SHARED MEMORY

1. SHARED MEMORY MAIN PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <sys/stat.h>
#include <math.h>

#define FILENAME "input"
#define RECORD_SIZE 100
#define CHUNK_FILE_SIZE 1000000

struct hash {
    char key[10];
    char value[90];
};

long chunks;
long queueCounter;
long op_prefix;
long op_counter;
long fileToMerge;
int noOfThreads;

//Mutex Objects for each method
static pthread_mutex_t mutex_queue=PTHREAD_MUTEX_INITIALIZER;


void divideFileIntoChunks();

void mergeChuncks();
void *mergeFiles();


// Run threads to sort chunk files
void *runThreads() {
    while (1) {
        // Locking thread to increamnet operation conunter
        pthread_mutex_lock(&mutex_queue);
        ++queueCounter;

        // if quecounter is grater than chunk then exit from while loop
        if(queueCounter >= chunks) {
            pthread_mutex_unlock(&mutex_queue);
            break;
        }
        pthread_mutex_unlock(&mutex_queue);

        char program[15];
```

```
        // Calling external compiled program to sort chunk
        sprintf(program, "./sort %ld", queueCounter);
        system(program);
    }
    return NULL;
}



int main(int argc, const char * argv[]) {
    int i;
    noOfThreads = atoi(argv[1]);

    printf("\nDividing data into number of chunks");
    divideFileIntoChunks();


    printf("\nSorting data");
    queueCounter = -1;
    pthread_t *pth= malloc( noOfThreads * sizeof(pthread_t));
    for(i=0; i<noOfThreads; i++)
        pthread_create(&pth[i],NULL, runThreads, "Threads");

    for(i=0; i<noOfThreads; i++)
        pthread_join(pth[i], NULL);

    free(pth);


    printf("\nMerging sorted data");
    mergeChuncks();

    return 0;
}



//-------- Divide File into number of chunks --------//

void divideFileIntoChunks() {
    FILE *fp = fopen(FILENAME, "r");
    char *buffer = malloc(CHUNK_FILE_SIZE * sizeof(char));

    if(!fp)
    return;

    // Calculating number of chunks
    fseek(fp, 0, SEEK_END);
    chunks = ceil(((double) ftell(fp))/CHUNK_FILE_SIZE);
    fseek(fp, 0L, SEEK_SET);

    printf("Chunks: %ld", chunks);

    long i;
    for(i=0; i<chunks; i++) {
        char str[15];
```

```
        sprintf(str, "%ld.txt", i+1);

        FILE *fp_t = fopen(str, "w+");
        if(!fp_t)
            continue;

        // Reading input file with block size 1 MB
        fread(buffer, sizeof(char), CHUNK_FILE_SIZE, fp);
        // Writing chunk file of size 1 MB
        fwrite(buffer, sizeof(char), CHUNK_FILE_SIZE, fp_t);

        fclose(fp_t);
    }
    fclose(fp);
    free(buffer);
}


//-------- Merge File from number of chunks --------//

void mergeChuncks() {
    op_prefix = 1; // Just for namkng merged files
    op_counter = chunks; // Count for total number of files

    while (1) {

        // Renaming final output file to output.txt
        if(op_counter == 1) {
            char oldname[15];
            sprintf(oldname, "%ld.txt", op_prefix);
            rename(oldname, "output.txt");
            break;
        }

        op_prefix = -1 * op_prefix;
        fileToMerge = op_counter;
        op_counter = 0;

        queueCounter = -1;

        // Creating threads to merge sorted files
        pthread_t *pth= malloc( noOfThreads * sizeof(pthread_t));
        int i;
        for(i=0; i<noOfThreads; i++)
            pthread_create(&pth[i],NULL, mergeFiles, "Threads");

        for(i=0; i<noOfThreads; i++)
            pthread_join(pth[i], NULL);

        free(pth);
    }
}


void *mergeFiles() {
    while (1) {
        char *program = malloc(20 * sizeof(char));
```

```
         // Locking thread to increamnet operation conunter
        pthread_mutex_lock(&mutex_queue);
        ++queueCounter;
        if(queueCounter >= fileToMerge) {
            pthread_mutex_unlock(&mutex_queue);
            break;
        }

        // If there is signle file reamining at lat of cycle..
        if(queueCounter+1 == fileToMerge && fileToMerge%2==1) {
            pthread_mutex_unlock(&mutex_queue);
            char oldname[15];
            char newname[15];

            sprintf(oldname, "%ld.txt", (queueCounter+1) * (-
op_prefix));
            sprintf(newname, "%ld.txt", (op_counter+1) * (op_prefix));
            rename(oldname, newname);
            op_counter++;
            continue;

