CS 105
I/O Lab

1 About Hints

There is a large "Hints" section (Section 7) at the end of this handout. Be sure to read the entire handout and the hints before starting work, and refer back to the hints frequently while you are writing and debugging your program.

2 Introduction

One of the most important approaches to fighting a pandemic is *contact tracing*: locating infected people and tracking down everyone whom they might have passed the disease to. Thus, it is helpful to have a database that keeps a record of everyone who has recently been near an infected person. And since we don't know *a priori* who will become infected, one approach that has been suggested is to use a smartphone application to record those close contacts, and then look them up in a database when an infection is discovered.

In this lab you will build a contact-tracing database and then run simulations to see how well contact tracing works.

2.1 Important Notes: Lab vs. Reality

Before we begin, we want to highlight some important differences between what we are doing in this lab and what goes on in the real world:

Privacy. A database that records interactions between people has enormous negative implications for civil liberties. It can reveal all sorts of private information, including medical issues (i.e., did someone visit a cardiologist?), planned job changes, political associations, marital infidelity, and much more. For that reason, automated contact tracing is controversial, and a good contact-tracing application tries to reduce the risk by anonymizing its database. Nevertheless, a database that allows government authorities to discover who visited whom is inherently problematic. For this lab, we are going to ignore those issues because we are building an example application, but you should think carefully about the implications of creating such a tool in the real world.

Simulation. This lab includes a simulation of the spread of an infection through a large population. We want to emphasize that the epidemiological model used in this lab has **almost zero relationship** to how diseases actually spread. You will find it amusing to play with the various variables and see how they affect the results, but do not be misled into believing that what you learn has any direct applicability to the real world. A realistic model is much more complex and is far beyond the scope of this lab.

2.2 Logistics

As always, you must work with your partner. Handin will be electronic, using `cs105submit`. (You can type "`make submit`" to submit this lab; you will be prompted for the assignment number.)

3 Overview

The purpose of this lab is to allow you to become comfortable with the Unix I/O system, and in particular to help you understand how a complex data structure can built on secondary storage (the disk drive). You will implement a data structure called a B+ tree, which will be stored in a disk file. The tree will serve as a database that holds information about the incidence of a disease in a population; a contact-tracing application (that we provide) will use the database to simulate tracing the spread of the disease.

4 B+ Trees

One of the coolest data structures in computer science is the B-tree and its variants, including the B+ tree, the B* tree, and the B$^\epsilon$ tree. You may have studied B-trees in your data structures course, or you might have previously encountered 2,3,4-trees, which are somewhat related. We will be building a very minor variation on a B+ tree. B+ trees are dictionaries designed to hold more data than can comfortably fit in a typical computer's memory.

In essence, a B+ tree is simply a balanced n-ary tree where n is very large. For example, if n is 256 (2^8), then a three-level tree can access 2^{24} = 16M items (compare a balanced binary tree, where 16M items would require 24 levels). The large fanout and shallow depth are critically important because, unlike most other data structures, a B+ tree is stored on disk rather than in memory. Descending one level requires a disk I/O, which typically takes 5–10 ms. Thus, looking a single item up in a 3-level B+ tree would take 15–30 ms,[1] whereas a corresponding 24-level binary tree would take 120–240 ms, or nearly a quarter of a second. That 8-to-1 advantage becomes critically important in the face of multiple lookups.[2]

To ensure that a B+ tree is balanced, insertions and deletions happen only at the leaves of the tree. If a leaf becomes full, it is *split* into two half-full leaves, and a pointer to the new node is

[1] In fact, the number is usually far less because the internal nodes can often be found in the operating system's buffer cache, so that no I/O is necessary to fetch them. Only the "leaf" node must be accessed, reducing the cost to just 5–10 ms.

[2] Real B+ trees usually have even greater fanouts: 1K, 16K, or larger.

inserted into the node immediately above. If that node becomes full, it too is split, and so on up to the root. Finally, if the root is filled, it is split in half and a new root is created just above it. It is only in that case that the tree depth increases.

4.1 Our B+ Trees vs. Real Ones

Note: You can safely skip this section when you are getting started on the lab and read it later instead.

There are a few minor differences between our B+ Trees and a robust production implementation. All of the differences are designed to simplify the code:

- We don't support deletion; deletion is a must in the real world.
- Our aggressive approach to splitting can be a bit wasteful of both space and time.
- Since the keys in an internal node represent the "boundaries" between the block pointers, the required number of keys is one less than the number of pointers. Dropping one key would allow one more pointer, giving a slightly larger fanout and thus a (possibly) shallower tree.
- "Range" queries that ask for a range of keys are a common operation. To support range queries, the leaf nodes should be managed as a singly linked list with pointers to siblings.
- It's unrealistic to require fixed-sized keys and values. A flexible B+ tree will allow completely variable sizes for both (at the cost of some extra overhead and lower fanout, since some space then needs to be used to keep track of the sizes of things).

