NC STATE UNIVERSITY

CSC 541
Assignment 1

In-Memory vs. Disk-Based Searching

Introduction

The goals of this assignment are two-fold:

1. To introduce you to random-access file I/O in UNIX using C.
2. To investigate time efficiency issues associated with in-memory versus disk-based searching strategies.

This assignment uses two "lists" of integer values: a key list and a seek list. The key list is a collection of integers $K = (k_0, ..., k_{n-1})$ representing n keys for a hypothetical database. The seek list is a separate collection of integers $S = (s_0, ..., s_{m-1})$ representing a sequence of m requests for keys to be retrieved from the database.

You will implement two different search strategies to try to locate each s_i from the seek list:

- **Linear search.** A sequential search of K for a key that matches the current seek value s_i.
- **Binary search.** A binary search through a sorted list of keys K for a key that matches the current seek value s_i. The fact that the keys are sorted allows approximately half the remaining keys to be ignored from consideration during each step of the search.

Each of the two searches (linear and binary) will be performed in two different environments. In the first, the key list K will be held completely in memory. In the second, individual elements $k_i \in K$ will read from disk as they are needed.

Key and Seek Lists

The key and seek lists are provided to you as binary files. Each binary file contains a sequence of integer values stored one after another in order. You can download examples of both files from the Supplemental Material section of this web page.

Be sure to capture these files as binary data. The example file sizes should be 20,000 bytes for the key file, and 40,000 bytes for the seek file. For simplicity, the remainder of the assignment refers only to `key.db` and `seek.db`.

Note. The files we're providing here are meant to serve as examples only. Apart from holding integers, **you cannot make any assumptions** about the size or the content of the key and seek files we will use to test your program.

Program Execution

Your program will be named `assn_1` and it will run from the command line. Three command line arguments will be specified: a search mode, the name of the key file, and the name of the seek file.

```
assn_1 search-mode keyfile-name seekfile-name
```

Your program must support four different search modes.

1. `--mem-lin` Read the key file into memory. Perform a linear search for each seek element s_i in the seek file.
2. `--mem-bin` Read the key file into memory. Perform a binary search for each seek element s_i in the seek file.
3. `--disk-lin` Read each k_i from the key file as it is needed. Perform a linear search for each seek element s_i in the seek file.
4. `--disk-bin` Read each k_i from the key file as it is needed. Perform a binary search for each seek element s_i in the seek file.

For example, executing your program as follows

```
assn_1 --disk-lin key.db seek.db
```

would search for each element in `seek.db` using an on-disk linear search within `key.db`.

In-Memory Sequential Search

If your program sees the search mode `--mem-lin`, it will implement an in-memory sequential search of the key list stored in `key.db`. The program should perform the following steps.

1. Open and read `seek.db` into an appropriately-sized integer array `S`.
2. Open and read `key.db` into an appropriately-sized integer array `K`.
3. Create a third array of integers called `hit` of the same size as `S`. You will use this array to record whether each seek value `S[i]` exists in `K` or not.
4. For each `S[i]`, search `K` sequentially from beginning to end for a matching key value. If `S[i]` is found in `K`, set `hit[i]=1`. If `S[i]` is not found in `K`, set `hit[i]=0`.

You must record how much time it takes to open and load `key.db`, and to then determine the presence or absence of each `S[i]`. This is the total cost of performing the necessary steps in an in-memory sequential search. Be sure to measure only the time needed for these two steps: loading `key.db` and searching `K` for each `S[i]`. Any other processing should not be included.

In-Memory Binary Search

If your program sees the search mode `--mem-bin`, it will implement an in-memory binary search of the key list stored in `key.db`. The keys in `key.db` are stored in sorted order, so they can be read and searched directly. Your program should perform the following steps.