        }
        else {
            ++queueCounter;
            // Execute external merge program
            sprintf(program, "./merge %ld %ld %ld", op_prefix,
op_counter, queueCounter - 1);
            op_counter++;
            pthread_mutex_unlock(&mutex_queue);
        }
        system(program);
    }
    return NULL;
}
```

2. SORTING PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <math.h>


#define RECORD_SIZE 100
#define CHUNK_FILE_SIZE 1000000

struct hash {
    char key[10];
    char value[90];
};

void sortingChunkedFiles(long counter);
void merge(struct hash arr[],long min,long mid,long max, long
totalNumberOfRecords);
void mergeSort(struct hash records[], long l, long r, long
totalNumberOfRecords);


int main(int argc, const char * argv[]) {

    sortingChunkedFiles(atoi(argv[1]));

    return 0;
}


//-------- Merge Sort Algorithm --------//
void sortingChunkedFiles(long counter) {
    char str[15];
    long j;
    sprintf(str, "%ld.txt", counter+1);

    FILE *fp_t = fopen(str, "r+");
    if(!fp_t) {
        printf("Error file reading file %s", str);
        return;
    }

    struct stat stats;
    stat(str, &stats);
    long no_rec = stats.st_size/RECORD_SIZE;
    no_rec = ceil((double)no_rec/RECORD_SIZE)*RECORD_SIZE;

    struct hash *records = malloc(no_rec* sizeof(struct hash));

    char *str1 = malloc(RECORD_SIZE * sizeof(char));
    for (j=0; j<no_rec; j++) {
        // get a record from chuck file
```

```
        fgets(str1, RECORD_SIZE, fp_t);
        if(strcmp(str1, "\n")==0)
            fgets(str1, RECORD_SIZE, fp_t);

        // Making record compatible with gensort
        str1[98] = '\r';
        str1[99] = '\n';

        strncpy(records[j].key, str1, 10); // Extracting key
        strncpy(records[j].value, str1+12, RECORD_SIZE -11);    //
Extracting value
    }

    fclose(fp_t);

    // Call sorting fuction
    mergeSort(records, 0, no_rec - 1, no_rec);

    sprintf(str, "%ld.txt", counter+1);
    fp_t = fopen(str, "w");
    if(!fp_t) {
        printf("Error file reading file %s", str);
        return;
    }

    // Write sorted record to file
    for (j=0; j<no_rec; j++)
        fprintf(fp_t, "%.10s  %s", records[j].key, records[j].value);

    free(str1);
    free(records);

    fclose(fp_t);
}


void merge(struct hash arr[],long min,long mid,long max, long
totalNumberOfRecords) {
    struct hash *tmp = malloc(totalNumberOfRecords * sizeof(struct
hash));
    long i,j,k,m;
    j=min;
    m=mid+1;

    // Comapre two records from each set with other
    for(i=min; j<=mid && m<=max ; i++) {
        if (strcmp(arr[j].key,arr[m].key) <= 0) {
            memcpy(&tmp[i], &arr[j], RECORD_SIZE);
            j++;
        }
        else {
            memcpy(&tmp[i], &arr[m], RECORD_SIZE);
            m++;
        }
    }

    // Append records to sorted array from set1
```

```
    if(j>mid) {
        for(k=m; k<=max; k++) {
            memcpy(&tmp[i], &arr[k], RECORD_SIZE);
            i++;
        }
    }
    // Append records to sorted array from set2
    else {
        for(k=j; k<=mid; k++) {
            memcpy(&tmp[i], &arr[k], RECORD_SIZE);
            i++;
        }
    }

    // Copy again everything to original array of records
    for(k=min; k<=max; k++)
        memcpy(&arr[k], &tmp[k], RECORD_SIZE);
}


/* l is for left index and r is right index of the sub-array of arr to
be sorted */
void mergeSort(struct hash records[], long l, long r, long
totalNumberOfRecords) {
    if (l < r)  {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        long m = (l+r)/2;

        // Sort first and second halves
        mergeSort(records, l, m, totalNumberOfRecords);
        mergeSort(records, m+1, r, totalNumberOfRecords);

        merge(records, l, m, r, totalNumberOfRecords);
    }
}
```

3. MERGING PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <math.h>

#define RECORD_SIZE 100

struct hash {
    char key[10];
    char value[90];
};

void mergeChuncks(long, long, long);
struct hash *getRecord(FILE *fp);
void writeToFile(FILE *fp, struct hash *record);

int main(int argc, const char * argv[]) {


    mergeChuncks(atol(argv[1]), atol(argv[2]), atol(argv[3]));

    return 0;
}


//---------------------------------------- Merge File from number of
chunks ----------------------------------------//

void mergeChuncks(long op_prefix, long op_counter, long fileCounter) {
    char str[15];
    long recCounter1 = 0;
    long recCounter2 = 0;
    struct stat stats1, stats2;

