mutex_unlock(&mutex_log_write);
341     return nullptr;
342 }
343
344
345 /**
346  * Transforms the many reduce-thread-data-structures to one big map.
347  * Same as shuffle, but for reduce.
348  */
349 void finalize_output()
350 {
351     pair<k3Base*, v3Base*> currPair;
352     for (size_t i = 0; i < reduceResultLists.size(); ++i)
353     {
354         while( reduceResultLists[i].second != nullptr &&
355               !reduceResultLists[i].second->empty() )
356         {
357             currPair = reduceResultLists[i].second->front();
358             reduceResultLists[i].second->pop();
359             finalOutputMap.insert(currPair);
360         }
361     }
362 }
363
364
365 /**
366  * The main framework function.
367  */
368 OUT_ITEMS_LIST runMapReduceFramework(MapReduceBase &mapReduce,
369         IN_ITEMS_LIST &itemsList, int multiThreadLevel)
370 {
371     // open the log file, write and start measuring time
372     std::ofstream ofs(LOG_FILENAME, LOG_OPEN_MODE);
373     ofs << MSG_FRAMEWORK_START(multiThreadLevel) << endl;
374     std::chrono::high_resolution_clock::time_point t1 =
375         std::chrono::high_resolution_clock::now();
376
377     // Initialize mutexes, cvs, datastructures...
378     init(multiThreadLevel);
379     // Turn the in_items_list into a vector for random access
380     vector<IN_ITEM> itemsVector( std::begin(itemsList),
381                     std::end(itemsList)  );
382
383     MapData mapdata = {.mapReduce = mapReduce, .itemsList = itemsVector,
384         .numOfItems = itemsVector.size(),
385         .nextToRead = 0, .logofs = ofs};
386
387     // Create the ExecMap threads!
388     my_pthread_mutex_lock(&mutex_em_map); // Barrier to prevent sigsegv
389     for (int i = 0; i < multiThreadLevel; ++i)
390     {
391         my_pthread_mutex_lock(&mutex_log_write);
392         ofs << MSG_THREAD_CREATED(THREAD_EXECMAP) << getTimeStr()
393             << endl;
394         my_pthread_mutex_unlock(&mutex_log_write);
395         my_pthread_create(&threadList[i], nullptr, &ExecMap, &mapdata);
396     }
397     // Initialize all threads' maps
398     for (int i = 0; i < multiThreadLevel; ++i)
399     {
```

