CSE410 Operating Systems Spring 2015
Project 2: Introduction to Threads
Due: 23:59 Thursday, April 2, 2015

1 Overview and Background

In this lab assignment you will gain first hand experience with threads. You will develop a parallel sorting and searching application that will read employee records from a file, create a database, sort the records based on one or more attributes, search for a keyword in the database, and finally write the sorted database into a new file. The tasks of the application will be divided into three main threads. The thread **readerThread** reads the input file and creates the database. The thread **writerThread** writes the sorted database into the output file. The thread **sorterThread** is responsible for sorting the records and searching for a keyword in the database. You have to implement 'parallel merge sort' for sorting the database based on one or more attributes. You also have to find the frequency of a keyword occurring in the database. In order to perform simultaneous searching and sorting, the **sorterThread** may need to create a child thread and assign to its child thread half of the task of sorting a specific part of the database and finding the keyword in that part. That is, each child thread may recursively create more child threads and assign sorting tasks to its children. Each thread also records the cpu time it consumed to perform its task and reports this information back to its parent thread along with the frequency of the keyword. Final output of the application will show the frequency of the keyword in the entire database and total cpu time used by the program.

1.1 Record

A record is a collection of attributes of an employee. Each line of the input file represents a record. There are 8 attributes of an employee - id, gender, first name, middle initial, last name, city name, state name, and social security number. The input file is in csv format; all attributes will be separated by a comma. For example, the following entry represents an employee named Leonardo W DiCaprio from Los Angeles, CA. His employee id is 1 and social security number is 261-68-0688.

```
1, male, Leonardo, W, DiCaprio, Los Angeles, CA, 261-68-0688
```

A data structure *'employeeRecord'* is defined to store an employee record. You will find this code in the skeleton program which will be discussed later.

1.2 Input

The user will provide the following information as input on the command line.

- **inputfilename** - the input file must be in csv format. Each line contains the record of one employee and comprises the following attributes: id, gender, first name, middle name/initial, last name, city name, state name, and social security number.
- **outputfilename** - the output should also be stored in csv format.
- **keyword** - a state name abbreviation (e.g., MI for Michigan, NY for New York, etc.)
- **minsize** - denotes the minimum records needed to create child thread. If the number of records in a sorting task is less than or equal to minsize, no child thread should be created, but rather the parent should handle the task.
- **sortingattributes** - one or more numbers representing the indices of the sorting criteria. Note: 0 stands for id, 1 for gender, 2 for first name, 3 for middle name, 4 for last name, 5 for city name, 6 for state name, and 7 for social security number.

Let us give an example of input to the program. Assuming the executable program name is 'proj2', here is a typical example of starting the program with input.

```
./proj2 input.csv output.csv MI 10 6 5
```

The program must interpret the above input as follows: the program reads employee records from the file 'input.csv', sorts the records of the database by state name and then by city name, records how many employees are from Michigan, and writes the sorted database in the output file 'output.csv'. A child thread with the task of sorting 10 records or less will handle the sorting of those records itself, without creating any additional threads.

1.3 Output

Your program should print the total cpu time consumed in reading, sorting, and writing tasks. Please see Sec. 5 for an example of program output.

1.4 Libraries

You need to use pthread library to complete this sorting application.

1.4.1 Pthreads

You can find information and examples for pthreads at Linux Tutorial as well as many other Internet sites. The following functions of pthread library will be useful for the program.

- *pthread_create*()
- *pthread_join*()
- *pthread_exit*()

The following is a simple example C++ program (thread.cpp) that creates a thread and passes an integer argument.

thread.cpp

```
#include <pthread.h>
#include <iostream>

using namespace std;

// entry function for thread
void* thread_handler(void* arg)
{
    int* argument = (int *) arg;
    cout << "argument value " << *argument << endl;
    // do the task
    pthread_exit(NULL);

}

int main(void)
{
    pthread_t tid;
    int argument = 3;
    int ret = pthread_create(&tid, NULL, thread_handler, (void *)&argument);

    if(ret){
        cout << " unable to create thread " <<__FUNCTION__ << endl;
    }
    pthread_join(tid,NULL);

    pthread_exit(NULL);
```

```
}
```

1.4.2 Thread synchronization

In order to handle synchronization between threads, following functions will be useful

- *pthread_mutex_lock*()
- *pthread_mutex_unlock*()
- *pthread_cond_wait*()
- *pthread_cond_signal*()
- *pthread_cond_broadcast*()

You may find this example page very useful.

