Tutorial 8

CS3103 Operating Systems

Student Name: _____

Student No. : | | | | | | | | |

Day: ☐ Monday ☐ Tuesday

Time: ☐ 10:00 - 10:50 ☐ 11:00 - 11:50 ☐ 12:00 - 12:50 ☐ 14:00 - 14:50 ☐ 18:00 - 18:50

File System

Introduction

Topics to be covered in this tutorial include:

- Study how file system stat changes as various operations take place and how the on-disk state of file system is represented.

Acknowledgement

This tutorial was adapted from OSTEP book written by Remzi and Andrea Arpaci-Dusseau at the University of Wisconsin. This free OS book is available at http://www.ostep.org.

Getting Started

1. Logging in to the Linux server

- Start the SSH client, e.g., MobaXterm or Xshell.
- Log in to the Linux server using the following details:

  Host Name: gateway.cs.cityu.edu.hk

  User Name: your EID (e.g., cctom2)

  Password: your password

💡 **Your password will not be shown on the screen as you type it, not even as a row of stars (******).**

NOTE: The shell will always give you a prompt if it is ready to accept commands. The shell prompt normally ends in a `$` sign as we use in this tutorial. Some shell prompts end in `%` or `>` instead. Never copy/type the shell prompt used in this tutorial.

NOTE: Please don't forget to log out (use the `exit` command) after you finish your work.

2. Getting the Simulators

This tutorial lets you study how file system state changes as various operations take place. The simulators are available at */public/cs3103/tutorial8* on the gateway server:

- `vsfs.py`: it simulates a file system that changes when different operations happen.

As before, follow the same copy procedure as you did in the previous tutorials to get the simulators.

Introduction to vsfs.py

This simulator, vsfs.py, allows you to get close to how file system state changes when different operations take place by user and system behaviors.

The possible operations are:

- mkdir() – creates a new directory
- create() – 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:6] [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:

```
$ ./vsfs.py -h
Usage: paging-policy.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
```

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:seed" used to specify a different random seed. For example:

```
$ ./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:

```
$ ./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:

```
$ ./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").

Questions

All questions should be answered on the separate answer sheet provided.

1. Run the simulator with some different random seeds (say 17, 18, 19, 20), and see if you can figure out which operations must have taken place between each state change.

2. Now do the same, using different random seeds (say 21, 22, 23, 24), except run with the -r flag, thus making you guess the state change while being shown the operation. What can you conclude about the inode and data-block allocation algorithms, in terms of which blocks they prefer to allocate?

3. Now reduce the number of data blocks in the file system, to very low numbers (say two), and run the simulator for a hundred or so requests. What types of files end up in the file system in this highly constrained layout? What types of operations would fail?

4. Now do the same, but with inodes. With very few inodes, what types of operations can succeed? Which will usually fail? What is the final state of the file system likely to be?