4.2 Handout

The handout is distributed in a `tar` file named `iolab-handout.tar`, which you will find linked from the lab Web page. Start by copying `iolab-handout.tar` to a (protected) directory in which you plan to do your work. Then give the following command:

```
tar xvf iolab-handout.tar
```

This will cause a number of files to be unpacked in the directory:

`Makefile` A Makefile that will build the contact-tracing application, including your B+ Tree implementation. You should always compile using `make` so that you compile with the correct options.

`contact_trace.c` The contact-tracing application itself. You should not modify this file.

`btree.h` A header file that contains the API declarations for the B+ tree implementation. You should not modify this file.

`btree.c` A skeleton for your B+ tree implementation. This is the file you will modify, and is the only file you will hand in.

`sample_solution` A runnable sample solution to the lab.

`iolab.pdf` A copy of this writeup.

5 The Contact-Tracing Application

The B+ tree will serve as the database for a simple epidemiological contact-tracing program. The program operates in 5 phases:

1. Create N people (default 4000), each with a unique (random) 32-bit identifier, and insert them into the database.
2. Mark a single person as known to be infected with a disease.
3. Simulate C contacts (default 40000) among people as follows:
   (a) Select two people at random. (See below for a discussion of semi-isolation groups.)
   (b) Update each person's "who has been near me" table with the identity of the other person. If the table is full, discard the oldest entry in the table.
   (c) If one person is healthy and the other is infected (whether known or unknown to the epidemiologists) possibly mark the healthy person as infected but unknown.
4. Repeatedly trace contacts until the table stops changing:
   (a) For each person in the table who is known to be infected, look up their contacts in the database.
   (b) For each of those contacts "test" them for the disease: if they are infected but unknown, mark them as known.
5. Finally, print a single character for each person: "." if they are healthy, "?" if they are infected but not known, and "X" if they are infected and known. Doing so will produce a nice graphic in your terminal.

The results of the program are left in a file named `people.btree`. If you wish, you can experiment further by specifying the "-r" switch (see below).

5.1 Options

The contact-tracing program accepts the following switches (plus a few others):

-c n Keep track of only the n most recent contacts (default 5, maximum 13).

-g n Divide the population into groups of size n; all group members can interact with each other but only one group member (the "leader") is allowed to interact with anyone outside the group, and then only with other "leaders."

-N n Simulate n interactions between people. Note that this parameter is population-wide; if you specify `-N 10` in a population of 4000, ten randomly chosen people will interact once each with a single other person—which probably means the disease won't spread at all. The default is 10 interactions per person, i.e., 40,000 interactions for a population of 4,000.

-n n Simulate a population of n people. Obviously, larger values of n will take longer and will fill up your screen with more data. The program will start to misbehave if n is more than about 100,000,000 (but it will also become excruciatingly slow—and will fill up your disk quota—before then). The default is 4,000, which will look nice on your screen.

-p After the population is created, print a representation of the tree to standard output. This output shows the nesting of the tree and the keys in each entry, but not the values. You may find it useful for debugging your code.

-r Restart the simulation from the state saved in the B+ tree. In this case the number of people is taken from the database, but new interactions are generated and you can change the number of tracked contacts. You will normally want to specify a different value for the "-s" switch if you use "-r".

-s n Set the seed of the random-number generator to n (default 0). We use a constant seed so that the results of the simulation will be reproducible, but if you try different seeds you watch the results vary from run to run.

-t x Set the probability of disease transmission upon each contact (0-1, default 0.15).

5.2 Running the Application

We have provided a sample solution to the problem (cleverly named "`sample_solution`"). At any point, you can run it to see what the output should look like. Furthermore, the Python script named "`test_contact`" will compare the output of your program with the sample solution (see Section 5.2.2).

5.2.1 Trial Runs

We suggest that you begin by running the sample solution in several ways:

`./sample_solution` The default parameters infect a quite a few people—but they don't identify very many of the infections.

`./sample_solution -c 13` Tracking more contacts helps a lot—the question marks have almost all turned into X's.

`./sample_solution -t 0.45` Unsurprisingly, increasing the transmission probability increases the number of infections. Less obviously, it also makes tracing work better (why?).

`./sample_solution -N 800000` Increasing the number of interpersonal interactions also infects more people—in fact, doubling the interactions gives more infections (3619) than tripling the transmission probability does (3449).

