This homework covers OSTEP chapters 40–41. It is due in class Monday, May 4.

While you may discuss ideas and strategies for problems with other students, you should always make the first attempt on a problem yourself and **you must write up your own solutions in your own words**. *You may not collaboratively write solutions or copy a solution that one person in the group writes up. You also may not look for, copy, or use solutions from other sources, including from generative AI like ChatGPT, even if you make changes. There's no such thing as using someone else's solution for a problem "as an example" for writing your own.*

Preliminaries

- Copy the directories `file-implementation` and `file-ffs` (and their contents) from `/classes/cs331` to your `~/cs331` directory. (You won't be writing any code so these don't need to go into your `workspace` directory.)

These directories contain the programs `vsfs.py` and `ffs.py`, respectively, along with `README`s with instructions and (in the latter case) supporting files. You can also find the text of the `README`s in the appendix sections A and B at the end of this document.

Exercises

Chapter 40 homework problems (at the very end of chapter 40) —

1. Do #1–2 for practice — make sure you understand what happens in the file system for each of the operations and think about the question posed regarding allocation algorithms. Look at the `README` to see what operations are possible and to understand the simulator's output. Check your work with the `-c` flag. There is nothing to hand in for these exercises.

2. For #3–4, running the simulator with the extremely low values suggested is probably not that helpful. Instead just think about the filesystem operations the simulator includes — which require inodes but few or no new data blocks and which require data blocks but few or no new inodes? Address the questions posed based on that.

Chapter 41 homework problems (at the very end of chapter 41) —

3. Do #1–2 for practice — make sure you understand both the input (what the contents of `in.largefile` means) and the output from the simulator, and think about what you expect to see before running with the `-c` flag. There is nothing to hand in for these exercises.

4. Do #3. Run with `-S` to see the specific blocks allocated and make sure you understand the simulator's output — the block addresses listed are numbered consecutively within the data region, but on the disk each group contains both inodes and data blocks. The computed filespan includes the inode blocks. Address the question posted (about the effect of the `-L` threshold on the filespan) and explain why that occurs.

5. Do #4 for practice — make sure you understand both the input (what the contents of `in.manyfiles` means) and the output from the simulator, and think about what you expect to see before running with the `-c` flag. There is nothing to hand in for this exercise.

6. Do #5–6. Again, make sure you understand the output from the simulator and think about what you expect to see before running with the `-c` flag. Address the questions posed (about how well FFS minimizes dirspan and the effect of the inode table size). Also address filespan — what impact does changing the inode table size have on filespan? In all cases, explain your answer — why?

7. Do #7. Running with `-A 3` instead of `-A 2` may make it easier to see and understand what is going on. Address the questions posed (about the effect on dirspan and why the policy might be good) and explain why the policy has the effect it does.

8. Do #8–9. For #9, use `in.fragmented2` instead and try each value of `-C` up to 20 or 25 to see what is going on. Address the questions posed for both problems (about the layout of file `/i` in both files and the effects of `-C` on the layout, filespan, and dirspan) and explain why the policy has the effect it does.

A file-implementation

Use this tool, `vsfs.py`, to study how file system state changes as various operations take place. The file system begins in an empty state, with just a root directory. As the simulation takes place, various operations are performed, thus slowly changing the on-disk state of the file system.

The possible operations are:

- mkdir() - creates a new directory
- creat() - creates a new (empty) file
- open(), write(), close() - appends a block to a file
- link() - creates a hard link to a file
- unlink() - unlinks a file (removing it if linkcnt==0)

To understand how this homework functions, you must first understand how the on-disk state of this file system is represented. The state of the file system is shown by printing the contents of four different data structures:

- inode bitmap: indicates which inodes are allocated
- inodes: table of inodes and their contents
- data bitmap: indicates which data blocks are allocated
- data: indicates contents of data blocks

The bitmaps should be fairly straightforward to understand, with a 1 indicating that the corresponding inode or data block is allocated, and a 0 indicating said inode or data block is free.