```
400             mapResultLists[i] = std::make_pair(threadList[i],
401                     new(std::nothrow) K2_V2_LIST);
402             if (mapResultLists[i].second == nullptr)
403             {
404                 handleError("new operator");
405             }
406         }
407         my_pthread_mutex_unlock(&mutex_em_map); // Remove the barrier
408
409         // Create the shuffle thread
410         my_pthread_mutex_lock(&mutex_log_write);
411         ofs << MSG_THREAD_CREATED(THREAD_SHUFFLE) << getTimeStr() << endl;
412         my_pthread_mutex_unlock(&mutex_log_write);
413         my_pthread_create(&shuffleThread, nullptr, &Shuffle, nullptr);
414
415         // Wait for the ExecMaps to finish
416         for (int i = 0; i < multiThreadLevel; ++i)
417         {
418             my_pthread_join(threadList[i], nullptr);
419         }
420         // Wait for the shuffle to finish
421         my_pthread_mutex_lock(&mutex_more_to_shuffle);
422         more_to_shuffle = false;
423         my_pthread_mutex_unlock(&mutex_more_to_shuffle);
424         my_pthread_cond_signal(&cv_more_to_shuffle);
425         my_pthread_join(shuffleThread, nullptr);
426
427         // Stop time measuring
428         std::chrono::high_resolution_clock::time_point t2 =
429             std::chrono::high_resolution_clock::now();
430         auto mapShuffleDuration = std::chrono::duration_cast
431             <std::chrono::nanoseconds>(t2 - t1).count();
432         ofs << MSG_THREAD_TERMINATED(THREAD_SHUFFLE) << getTimeStr() << endl;
433
434         // Transform the shuffle output map to a vector.
435         reduceIn.resize(shuffleOut.size());
436         std::move(shuffleOut.begin(), shuffleOut.end(), reduceIn.begin());
437         // Initialize the reducedata struct
438         ReduceData reducedata = {.mapReduce = mapReduce, .threadNum = 0,
439             .numOfItems = reduceIn.size(), .nextToRead = 0, .logofs = ofs};
440
441         // Measure reduce time
442         t1 = std::chrono::high_resolution_clock::now();
443         // Reduce!
444         for (int i = 0; i < multiThreadLevel; ++i)
445         {
446             my_pthread_mutex_lock(&mutex_log_write);
447             ofs << MSG_THREAD_CREATED(THREAD_EXECREDUCE) << getTimeStr()
448                 << endl;
449             my_pthread_mutex_unlock(&mutex_log_write);
450             reducedata.threadNum = i;
451             my_pthread_create(&threadList[i], nullptr,
452                     &ExecReduce, &reducedata);
453         }
454         // Wait for reduce to finish
455         for (int i = 0; i < multiThreadLevel; ++i)
456         {
457             my_pthread_join(threadList[i], nullptr);
458         }
459         // "Shuffle" reduce's output!
460         finalize_output();
461         // Convert to output type and return.
462         OUT_ITEMS_LIST finalOutput(finalOutputMap.size());
463         std::move(finalOutputMap.begin(), finalOutputMap.end(),
464                 finalOutput.begin());
465         // Clean up
466         cleanup();
467
```

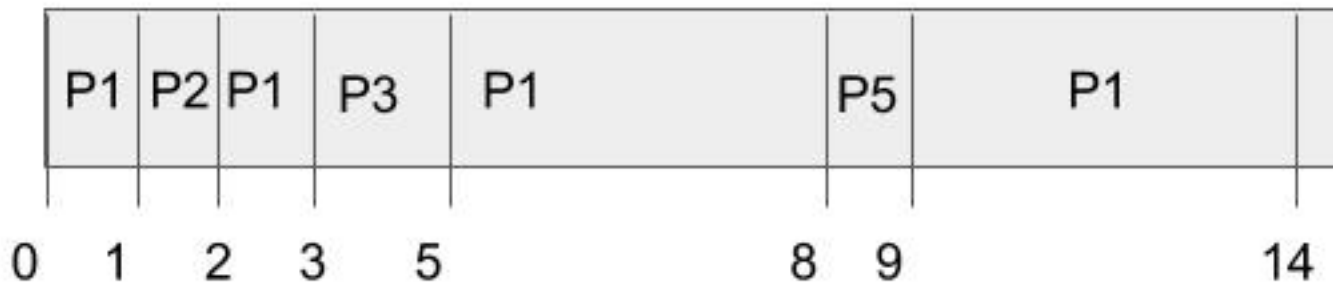
```
468     // Stop measuring time, write to log and return
469     t2 = std::chrono::high_resolution_clock::now();
470     auto reduceDuration = std::chrono::duration_cast
471         <std::chrono::nanoseconds>(t2 - t1).count();
472     ofs << MSG_MAP_SHUFFLE_TIME(mapShuffleDuration) << endl;
473     ofs << MSG_REDUCE_TIME(reduceDuration) << endl;
474     ofs << MSG_FRAMEWORK_END << endl;
475     return finalOutput;
476 }
477
478
479 /**
480  * Add <k2*,v2*> to the map-threads data structures
481  */
482 void Emit2(k2Base *key, v2Base *val)
483 {
484     // Figure out which thread is handling this function
485     pthread_t currThread;
486     currThread = pthread_self();
487     size_t end = mapResultLists.size();
488     // Add the given key,val pair to the correct list:
489     for (size_t i = 0; i < end; ++i)
490     {
491         if (pthread_equal(mapResultLists[i].first, currThread))
492         {
493             mapResultLists[i].second->push_back(
494                             std::make_pair(key, val) );
495             return;
496         }
497     }
498 }
499
500 /**
501  * Add <k3*,v3*> to the reduce-threads data structures
502  */
503 void Emit3(k3Base *key, v3Base *val)
504 {
505     pthread_t currThread;
506     currThread = pthread_self();
507     size_t end = reduceResultLists.size();
508     for (size_t i = 0; i < end; ++i)
509     {
510         if (pthread_equal(reduceResultLists[i].first, currThread))
511         {
512             reduceResultLists[i].second->push(
513                             std::make_pair(key, val));
514             return;
515         }
516     }
517 }
```

5 Priority.jpg

6 RR.jpg



7 SRTF.jpg



8 Search.cpp

```cpp
#include "MapReduceClient.h"
#include "MapReduceFramework.h"

#include <cerrno>
#include <string>
#include <vector>
#include <iostream>
#include <dirent.h>
#include <cstring>