    // Calculating number of records in file1
    sprintf(str, "%ld.txt", (fileCounter+1) * (-op_prefix));
    FILE *fp1 = fopen(str, "r");
    stat(str, &stats1);
    long noRec1 = stats1.st_size/RECORD_SIZE;
    noRec1 = ceil((double)noRec1/RECORD_SIZE)*RECORD_SIZE;

    // Calculating number of records in file2
    sprintf(str, "%ld.txt", (fileCounter+2) * (-op_prefix));
    FILE *fp2 = fopen(str, "r");
    stat(str, &stats2);
    long noRec2 = stats2.st_size/RECORD_SIZE;
    noRec2 = ceil((double)noRec2/RECORD_SIZE)*RECORD_SIZE;

    sprintf(str, "%ld.txt", (op_counter+1) * (op_prefix));
    FILE *fp3 = fopen(str, "w");

    struct hash *record1 = getRecord(fp1);
```

```
    struct hash *record2 = getRecord(fp2);


    // Compare record and append to output file accordingly
    while(1) {
        if (strcmp(record1->key, record2->key) <= 0) {
            writeToFile(fp3, record1);
            record1 = getRecord(fp1);
            recCounter1++;
            if(recCounter1 == noRec1)
                break;
        }
        else
        {
            writeToFile(fp3, record2);
            record2 = getRecord(fp2);
            recCounter2++;
            if(recCounter2 == noRec2)
                break;
        }
    }

    // Append records of file1 to output file1
    while (recCounter1 < noRec1) {
        record1 = getRecord(fp1);
        writeToFile(fp3, record1);
        recCounter1++;
    }
    free(record1);
    fclose(fp1);

    // Append records of file1 to output file2
    while (recCounter2 < noRec2) {
        record2 = getRecord(fp2);
        writeToFile(fp3, record2);
        recCounter2++;
    }
    fflush(fp3);


    free(record2);
    fclose(fp2);

    fclose(fp3);

    sprintf(str, "%ld.txt", (fileCounter+1) * (-op_prefix));
    remove(str);
    sprintf(str, "%ld.txt", (fileCounter+2) * (-op_prefix));
    remove(str);
}

struct hash *getRecord(FILE *fp) {
    // Read record from file
    char *str = malloc(RECORD_SIZE * sizeof(char));
    fgets(str, RECORD_SIZE, fp);
    struct hash *record = malloc(sizeof(struct hash));
    if(strcmp(str, "\n")==0)
```

```
        fgets(str, RECORD_SIZE, fp);

    // Making record compatible with gensort
    str[98] = '\r';
    str[99] = '\n';

    strncpy(record->key, str, 10); // Extract key
    strncpy(record->value, str+12, RECORD_SIZE -11); // Extract value

    free(str);

    return record;
}

void writeToFile(FILE *fp, struct hash *record) {
    // Write record to output file
    if(strlen(record->key) > 9)
        fprintf(fp, "%.10s  %s", record->key, record->value);
}
```

HADOOP

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class HadoopSort {
    public static class TokenizerMapper
                extends Mapper<Object, Text, Text, Text>{

        private Text word1 = new Text();
        private Text word2 = new Text();

        public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
            String str= value.toString();
            word1.set(str.substring(1, 10));
            word2.set(str.substring(12) + "\r\n");
            context.write(word1, word2);
        }
    }

    public static class IntSumReducer
                extends Reducer<Text,Text,Text,Text> {
        public void reduce(Text key, Text value, Context context)
                throws IOException, InterruptedException {
            context.write(key, value);
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        conf.set("mapreduce.output.textoutputformat.separator","  ");

        Job job = Job.getInstance(conf, "Sort");
        job.setJarByClass(HadoopSort.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

SPARK

```
from __future__ import print_function
import sys
from pyspark import SparkContext


def toPlainText(data):
    line = list('  '.join(str(d) for d in data))
    line.extend(['\n'])
    return "".join(line)

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: sort <input file> <output file>",
file=sys.stderr)
        exit(-1)
    sc = SparkContext(appName="Data Sorting..")
    lines = sc.textFile("hdfs://localhost:9000/" + sys.argv[1], 1)

    sortedCount = lines.flatMap(lambda x: x.split('\r\n')) \
        .map(lambda x: (x[:10], x[12:])) \
        .sortByKey() \
        .map(toPlainText)

    sortedCount.saveAsTextFile("hdfs://localhost:9000/" +
sys.argv[2])

    sc.stop()
```