The inodes each have three fields: the first field indicates the type of file (e.g., f for a regular file, d for a directory); the second indicates which data block belongs to a file (here, files can only be empty, which would have the address of the data block set to -1, or one block in size, which would have a non-negative address); the third shows the reference count for the file or directory. For example, the following inode is a regular file, which is empty (address field set to -1), and has just one link in the file system:

```
[f a:-1 r:1]
```

If the same file had a block allocated to it (say block 10), it would be shown as follows:

```
[f a:10 r:1]
```

If someone then created a hard link to this inode, it would then become:

```
[f a:10 r:2]
```

Finally, data blocks can either retain user data or directory data. If filled with directory data, each entry within the block is of the form (name, inumber), where "name" is the name of the file or directory, and "inumber" is the inode number of the file. Thus, an empty root directory looks like this, assuming the root inode is 0:

```
[(.,0) (..,0)]
```

If we add a single file "f" to the root directory, which has been allocated inode number 1, the root directory contents would then become:

```
[(.,0) (..,0) (f,1)]
```

If a data block contains user data, it is shown as just a single character within the block, e.g., "h". If it is empty and unallocated, just a pair of empty brackets ([]) are shown.

An entire file system is thus depicted as follows:

```
inode bitmap 11110000
inodes       [d a:0 r:3] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] ...
data bitmap  11100000
data         [(.,0) (..,0) (y,1) (z,2) (f,3)] [u] [(.,3) (..,0)] [] ...
```

This file system has eight inodes and eight data blocks. The root directory contains three entries (other than "." and ".."), to "y", "z", and "f". By looking up inode 1, we can see that "y" is a regular file (type f), with a single data block allocated to it (address 1). In that data block 1 are the contents of the file "y": namely, "u". We can also see that "z" is an empty regular file (address field set to -1), and that "f" (inode number 3) is a directory, also empty. You can also see from the bitmaps that the first four inode bitmap entries are marked as allocated, as well as the first three data bitmap entries.

The simulator can be run with the following flags:

```
prompt> vsfs.py -h
Usage: vsfs.py [options]

Options:
  -h, --help            show this help message and exit
  -s SEED, --seed=SEED  the random seed
  -i NUMINODES, --numInodes=NUMINODES
                        number of inodes in file system
  -d NUMDATA, --numData=NUMDATA
                        number of data blocks in file system
  -n NUMREQUESTS, --numRequests=NUMREQUESTS
                        number of requests to simulate
  -r, --reverse         instead of printing state, print ops
  -p, --printFinal      print the final set of files/dirs
  -c, --compute         compute answers for me
```

A typical usage would simply specify a random seed (to generate a different problem), and the number of requests to simulate. In this default mode, the simulator prints out the state of the file system at each step, and asks you which operation must have taken place to take the file system from one state to another. For example:

```
prompt> ./vsfs.py -n 6 -s 16
...
Initial state

inode bitmap  10000000
inodes        [d a:0 r:2] [] [] [] [] [] [] []
data bitmap   10000000
data          [(.,0) (..,0)] [] [] [] [] [] [] []

Which operation took place?

inode bitmap  11000000
inodes        [d a:0 r:2] [f a:-1 r:1] [] [] [] [] [] []
data bitmap   10000000
data          [(.,0) (..,0) (y,1)] [] [] [] [] [] [] []

Which operation took place?

inode bitmap  11000000
inodes        [d a:0 r:2] [f a:1 r:1] [] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1)] [u] [] [] [] [] [] []

Which operation took place?

inode bitmap  11000000
inodes        [d a:0 r:2] [f a:1 r:2] [] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1) (m,1)] [u] [] [] [] [] [] []

Which operation took place?

inode bitmap  11000000
inodes        [d a:0 r:2] [f a:1 r:1] [] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1)] [u] [] [] [] [] [] []

Which operation took place?
```

```
inode bitmap  11100000
inodes        [d a:0 r:2] [f a:1 r:1] [f a:-1 r:1] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1) (z,2)] [u] [] [] [] [] []

Which operation took place?

inode bitmap  11110000
inodes        [d a:0 r:3] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] [] [] []
data bitmap   11100000
data          [(.,0) (..,0) (y,1) (z,2) (f,3)] [u] [(.,3) (..,0)] [] [] [] [] []
```