#define ARG_STR_TO_FIND 1
#define ARG_DIR_START 2

#define EXIT_SUCC 0
#define EXIT_FAIL 1
#define THREAD_LEVEL 5
#define CURR_DIR "."
#define PARENT_DIR ".."

using std::string;

// Constants:
const int MIN_ARGS = 2;
const string USAGE_ERROR_MESSAGE = "Usage: <substring to search> "
    "<folders, separated by spaces>";


/**
 * Handle errors as required
 */
static void handleError(const string funcName)
{
    std::cerr << "MapReduceFramework Failure: " << funcName << " failed."
         << std::endl;
    exit(EXIT_FAIL);
}

/**
 * The directory name key (k1Derived)
 */
class DirNameKey : public k1Base
{
public:
    DirNameKey(const string d)  : dirName(d) {}

    virtual ~DirNameKey() {}

    virtual bool operator<(const k1Base &other) const override
    {
        const DirNameKey &dirother =
            dynamic_cast<const DirNameKey&> (other);
        return dirName < dirother.dirName;
    }

    const string getDirName() const
    {
        return dirName;
    }
private:
```

```
60      const string dirName;
61  };
62
63  /**
64   * The string to search in the file names
65   */
66  class StringToFind : public v1Base
67  {
68  public:
69      StringToFind(const string toFind) : str(toFind) {}
70
71      string getStr() const
72      {
73          return str;
74      }
75  private:
76      const string str;
77  };
78
79
80  /**
81   * The k2 class (k2Derived) - the filename (inside the input directories)
82   */
83  class FileNameKey1 : public k2Base
84  {
85  public:
86      FileNameKey1(const string f) : fileName(f) {}
87
88      virtual ~FileNameKey1(){}
89
90      virtual bool operator<(const k2Base &other) const override
91      {
92          const FileNameKey1 &fileOther =
93              dynamic_cast<const FileNameKey1&>(other);
94          return fileName < fileOther.fileName;
95      }
96
97      const string getFileName() const
98      {
99          return fileName;
100     }
101
102 private:
103     const string fileName;
104 };
105
106
107 /**
108  * The v2 class (v2Derived) - holds a pointer to the k2 for later
109  * deletion
110  */
111 class v2Deriv : public v2Base
112 {
113 public:
114     v2Deriv(FileNameKey1 *k2p) : k2Pointer(k2p) {}
115
116     virtual ~v2Deriv()
117     {
118         delete k2Pointer;
119         k2Pointer = nullptr;
120     }
121 private:
122     FileNameKey1 *k2Pointer;
123 };
124
125
126
127 /**
```

```
128  * The k3 class (k3Derived) - the filename (inside the input directories)
129  * This is the same as k2
130  */
131 class FileNameKey2 : public k3Base
132 {
133 public:
134     FileNameKey2(const string f) : fileName(f) {}
135 
136     virtual ~FileNameKey2(){}
137 
138     virtual bool operator<(const k3Base &other) const override
139     {
140         const FileNameKey2 &fileOther =
141             dynamic_cast<const FileNameKey2&>(other);
142         return fileName < fileOther.fileName;
143     }
144 
145     const string getFileName() const
146     {
147         return fileName;
148     }
149 
150 private:
151     const string fileName;
152 };
153 
154 /**
155  * The v3 class (v3Derived) - Holds the number of times the k3 filename
156  * appeared (in total)
157  */
158 class FileCountValue : public v3Base
159 {
160 public:
161 
162     FileCountValue(int count) : myCount(count) {}
163 
164     virtual ~FileCountValue() {}
165 
166     int getCount() const
167     {
168         return myCount;
169     }
170 
171 private:
172     const int myCount;
173 };
174 
175 /**
176  * The map and reduce functions implementation
177  */
178 class MyMapReduce : public MapReduceBase {
179 public:
180 
181     virtual void Map(const k1Base *const key, const v1Base *const val)
182         const override
183     {
184         // Downcast to k1*
185         const DirNameKey* const pdirkey = (const DirNameKey* const)key;
186         const StringToFind* const ptoFind =
187             (const StringToFind* const)val;
188 
189         // Open the directory and iterate over files in it
190         string dirName = pdirkey->getDirName();
191 
192         DIR *pdir = opendir(dirName.c_str());
193         if (pdir == nullptr)
194         {
195                 // Skip this file if it is not a directory.
```