`./sample_solution -t 0.45 -g 5` Limiting most interactions to small groups reduces the spread!

`./sample_solution -n 30 -p` Will show you a very small B+ tree.

Feel free to play around more—but always remember that this is an inaccurate simulation.

5.2.2 Testing Your Version

At first, you should run the application with a VERY small number of people. We recommend that you start with three, because three people will fit into a single leaf:

```
./contact_trace -n 3
```

After you get that working, try a few more people (still in the single digits) to test your leaf-splitting. When you get over 29 people your code will have to split the root node, and at 301 it'll split internal nodes (after which point things should start to become pretty robust).

When you are pretty sure you have a correct answer, you can use the "`test_contact`" Python script to compare your output with the output of the sample solution. If there are any differences between the two, they will be highlighted in blue. You can choose different highlight colors with the `-h` switch, and you can give other contact-tracing switches as long as `-h` is given first. For example:

```
./test_contact -h white -n 3000
```

will run with 3000 people and highlight differences in white. Try different highlight colors—including the color of your background—to see which is best. (You can look at the Python code to see the available choices.)

6 Detailed Instructions

Your job is to implement a B+ tree. We have provided a handout that contains several files; you will modify only one file, `btree.c`.

6.1 Implementing the B+ Tree

At the top of `btree.c`, just after the `#include` statements, is a data structure where you must put your names and CS logins. Do that now, before you forget! Be sure to maintain the format of information in the data structure (including the NULL at the end); our grading software expects that format.

6.2 B+ Tree Structure

Because disk I/O is so expensive, a B+ tree is carefully laid out to play well with the disk. In particular, each node of the tree is the same size, and that size is deliberately chosen to be the same size as a disk block so that it can be efficiently read from or written to the disk drive.[3]

Our B+ tree is a dictionary, also known as a key-value store. The key is an 8-byte integer (a C `unsigned long`), defined as type `b_key` in the provided `btree.h` header file. The value is an arbitrary byte array of up to 56 bytes, defined as type `b_value` in the header. (The provided contact-tracing application uses that space to store a `struct human`.)

[3]For this lab, that's a slight lie. The lab can be built with different node sizes, and we will start with a size smaller than a block because doing so will help you debug your code.

The B+ tree makes a distinction between *internal* and *leaf* nodes. An internal node does not contain data; it only has keys (type `b_key`) and the offsets of other nodes (type `b_offset`). A leaf node contains pairs of keys and values. In the header file, both types of nodes are encompassed in a `b_block`, which is the size of a (simulated) disk block. You can choose that size when you compile your program.

6.2.1 Internal Nodes

The exact layout of an internal node is as follows:

```
typedef struct {
    b_key       key;    // First key in block
    b_offset    offset; // Offset of block in b_tree file
}
    b_kp_pair;

typedef struct {
    b_type      type;   // Must be B_INTERNAL
    b_count     count;  // Number of key/offset pairs in this node
    b_kp_pair   pairs[B_KP_PAIRS_PER_BLOCK];
}
    b_internal;
```

where `B_KP_PAIRS_PER_BLOCK` is calculated so that a `b_node` will just fit inside a disk block. The `type` field identifies the block as being an internal node (compare leaf nodes, below).

Each entry in `pairs`, say `pairs[i]`, describes a subtree rooted at `pairs[i].offset`. That subtree contains all keys that are greater than or equal to `pairs[i].key` and less than `pairs[i+1].key`. (The last entry in `pairs` describes all keys greater than its key.)

For example, consider a node that covers keys from 1000–1999 and has three leaf nodes below it. The first leaf (*A*) contains keys in the range 1000–1199, the second (*B*) covers 1200–1699, and the third (*C*) is for 1700–1999. (For simplicity, we refer to the leaves by labels rather than their true disk addresses.)

Field	Value
`type`	`B_INTERNAL`
`count`	3
`pairs[0]`	1000 / *A*
`pairs[1]`	1200 / *B*
`pairs[2]`	1700 / *C*

When we are looking up a key k in a node, we want to find the largest value of i such that `pairs[`i`].key` is less than or equal to k.[4] So if $k = 1234$, we will locate `pairs[1]` and then follow its block pointer to search block B for the key.

A minor note: you may be wondering what we should do if k is less than `pairs[0].key`. The answer is that it's an error: the root node starts with a key of 0, and any internal node should only be reached if k is in its range (e.g., we should only look inside node B if $1200 \leq k < 1700$).

[4] A binary search is easy to write and is the most efficient approach.

Although in this example we only have two levels, in general we can have many. An internal node might cover a range of a million keys; each entry in the `pairs` array might then lead to a child node that covers about 100,000 keys.