1. Open and read `seek.db` into an appropriately-sized integer array `S`.
2. Open and read `key.db` into an appropriately-sized integer array `K`.
3. Create a third array of integers called `hit` of the same size as `S`. You will use this array to record whether each seek value `S[i]` exists in `K` or not.
4. For each `S[i]`, use a binary search on `K` to find a matching key value. If `S[i]` is found in `K`, set `hit[i]=1`. If `S[i]` is not found, set `hit[i]=0`.

You must record how much time it takes to open and load `key.db`, and to then determine the presence or absence of each `S[i]`. This is the total cost of performing the necessary steps in an in-memory binary search. Be sure to measure only the time needed for these two steps: loading `key.db` and searching `K` for each `S[i]`. Any other processing should not be included.

Recall. To perform a binary search for `S[i]` in an array `K` of size `n`, begin by comparing `S[i]` to `K[n/2]`.

- If `S[i] == K[n/2]`, the search succeeds.
- If `S[i] < K[n/2]`, recursively search the lower subarray `K[0]` ... `K[(n/2)-1]` for `S[i]`.
- Otherwise, recursively search the upper subarray `K[(n/2)+1]` ... `K[n-1]` for `S[i]`.

Continue recursively searching for `S[i]` and dividing the subarray until `S[i]` found, or until the size of the subarray to search is `0`, indicating the search has failed.

On-Disk Sequential Search

For on-disk search modes, you will not load `key.db` into an array in memory. Instead, you will search the file directly on disk.

If your program sees the search mode `--disk-lin`, it will implement an on-disk sequential search of the key list stored in `key.db`. The program should perform the following steps.

1. Open and read `seek.db` into an appropriately-sized integer array `S`.
2. Open `key.db` for reading.
3. Create a second array of integers called `hit` of the same size as `S`. You will use this array to record whether each seek value `S[i]` exists in K or not.
4. For each `S[i]`, search `key.db` sequentially from beginning to end for a matching key value by reading K_0 and comparing it to `S[i]`, reading K_1 and comparing it to `S[i]`, and so on. If `S[i]` is found in `key.db`, set `hit[i]=1`. If `S[i]` is not found in `key.db`, set `hit[i]=0`.

You must record how much time it takes to to determine the presence or absence of each `S[i]` in `key.db`. This is the total cost of performing the necessary steps in an on-disk sequential search. Be sure to measure only the time needed to search `key.db` for each `S[i]`. Any other processing should not be included.

Note. If you read past the end of a file in C, its EOF flag is set. Before you can perform any other operations on the file, you must reset the EOF flag. There are two ways to do this: (1) close and re-open the file; or (2) use the `clearerr()` function to clear the `FILE` stream's EOF and error bits.

On-Disk Binary Search

If your program sees the search mode `--disk-bin`, it will implement an on-disk binary search of the key list stored in `key.db`. The keys in `key.db` are stored in sorted order, so they can be read and searched directly. The program should perform the following steps.

1. Open and read `seek.db` into an appropriately-sized integer array `S`.
2. Open `key.db` for reading.
3. Create a second array of integers called `hit` of the same size as `S`. You will use this array to record whether each seek value `S[i]` exists in K or not.
4. For each `S[i]`, use a binary search on `key.db` to find a matching key value. If `S[i]` is found in `key.db`, set `hit[i]=1`. If `S[i]` is not found in `key.db`, set `hit[i]=0`.

You must record how much time it takes to to determine the presence or absence of each `S[i]` in `key.db`. This is the total cost of performing the necessary steps in an on-disk binary search. Be sure to measure only the time needed to search `key.db` for each `S[i]`. Any other processing should not be included.

Programming Environment

All programs must be written in C, and compiled to run on the `remote.eos.ncsu.edu` Linux server. Any ssh client can be used to access your Unity account and AFS storage space on this machine.