```
196             if (errno == ENOTDIR || errno == ENOENT)
197             {
198                 return;
199             }
200             handleError("opendir");
201         }
202         struct dirent *pent = nullptr;
203         string currFile;
204         while ( (pent = readdir(pdir)) )
205         {
206             currFile = pent->d_name;
207             // Ignore '.' and '..' folders
208             if (currFile.compare(CURR_DIR) == 0 ||
209             currFile.compare(PARENT_DIR) == 0 ||
210                 // or if searched string was not found.
211             currFile.find(ptoFind->getStr()) == string::npos)
212             {
213                 continue;
214             }
215             FileNameKey1* pKey =
216                 new(std::nothrow) FileNameKey1(pent->d_name);
217             v2Deriv* pVal = new(std::nothrow) v2Deriv(pKey);
218             if (pKey == nullptr || pVal == nullptr)
219             {
220                 handleError("new");
221             }
222             Emit2(pKey, pVal);
223         }
224         if (closedir(pdir) < 0)
225         {
226             handleError("closedir");
227         }
228
229     }
230
231     virtual void Reduce(const k2Base *const key, const V2_LIST &vals)
232         const override
233     {
234         const FileNameKey1* const pfileName =
235             (const FileNameKey1* const)key;
236
237         FileNameKey2 *pKey =
238            new(std::nothrow) FileNameKey2(pfileName->getFileName());
239         FileCountValue *pVal =
240             new(std::nothrow) FileCountValue(vals.size());
241         if (pKey == nullptr || pVal == nullptr)
242         {
243             handleError("new");
244         }
245         Emit3(pKey, pVal);
246         // Delete the k2,v2 pairs
247         v2Deriv* currV2Item;
248         for (auto it = vals.begin(); it != vals.end(); ++it)
249         {
250             currV2Item = dynamic_cast<v2Deriv*>(*it);
251             delete currV2Item;
252             currV2Item = nullptr;
253         }
254     }
255 };
256
257
258 /**
259  * Delete allocated <k1,v1> <k3,v3> pairs from input/output lists
260  */
261 void cleanup(IN_ITEMS_LIST& inItems, OUT_ITEMS_LIST& outItems)
262 {
263     for (auto it = inItems.begin(); it != inItems.end(); ++it)
```

```
264         {
265             delete it->first;
266             delete it->second;
267         }
268         for (auto it = outItems.begin(); it != outItems.end(); ++it)
269         {
270             delete it->first;
271             delete it->second;
272         }
273     }
274
275     /**
276      * Runs the framework and prints the results
277      */
278     void runMapReduce(int numOfDirs, string strToFind,
279             std::vector<string>& dirName)
280     {
281         // Prepare the input
282         IN_ITEMS_LIST inItems;
283         MyMapReduce mapReduce;
284         std::pair<k1Base*, v1Base*> currPair;
285         for (int i = 0; i < numOfDirs; ++i)
286         {
287             currPair = std::make_pair(new DirNameKey(dirName[i]),
288                     new StringToFind(strToFind));
289             inItems.push_back(currPair);
290         }
291
292         // Run MapReduce
293         OUT_ITEMS_LIST outItems =
294             runMapReduceFramework(mapReduce, inItems, THREAD_LEVEL);
295
296         // Print the [sorted] output
297         for (auto it = outItems.begin(); it != outItems.end(); ++it)
298         {
299             FileNameKey2* filename =
300                 dynamic_cast<FileNameKey2*>(it->first);
301             FileCountValue* count =
302                 dynamic_cast<FileCountValue*>(it->second);
303
304             // For each occurence of the filename, print it.
305             for (int i = 0; i < count->getCount(); ++i)
306             {
307                 std::cout << filename->getFileName() << std::endl;
308             }
309         }
310         cleanup(inItems, outItems);
311     }
312
313     int main(int argc, char* argv[])
314     {
315         if (argc >= MIN_ARGS)
316         {
317             string strToFind = argv[ARG_STR_TO_FIND];
318             std::vector<string> dirNames(argv + ARG_DIR_START, argv + argc);
319             // Minus 2 for this file name (Search.cpp) and the str to find
320             runMapReduce(argc - 2, strToFind, dirNames);
321         }
322         else
323         {
324             std::cerr << USAGE_ERROR_MESSAGE << std::endl;
325             std::exit(EXIT_FAIL);
326         }
327         return EXIT_SUCC;
328     }
```

9 error handle.h