When run in this mode, the simulator just shows a series of states, and asks what operations caused these transitions to occur. Running with the "-c" flag shows us the answers. Specifically, file "/y" was created, a single block appended to it, a hard link from "/m" to "/y" created, "/m" removed via a call to unlink, the file "/z" created, and the directory "/f" created:

```
prompt> vsfs.py -n 6 -s 16 -c
...
Initial state

inode bitmap  10000000
inodes        [d a:0 r:2] [] [] [] [] [] [] []
data bitmap   10000000
data          [(.,0) (..,0)] [] [] [] [] [] [] []

creat("/y");

inode bitmap  11000000
inodes        [d a:0 r:2] [f a:-1 r:1] [] [] [] [] [] []
data bitmap   10000000
data          [(.,0) (..,0) (y,1)] [] [] [] [] [] [] []

fd=open("/y", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);

inode bitmap  11000000
inodes        [d a:0 r:2] [f a:1 r:1] [] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1)] [u] [] [] [] [] [] []

link("/y", "/m");
```

```
inode bitmap  11000000
inodes        [d a:0 r:2] [f a:1 r:2] [] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1) (m,1)] [u] [] [] [] [] [] []

unlink("/m");

inode bitmap  11000000
inodes        [d a:0 r:2] [f a:1 r:1] [] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1)] [u] [] [] [] [] [] []

creat("/z");

inode bitmap  11100000
inodes        [d a:0 r:2] [f a:1 r:1] [f a:-1 r:1] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1) (z,2)] [u] [] [] [] [] [] []

mkdir("/f");

inode bitmap  11110000
inodes        [d a:0 r:3] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] [] [] []
data bitmap   11100000
data          [(.,0) (..,0) (y,1) (z,2) (f,3)] [u] [(.,3) (..,0)] [] [] [] [] []
```

You can also run the simulator in "reverse" mode (with the "-r" flag), printing the operations instead of the states to see if you can predict the state changes from the given operations:

```
prompt> ./vsfs.py -n 6 -s 16 -r
Initial state

inode bitmap  10000000
inodes        [d a:0 r:2] [] [] [] [] [] [] []
data bitmap   10000000
data          [(.,0) (..,0)] [] [] [] [] [] [] []

creat("/y");

  State of file system (inode bitmap, inodes, data bitmap, data)?

fd=open("/y", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);
```

```
  State of file system (inode bitmap, inodes, data bitmap, data)?

link("/y", "/m");

  State of file system (inode bitmap, inodes, data bitmap, data)?

unlink("/m")

  State of file system (inode bitmap, inodes, data bitmap, data)?

creat("/z");

  State of file system (inode bitmap, inodes, data bitmap, data)?

mkdir("/f");

  State of file system (inode bitmap, inodes, data bitmap, data)?
```

A few other flags control various aspects of the simulation, including the number of inodes ("-i"), the number of data blocks ("-d"), and whether to print the final list of all directories and files in the file system ("-p").

B file-ffs

This is the README for `ffs.py`, a simulator of FFS allocation policies. Use it to study FFS behavior under different file and directory creation scenarios.

The tool is invoked by specifying a command file with the -f flag, which consists of a series of file create, file delete, and directory create operations.

For example, run:

```
prompt> ./ffs.py -f in.example1 -c
```

to see the output from the first example in the chapter on how FFS based allocation works.

The file `in.example1` consists of the following commands:

```
dir /a
dir /b
file /a/c 2
file /a/d 2
file /a/e 2
file /b/f 2
```

This tells the simulator to create two directories (/a and /b) and four files (/a/c, /a/d, /a/e, and /b/f). The root directory is created by default.