Reading Binary Integers

C's built-in file operations allow you to easily read integer data stored in a binary file. For example, the following code snippet opens a binary integer file for input and reads three integers: the first integer in the file, the third integer from the start of the file, and the second integer from the end of the file.

```
#include <stdio.h>

FILE *inp;          /* Input file stream */
int   k1, k2, k3;   /* Keys to read */

inp = fopen( "key.db", "rb" );

fread( &k1, sizeof( int ), 1, inp )

fseek( inp, 2 * sizeof( int ), SEEK_SET );
fread( &k2, sizeof( int ), 1, inp )

fseek( inp, -2 * sizeof( int ), SEEK_END );
fread( &k3, sizeof( int ), 1, inp )
```

Measuring Time

The simplest way to measure execution time is to use `gettimeofday()` to query the current time at appropriate locations in your program.

```
#include <sys/time.h>

struct timeval tm;

gettimeofday( &tm, NULL );
```

```
printf( "Seconds: %d\n", tm.tv_sec );
printf( "Microseconds: %d\n", tm.tv_usec );
```

Comparing `tv_sec` and `tv_usec` for two `timeval` structs will allow you to measure the amount of time that's passed between two `gettimeofday()` calls.

Writing Results

Results for each key in `S[i]`, and the total time needed to perform all searching, must be written to the console before your program terminates. The format of your output must conform to the following rules.

1. Print one line for each `S[i]` in the order it occurs in `seek.db`. The line must contain the value of `S[i]` padded to be twelve characters wide, a colon and a space, and the text `Yes` if `hit[i]==1` (key found) or the text `No` if `hit[i]=0` (key not found). The simplest way to do this is to use a `printf` statement, for example:

   ```
   printf( "%12d: Yes", S[i] );
   ```

2. Print the total execution time for your program's search operations as a single line with the label `Time:` followed by a space, and the execution time in seconds and microseconds. Assuming the execution time is stored in a `timeval` struct called `exec_tm`, you can use a `printf` statement to do this.

   ```
   printf( "Time: %ld.%06ld", exec_tm.tv_sec, exec_tm.tv_usec );
   ```

You can capture your program's results for further examination or validation by redirecting its output to an external file, for example, to a file called `output.txt`, as follows.

```
assn_1 --mem-lin key.db seek.db > output.txt
```

Your assignment will be run automatically, and the output it produces will be compared to known, correct output using `diff`. Because of this, **your output must conform to the above requirements exactly**. If it doesn't, `diff` will report your output as incorrect, and it will be marked accordingly.

Supplemental Material

In order to help you test your program, we provide an example `key.db` and `seek.db`, as well as the output that your program should generate when it processes these files.

- `key.db`, a binary integer key list file containing 5000 keys,
- `seek.db`, a binary integer seek list file containing 10000 keys, and
- `output.txt`, the output your program should generate when it processes `key.db` and `seek.db`.

You can use `diff` to compare output from your program to our `output.txt` file. Note that the final line containing the execution time most likely won't match, but if your program is running properly and your output is formatted correctly, all of the key searches should produce identical results.

Please remember, as emphasized previously, the files we're providing here are meant to serve as examples only. Apart from holding integers, **you cannot make any assumptions** about the size or the content of the key and seek files we will use to test your program.

Hand-In Requirements

Use Moodle (the online assignment submission software) to submit the following files:

- `assn_1`, a Linux executable of your finished assignment, and
- all associated source code files (these can be called anything you want).

There are four important requirements that your assignment must satisfy.

1. Your executable file must be named exactly as shown above. The program will be run and marked electronically using a script file, so using a different name means the executable will not be found, and subsequently will not be marked.
2. Your program must be compiled to run on `remote.eos.ncsu.edu`. If we cannot run your program, we will not be able to mark it, and we will be forced to assign you a grade of 0.
3. Your program must produce output that exactly matches the format described in the Writing Results section of this assignment. If it doesn't, it will not pass our automatic comparison to known, correct output.

4. You must submit your source code with your executable prior to the submission deadline. If you do not submit your source code, we cannot MOSS it to check for code similarity. Because of this, any assignment that does include source code will be assigned a grade of 0.

Updated 29-Nov-19