Operating Systems - Monsoon 2024

November 27, 2024

Assignment 4 (Total points: 100)
Due:

Instructions.
1. Follow all the instructions in the questions closely.
2. Code should be written in C language.
3. Files should be named q[question number].c unless stated otherwise. For example: q1.c
4. Submit a .zip named [RollNo].zip file containing source code files.
5. During the Assignment demos, questions about the code and how it works will be asked. Ensure that you know the code you are submitting.
6. This assignment can be worked on in a group of at most two students.
7. After the deadline, for every 6 hrs, 5% of points will be cut.

Q1. Write a program in C to retrieve the temperature measurement values from a text file and calculate the **min**, **mean** and **max** temperature per weather station. You need to implement two variations of this program (1) using **mmap** and (2) using **fread** (or any file read function). **[20 points]**

Input : Text file containing 50 million rows of weather station data where each line contains the data in the following format:

station_name;temperature

Eg :

Delhi;30.9
Lalmonirhat;23.0
Fernley;39.8
Delhi;29.6
San Antonio de los Baños;24.9
Andalusia;60.5
Shuilin;16.2

Output : The program should output the following for each weather station
Delhi min=<min> mean=<mean> max=<max>
Fernley min=<min> mean=<mean> max=<max>
Andalusia min=<min> mean=<mean> max=<max>

Along with this the program must also print the **total time taken** to process the dataset using **fread** and **mmap.**

Notes:

- You are allowed to use system calls like **madvice** with **MADV_SEQUENTIAL** in order to improve the performance of mmap.

Deliverables:

- Q1.c contains the implementation of this program using **fread** and **mmap.**
- **Makefile** to compile the program.
- A small writeup containing why one approach is better/more efficient than the other approach for this dataset.
    - You can also include the number of system calls invoked using **strace** to indicate the performance bottlenecks.
    - You can also explain how each of these API's work under the hood in order to explain the performance bottlenecks.
- Output containing the **min, mean** and **max** temperature value for each weather station along with total time required for each approach to process the dataset.

Q2. Write a C program to simulate the Shortest Seek Time First (SSTF) disk scheduling algorithm and calculate both seek time and rotational latency for a given set of requests. Assume the disk's rotational speed is 72 RPM, the seek time per track is 2 ms, and the initial position of the head is 100. There are 5 tracks and 100 sectors per track. Tracks are uniformly aligned, i.e., when tracks change, head position changes as 125 -> 225 or 125 -> 25. **[15 points]**

Input:

The disk request queue of length 4, containing the disk sectors requested.

E.g.

```
Disk requests: 78 289 21 495
```

Output:

- Total seek time and rotational latency of each request.

E.g.

```
Seek Time 1: 0 ms
Rotational Latency 1: 0.11 ms
Seek Time 2: 4 ms
Rotational Latency 2: 0.5 ms
...
```

Deliverables:

- Q2.c which contains the implementation of this program.
- **Makefile** should compile this program.

Q3. For this question, you must use `lseek` system call to access file data randomly. You need to index and retrieve specific records from a binary file based on their byte location in the file. **[15 points]**

First, create a **students.rec** binary file using c. This file will contain the records of 5 students. Each record contains the following fields:

```
struct Student {
    int id;            // Incremental student ID (4 bytes)
    char name[20];     // Student name (20 bytes)
};
```

Use `fwrite` function to write the records into the binary file. **Do not submit the program used to create the *students.rec* file.**

Implement the following in Q3.c:

- Ask the user to input a student ID from 1 to 5.
- Open the **students.rec** file, and then use `lseek` to locate the student data in the file.
- Print the name of the found student.

Input:

- ID of the student to be searched.

E.g.

```
Enter student id: 3
```

Output:

- Name of the student corresponding to the input ID.

E.g.

```
Student name: John Doe
```

Deliverables:

- Q3.c, which contains the implementation of the program.
- A **students.rec** file.
- **Makefile** should compile this program.

Q4. Write a program in C to simulate the output redirection using the dup family of system calls(**dup, dup2, dup3).** **[20 points]**

This program consists of 2 parts :

Part 1: Simple Output Redirection

- Implement a function
  - That opens a file for writing (From user)
  - Uses **dup** / **dup2** to redirect **stdout** to file
  - Prints some random text to the redirected **stdout**
- Verify that the data is written in the file.

  ./q4 -p1 output_p1.txt // output.txt must contain some random text

Part 2: Interactive Simulation

- Simulate a simple shell-like command that redirects the **stdout** and **stderr** of any internal command into a file. Eg :

  ./q4 -p2 "ls -l" output_p2.txt // output.txt must contain the output of ls -l

Deliverables :

- Q4.c containing the implementation for both parts (We should be able to test the implementation of both parts)
- Makefile to compile the program
- output_p1.txt and output_p2.txt containing the output from both parts.

Q5. For this question, create a folder **q5** which should contain the following files:

- **list.c** : Contains a simple implementation of the *ls* command. When a directory path is passed to this program as an argument, it will print out all the subdirectories and files inside that directory. If no argument is passed, it lists the current directory. Use the directory related C functions.
- **countw.c** : Contains a simple implementation of the *wc* command. It will print out the word count of the file passed to it as an argument. If no argument is passed, it should throw an error message. Consider words as simply whitespace separated entities.
- **copy.c** : Contains a simple implementation of the *cp* command. We should be able to pass two arguments, the first would be a file path, and the second would be a directory path. The program will copy the given file to the directory passed as the argument. After copying, print out "`[file] copied to [dir]`" where `file` and `dir` are the arguments passed to the program. Print an error message if insufficient arguments are passed.
- **move.c :** Contains a simple implementation of the *mv* command. Should take two arguments representing the source and destination path. Should be able to move the entire content of the source directory into the destination directory.

- **main.c** : This program will run all the executables of the above programs as child processes using the fork-wait-exec sequence. First, create 3 child processes using fork. Then inside the child processes, call one of the exec functions to execute the above programs. One child will run one program. The parent process will wait for the child processes to finish and then return.
    - *The 'list' program should list the files in the parent of the current folder.*
    - *The 'countw' program should count the words in the 'Makefile'.*
    - *The 'copy' program should copy the 'list.c' inside this folder to the parent of this folder.*
    - *The 'move' program should be able to move the entire contents of the q5 folder into a new directory called **q5_files**.*
- **Makefile** : This makefile should build the first 3 programs before building the '*main'* program. Your files should be correctly compiled using this Makefile. The output of compiling a c file should be an executable file of the same name. For example: 'list.c' should be compiled to 'list' in Linux. **[30 points]**