6.2.2 Leaf Nodes

A leaf node is different from an internal node, because there is no need for block pointers but key/value pairs must be recorded:

```
typedef struct {
    b_key       key;    // Key for looking up this value
    b_value     value;  // Value associated with key
}
    b_kv_pair;

typedef struct {
    b_type      type;   // Must be B_LEAF
    b_count     count;  // Number of key/value pairs in this node
    b_kv_pair   kv[B_KV_PAIRS_PER_BLOCK];
}
    b_leaf;
```

where `B_KV_PAIRS_PER_BLOCK` is calculated so that a `b_leaf` will just fit inside a disk block. The `type` field identifies the block as being a leaf node (compare internal nodes, above).

Leaf nodes are conceptually simpler than internal nodes; a key is either present or not, and can quickly be found by a binary search.

6.2.3 Generic Nodes

When we read a block from the disk, we don't know *a priori* whether it is an internal node or a leaf node. Similarly, the code that writes data back should work regardless of which type of node it is. Finally, it's also critically important that the amount of data written is exactly the size of a disk block; in the structures above that's hard to guarantee.

For that reason, both kinds of nodes are encapsulated into a union:

```
typedef union {
    b_internal  internal; // Internal node with pointers
    b_leaf      leaf;     // Leaf with data
    char        pad[B_BLOCK_SIZE];
}
    b_block;
```

where `B_BLOCK_SIZE` is defined as the size of a disk block (256 by default; see Section 7 for a bit more about that). The `pad` field ensures that the union is at least as big as the desired size; the declarations of `b_node` and `b_leaf` are no larger than that size, so in the end a `b_block` is precisely correct. Note that since there is a `type` field, we can check the type of any block without knowing it beforehand. For example, if we have declared `data` as a `b_block`, we can write:

```
if (data.internal.type == B_INTERNAL)
    // Handle internal node
```

Accessing fields inside a block is a bit painful, since you would have to write things like `data.internal.count` to learn the number of key/pointer pairs in an internal node. So there are some `#define` declarations to simplify notation:

```
#define i_type  internal.type
#define i_count internal.count
#define i_pairs internal.pairs
#define l_type  leaf.type
#define l_count leaf.count
#define l_kv    leaf.kv
```

That way, you can write just `data.i_type` to get an internal node's type, `data.i_count` to get its count field, and so forth.

6.2.4 Insertions

As mentioned above, if a node is full when inserting a key, we must split that node into two half-full ones, perform the insertion, and then insert a pointer to the new node into the node above. It turns out that it's a bit simpler to be aggressive about node splitting: on the way down the tree, **if we might do an insertion** we check the size of each node we encounter and, if it is full, proactively split it. This approach sometimes splits nodes sooner than necessary (you should think about why) but is easier to write.

6.3 Functions You Need to Write

The skeleton file provided in `btree.c` contains "NEEDSWORK" and "END NEEDSWORK" comments to highlight things you need to look at. (A few of the NEEDSWORK comments are just things you need to read, but most mark places where you will need to write new code.)

An important detail: the code we have provided won't run as-is. In particular, it will crash if you implement `write_block` but not `search_internal`.

Here are the specific things you need to do; we have listed them in the order you should attack them rather than in the order they appear in the file:

`write_block` Doing I/O to the B+ tree requires functions that will read and write blocks at a given offset in the file. Your code won't work until `write_block` works. We have provided a full implementation of `read_block` for you, and you can study and use it as a guide for writing `write_block`—but notice that latter is simpler than `read_block`.

`find_end` Although you won't need this function right away, it is also very simple to implement. Read the manual page for `lseek` ("man 2 lseek") and then write the code.

`search_internal` This function is responsible for finding the proper child in an internal node; it also calls `split_internal` and `split_leaf` when necessary. Write the search code now, and note that you may have to make a small change later (see the comments).

`search_leaf` This function is responsible both for finding information in a leaf node, and for inserting data into a leaf. You will need to add code to do searches and inserts. Once you've done that, the test program should be able to run with up to 2 people (and you should test it at this point to be sure it works). In addition, if you compile with the command "`make B_BLOCK_SIZE=4096 clean all`", the test program should work with up to 62 people—and you should test that as well.

`split_root` We have provided a complete implementation of this function. However, you should study it carefully before you attack the next two functions, because it will serve as a useful guide.

`split_leaf` This function is called when a leaf node (`child_to_split`) becomes full. It must split the leaf into two nodes, and then update its parent by inserting a pointer to the new "sibling" node. (The parent is guaranteed to have room for the new pointer—why?) You can use `split_root` as a guide for writing your own code—note that `split_leaf`'s requirements are a bit simpler than `split_root`'s. After you write `split_leaf`, recompile without changing `B_BLOCK_SIZE` and test with increasing numbers of people—5 at first, then more until your code works with 29 people.