2 Implementation Guideline

In order to complete the assignment, you have to create threads, pass arguments (in the form of a structure), and implement parallel merge sort and insertion sort. Functions for reading data from a file and writing data to a file are provided in the skeleton code [link]. The workflow of the program is summarized in Fig. 1. Next, we describe the major tasks of the application.

- **Task 1 (create the threads):** In main(), you need to create three threads for reading, sorting and searching, and writing. During thread creation, you need to initialize thread attributes, pass a pointer of the argList structure. Make sure that the parent thread waits for the child threads to finish the task. Note that in most platforms, threads are joinable by default, i.e., the parent thread can wait on the child thread to finish (Please see the example code).
- **Task 2 (handle synchronization):** The sorterThread must wait for the readerThread to finish reading the file, and the writerThread must wait for the sorterThread to finish sorting task. In order to handle synchronization between threads, you may need to use *pthread_cond* and *pthread_mutex*. The skeleton code for writerThread might be useful in this regard.

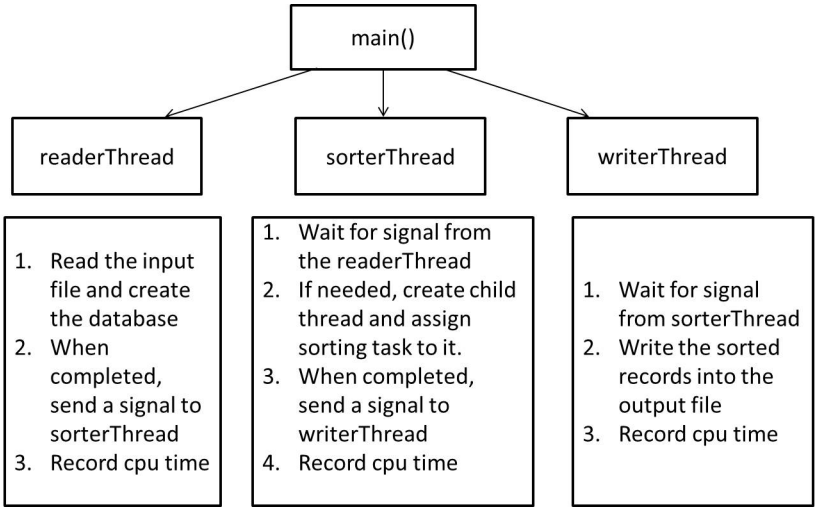

Figure 1: Workflow of the Sorting application

- **Task 3 (comparison):** The user might request the records to be sorted based on more than one attribute. Make sure that your program can handle any number of attributes in any order.
- **Task 4 (insertion sort):** You need to implement a variation of insertion sort that sorts the records, in addition to searching for a keyword in the records.
- **Task 5 (create child threads):** Based on user input and number of records, each thread may need to create child thread and later combine the results from the chid threads.
- **Task 6 (record cpu time):** You must record the cpu time used in each thread. Each thread will add the cpu time consumed by its child thread and return the sum to its parent thread. You should use $gettimeofday()$ function to calculate the cpu time.

IMPORTANT NOTE: You must develop your solution in C++. The skeleton code provided to you is written in C++. You should develop and test your code on machines in 3353 EB, **NOT** on the department servers. Please see Section 3.1 for the complete list of valid machines for testing your program.

3 Requirements, Deliverables and Grading

3.1 Requirements

Working alone or in pairs. You may work on this assignment individually or in pairs (**not** in groups of 3 or more, however). If you prefer to work in a pair, both students must submit a copy of the solution and identify their MSU NetID in a README file. If you prefer to work individually, please clearly state that you are working individually and include your MSU NetID in the README file.

Programming Language. You must implement this project in `C++`. The skeleton code is written in `C++`. Your submission must include a Makefile to compile your code.

Testing your code. Each Linux distribution might be slightly different. It is the students' responsibility to make sure that the lab submissions compile on *at least one* of the following machines in 3353 EB: `carl`, `ned`, `marge` or `skinner`. A statement must be provided in the README file's header. You will **not** be awarded any credit if your lab submission does not compile on any of those machines.

Checking memory leaks You may have to allocate memory. Please use **new** to allocate memory and **delete** to deallocate memory. Make sure that your program is **NOT** leaking memory. You will be penalized if your program causes segmentation fault or leaks any memory.