```
1  #ifndef _ERROR_HANDLE_H
2  #define _ERROR_HANDLE_H
3
4  #include <iostream>
5  #include <string>
6
7  #define EXIT_ERROR 1
8
9  using std::string;
10 using std::cerr;
11 using std::endl;
12
13 /**
14  * Handles an error with a function named <funcName> as specified in the
15  * exercise instructions.
16  */
17 void handleError(const string funcName)
18 {
19     cerr << "MapReduceFramework Failure: " << funcName << " failed."
20          << endl;
21     exit(EXIT_ERROR);
22 }
23
24 #endif
```

10 my pthread.h

```c
/**
 * This header is a wrapper for some of the pthread library functions
 * which adds error handling, as specified in the exercise instructions.
 * All functions don't return any value (void functions) and upon errors they
 * print the default error message and exit the process.
 */
#ifndef _MY_PTHREAD_H
#define _MY_PTHREAD_H

#include <pthread.h>
#include "error_handle.h"

#define PTHREAD_SUCCESS 0

void my_pthread_create(pthread_t *thread, const pthread_attr_t *attr,
        void *(*start_routine) (void *), void *arg)
{
    int ret_code = PTHREAD_SUCCESS;
    ret_code = pthread_create(thread, attr, start_routine, arg);
    if (ret_code != PTHREAD_SUCCESS)
    {
        handleError("pthread_create");
    }
}

void my_pthread_join(pthread_t thread, void **retval)
{
    int ret_code = PTHREAD_SUCCESS;
    ret_code = pthread_join(thread, retval);
    if (ret_code != PTHREAD_SUCCESS)
    {
        handleError("pthread_join");
    }
}

void my_pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr)
{
    int ret_code = PTHREAD_SUCCESS;
    ret_code = pthread_mutex_init(mutex, attr);
    if (ret_code != PTHREAD_SUCCESS)
    {
        handleError("pthread_mutex_init");
    }
}

void my_pthread_mutex_destroy(pthread_mutex_t *mutex)
{
    int ret_code = PTHREAD_SUCCESS;
    ret_code = pthread_mutex_destroy(mutex);
    if (ret_code != PTHREAD_SUCCESS)
    {
        handleError("pthread_mutex_destroy");
    }
}

void my_pthread_mutex_lock(pthread_mutex_t *mutex)
{
    int ret_code = PTHREAD_SUCCESS;
```

```
60      ret_code = pthread_mutex_lock(mutex);
61      if (ret_code != PTHREAD_SUCCESS)
62      {
63          handleError("pthread_mutex_lock");
64      }
65  }
66
67  void my_pthread_mutex_unlock(pthread_mutex_t *mutex)
68  {
69      int ret_code = PTHREAD_SUCCESS;
70      ret_code = pthread_mutex_unlock(mutex);
71      if (ret_code != PTHREAD_SUCCESS)
72      {
73          handleError("pthread_mutex_unlock");
74      }
75  }
76
77  void my_pthread_cond_init(pthread_cond_t *cond,
78          const pthread_condattr_t *attr)
79  {
80      int ret_code = PTHREAD_SUCCESS;
81      ret_code = pthread_cond_init(cond, attr);
82      if (ret_code != PTHREAD_SUCCESS)
83      {
84          handleError("pthread_cond_init");
85      }
86  }
87
88  void my_pthread_cond_destroy(pthread_cond_t *cond)
89  {
90      int ret_code = PTHREAD_SUCCESS;
91      ret_code = pthread_cond_destroy(cond);
92      if (ret_code != PTHREAD_SUCCESS)
93      {
94          handleError("pthread_cond_destroy");
95      }
96  }
97
98  void my_pthread_cond_signal(pthread_cond_t *cond)
99  {
100     int ret_code = PTHREAD_SUCCESS;
101     ret_code = pthread_cond_signal(cond);
102     if (ret_code != PTHREAD_SUCCESS)
103     {
104         handleError("pthread_cond_signal");
105     }
106 }
107
108 void my_pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
109 {
110     int ret_code = PTHREAD_SUCCESS;
111     ret_code = pthread_cond_wait(cond, mutex);
112     if (ret_code != PTHREAD_SUCCESS)
113     {
114         handleError("pthread_cond_wait");
115     }
116
117 }
118
119 void my_pthread_cond_timedwait(pthread_cond_t *cond,
120         pthread_mutex_t *mutex, const struct timespec *abstime)
121 {
122     /* int ret_code = PTHREAD_SUCCESS;
123     ret_code = */pthread_cond_timedwait(cond, mutex, abstime);
124     /* if (ret_code != PTHREAD_SUCCESS)
125     {
126         handleError("pthread_cond_timedwait");
127     } */
```

```
128
129 }
130
131 #endif
```