`split_internal` This is the last function you should write. Similar to `split_leaf`, this function is called when an internal node (`child_to_split`) becomes full. It must split the node into two nodes, and then update its parent by inserting a pointer to the new "sibling" node. Start your testing by tracing contacts for 30 people and then try larger numbers. You'll probably find that once it works for around 100 people, it will work equally well for 4,000.

7 Hints

- Don't try to write everything at once! Do things in the order given in Section 6.3 above, and test with just two people as soon as you have written the search code in `search_leaf`. Test again, with three people, after you've written the leaf insertion code. Continue to follow the testing advice above as you write more functions.
- Use gdb! Debuggers are amazingly useful for dealing with complicated data structures. A useful hint for this assignment is to print things in hexadecimal. For example, if "`node`" points to a B+ tree node, try "`p/x *node`". Even better, "`p/x node.leaf`" will show just the leaf fields of a leaf node.
- In gdb, use `step` and `next`, not `stepi` and `nexti`. You *really* don't want to debug at the instruction level for this lab.
- You may find it helpful to display the encoded tree on your screen. But since it's a binary file, you can't just bring it up in an editor or use "cat". Instead, try this command:

```
od -t x4 -A x people.btree | less
```

That will print the file in hexadecimal, which is useful when dealing with things like the people IDs used by the contact-tracing application. If you're not familiar with "less", it's a program that presents things to you one screen at a time. Hit the space bar to see more, or use the page up/down keys. Hit "q" when you're done looking, or "?" for more help (for example, you can do string searches, which can be very helpful in debugging). You should spend a bit of time learning how to interpret the hex dump as B+ tree nodes.

- As mentioned above, you should start by testing with a VERY small tree ("-n 3"). That will keep things from getting horribly complicated while you're still figuring out your code.
- As distributed, the Makefile sets up your B+ tree code to use a very small block size of 256 bytes. This size was chosen so that you would only need to insert a few people before your node-splitting code was invoked. Once you have your code working, you should test it with a more reasonable block size. You can do that on the `make` line:

  ```
  make B_BLOCK_SIZE=4096 clean all
  ```

  Be sure to test with a large block size! During grading, we will test with larger blocks as well with 256-byte ones. (Note that the larger the block size, the more people you'll need to ensure that the splitting code gets exercised.)

8 Output Format

All of the output for this lab is generated by the `contact_trace` program. There are two types of output: the result of contact tracing, and the debugging output from the "-p" switch.

8.1 Contact Tracing Output

The application generates one character for each individual, in a single long line that should automatically wrap on your terminal to form a 2-dimensional grid. There is a "." for a healthy person, an "X" for a person known to be infected, and a "?" for a person who is infected but has not been identified by contact tracing. For example, if your screen is 10 characters wide it might look like this:

```
X.......XX
..X..X.X.X
X.X.X...X.
XX..X....X
.X.XXXX.X.
.X...X..XX
XX..X....X
.X..X..X.X
..X...X..?
.X....XX.X
```

Note that the layout of the characters does NOT imply that the simulated people are physically next to each other; it's just to make it easy for humans to look at.

8.2 Debugging Output

Here is a sample of the output from the "-p" switch for a small tree:

```
ROOT @0x0
+-LEAF 0x00000000 - 0x66334873 @0x100
|   0x327b23c6
|   0x643c9869
+-LEAF 0x66334873 - 0x6b8b4567 @0x300
|   0x66334873
+-LEAF 0x6b8b4567 - 0xffffffffffffffff @0x200
|   0x6b8b4567
```

The plusses, dashes, and vertical bars show the physical structure of the tree. For each node in the tree, there is a line like this one:

```
+-LEAF 0x00000000 - 0x66334873 @0x100
```

The first word in the line is either LEAF or INTERNAL. Next are two hexadecimal numbers that give the range of keys covered by the node, in this case 0x0 through 0x66334873. Finally, the "@0x100" indicates that the node is at offset 0x100 (decimal 256) in the file.

Below an internal node, all the nodes below it are printed. Below a leaf, the output gives the keys in that leaf. An important detail is that the keys are printed in order, so if you're using binary search they should be sorted. Similarly, the keys in any given node (internal or leaf) should fall into a subrange of the keys in its parent.

9 Finishing the Lab

When you are done with the lab, use `cs105submit` to submit ONLY btree.c. Remember that you can submit more than once; we will grade only the last submission. So submit early and often!