The output of the simulator is the location of the inodes and data blocks of all extant files and directories. For example, from the run above, we would end up seeing (with the -c flag on, to show you the results):

```
prompt> ./ffs.py -f in.example1 -c

num_groups:       10
inodes_per_group: 10
blocks_per_group: 30

free data blocks: 289 (of 300)
free inodes:      93 (of 100)

spread inodes?    False
spread data?      False
contig alloc:     1

      0000000000 0000000000 1111111111 2222222222
      0123456789 0123456789 0123456789 0123456789

group inodes     data
```

```
    0 /--------- /--------- ---------- ----------
    1 acde------ accddee--- ---------- ----------
    2 bf-------- bff------- ---------- ----------
    3 ---------- ---------- ---------- ----------
    4 ---------- ---------- ---------- ----------
    5 ---------- ---------- ---------- ----------
    6 ---------- ---------- ---------- ----------
    7 ---------- ---------- ---------- ----------
    8 ---------- ---------- ---------- ----------
    9 ---------- ---------- ---------- ----------

prompt>
```

This first part of the output shows us various parameters of the simulation, from the number of FFS cylinder groups that are created, to some policy details. But the main part of the output is the actual allocation map:

```
      0000000000 0000000000 1111111111 2222222222
      0123456789 0123456789 0123456789 0123456789

group inodes     data
    0 /--------- /--------- ---------- ----------
    1 acde------ accddee--- ---------- ----------
    2 bf-------- bff------- ---------- ----------
    3 ---------- ---------- ---------- ----------
    4 ---------- ---------- ---------- ----------
    5 ---------- ---------- ---------- ----------
    6 ---------- ---------- ---------- ----------
    7 ---------- ---------- ---------- ----------
    8 ---------- ---------- ---------- ----------
    9 ---------- ---------- ---------- ----------
```

For this instantiation, we have created a file system with 10 groups, each with 10 inodes and 30 data blocks. Each group just shows the inodes and data blocks, and how they are allocated. If they are free, a - is shown; otherwise, a different symbol is shown per file.

If you want to see a mapping of the symbols to file names, you should use the -M flag:

```
prompt> ./ffs.py -f in.example1 -c -M
```

You'll then see a table at the bottom of the output shows the meanings of each symbol:

```
symbol  inode#  filename     filetype
/            0  /            directory
a           10  /a           directory
c           11  /a/c          regular
d           12  /a/d          regular
e           13  /a/e          regular
b           20  /b           directory
f           21  /b/f          regular
```

Here, you can see the root directory is represented by the symbol /, the file /a by the symbol a, and so forth.

Looking at the output, you can thus see a number of interesting things: - The root inode is in the first slot of the Group 0's piece of the inode table - The root data block is found in the first allocated data block (Group 0) - Directory /a was placed in Group 1, directory /b in Group 2 - The files (inodes and data) for each regular file are found in the same group as their parent inodes (as per FFS)

The rest of the options let you play around with FFS and some minor variants. They are:

```
prompt> ./ffs.py -h
Usage: ffs.py [options]

Options:
  -h, --help            show this help message and exit
  -s SEED, --seed=SEED  the random seed
  -n NUM_GROUPS, --num_groups=NUM_GROUPS
                        number of block groups
  -d BLOCKS_PER_GROUP, --datablocks_per_groups=BLOCKS_PER_GROUP
                        data blocks per group
  -i INODES_PER_GROUP, --inodes_per_group=INODES_PER_GROUP
                        inodes per group
  -L LARGE_FILE_EXCEPTION, --large_file_exception=LARGE_FILE_EXCEPTION
                        0:off, N>0:blocks in group before spreading file to
                        next group
  -f INPUT_FILE, --input_file=INPUT_FILE
                        command file
  -I, --spread_inodes   Instead of putting file inodes in parent dir group,
                        spread them evenly around all groups
  -D, --spread_data     Instead of putting data near inode,
                        spread them evenly around all groups
  -A ALLOCATE_FARAWAY, --allocate_faraway=ALLOCATE_FARAWAY
                        When picking a group, examine this many groups at a
                        time
  -C CONTIG_ALLOCATION_POLICY,
```

```
--contig_allocation_policy=CONTIG_ALLOCATION_POLICY
                      number of contig free blocks needed to alloc
-T, --show_spans      show file and directory spans
-M, --show_symbol_map
                      show symbol map
-B, --show_block_addresses
                      show block addresses alongside groups
-S, --do_per_file_stats
                      print out detailed inode stats
-v, --show_file_ops   print out detailed per-op success/failure
-c, --compute         compute answers for me
```

We'll explore more of these options in the homework.