3.2 Deadline and Deliverables

This lab is due no later than 23:59 (11:59 PM) on Thursday, April 2, 2015. No late submission will be accepted. You must submit your lab using the *handin* utility.
(`https://secure.cse.msu.edu/handin/`) Your submission should include:

- **All source files.** Submit all source files in your project directory. You should also include the skeleton code even if you did not modify it.
- **A makefile.** Include a makefile to compile your code. A makefile is provided in the skeleton code, you may modify it if needed.
- **A README file.** The README file should include a header, sample output and any relevant comments. Please provide the following items in header of the README: the MSU NetIDs of the submitting students, a list of machines on which you have compiled your code, the command used to compile your code. One sample README file header is as follows:

```
Student NetID: alice999, I am working with bob99999.
```

```
Responsibilities: Student 1 works on creating threads and sorting,
Student 2 works on synchronization, merging return values, remaining
tasks are shared equally between them.
Compilation tested on: ned, skinner, marge ...
Command for compile: make
```

Your README file should also include example output from your program, which will aid the TA in debugging if he cannot reproduce your results. You are also encouraged to include any other comments about your program in the README file.

3.3 Grading

This project is worth 75 points. You will not be awarded any points if your submission does not compile. The grading rubric is as follows:

```
General requirements: 5 points
_____  1 pts: coding standard, comments ... etc
_____  2 pts: README file
_____  2 pts: descriptive messages/outputs

Creating main threads: 10 points
_____ 10 pts: creating three threads from main

Managing sorting Threads: 45 points
_____  5 pts: creating child thread
_____  5 pts: passing arguments to child thread
_____  5 pts: combine the results
_____  5 pts: recording cpu time information
_____  7 pts: comparison on multiple attributes
_____  8 pts: handling synchronization
_____ 10 pts: searching and sorting (insertion sort)
Error checking: 15 points
_____  5 pts: handle error in thread creation and other functions
_____ 10 pts: no memory leakage
```

4 Skeleton Code

To assist you in this assignment, skeleton code comprising of three files is provided. The first is a makefile; executing the command `make` will generate the following executables: `proj2`. The remaining two files are described below.

4.1 data structure (record.h)

There are three data structures provided in *record.h* The first structure represents an employee record with the following attributes - id, gender, first, middle, and last name, city name, state name, and social security number. The second structure represents the argument data structure for sorterThread and all of its child threads. The **argList** contains following attributes - thread number, starting index, ending index of the dataset, the keyword, sorting criteria, and a pointer to the **retVal** structure. The return value structure **retVal** contains two attributes - frequency of the keyword and cpu time consumed by the thread.

This file also contains the global vector, dataSet, to hold the database, as well as constants for state information. Please refer to `record.h` in the skeleton code for details.

4.2 Support Functions (pmerge.cpp)

Following functions are fully provided in the skeleton code.

- *void * readData(void* arg)* - entry function for the readerThread
- *void * writeData(void* arg)* - entry function for the writerThread
- *void merge(int s1, int s2, int d1, int d2, vector < int > &v)* - merges two disjoint sorted arrays into one
- *void swap(int index1, int index2)* - swaps two records
- *double getTimeUsed(struct timeval &startTime, struct timeval &endTime)* - calculates the time difference between startTime and endTime
- *int validateParams(. . .)* - validates input parameters

4.3 Skeleton Program (pmerge.cpp)

You need to complete the following functions.

- *void * mergesort(void* arg)* - entry function for sorterThread
- *int insertion_sort(int startIndex, int endIndex, string keyword, vector<int> &v)* - returns the frequency of the keyword and sorts the record between startIndex and endIndex
- *int compareRecords(int index1, index2, vector<int> &v* - compares two records based on attributes in v
- *int main(int argc, char** argv)* - creates threads and prints the output

5 Examples

Following is an example of output from the program. Your output may differ, depending on how the threads are created, and how much cpu time each thread used. The output of the searching and sorting program is shown below.

```
<129 ######:~/cse410/ >./proj2 test.csv output.csv MI 10 4
Input parameters for this run:
    input file name: input.csv
    output file name:output.csv
    keyword: MI
    minsize: 10
    number of sorting attributes: 1
    sorting attributes: lastname
Output:
    Total CPU time for reading : 805 microseconds
    Total CPU time for sorting : 865 microseconds
    Total CPU time for writing :1671 microseconds
    Keyword found: 5 times
```