Lab 11 (CS): Alternative File Systems

Innopolis University
Course of Operating Systems

Exercise 1 (1/5)

Motivation:

- Sometimes it is needed to create a separate File System in an exiting File System and you don't want to re-partition the disk to allocate another sub-partition for this new File System.
- There's a way to create a File System on a virtual device which (the device) can then be mounted to the original File System, providing you with a File System in a File System.

Exercise 1 (2/5)

- What if you want to run a process, that you don't trust. That you don't want to see your files and data? What if you could isolate it in such a way that the process will see only what you want it to see?
- There are multiple ways to achieve it. One of which is `NameSpaces`. But today we will use another older way - `chroot`. This command allows you to change what is the *root* dir for the process.
- For example, you could create a separate file system, mount it onto a virtual device (loop) on a file, create a process and chroot it on this file. Everything that the process will then create can be transfered as a single file. Or removed. Or.. whatever.

Exercise 1 (3/5)

In a script **ex1.sh**, add command lines to do the following:

- Create a file *lofs.img* not less than 50 MiB. There're several ways (for example **dd** command).
- Setup a loop device on the created file, using `losetup`. If unsure, go to the `man losetup`.
- Create a Loop File System (LOFS) *ext4* on the created file, which, in fact, already is a device (`mkfs` commnd)
- Create a new empty directory `./lofsdisk`. Mount the created filesystem on the mount point `./lofsdisk`. See `mount` command. Voila. Now you can `cd` and use this filesystem.
- **Hint:** Some useful commands for this exercise: `dd`, `fallocate`, `mkfs`, `losetup`, `mount`.
- **Hint:** You need super user permissions to execute some commands.

Exercise 1 (4/5)

- Add two files **file1**, **file2** to the **LOFS** where **file1** contains your first name, and **file2** contains your last name.
- `[Do not need to add this step to the script]` Write a simple C program `ex1.c` which will list the contents of the `root` directory (/) (use **opendir** and **readdir**).
- Define a function get_libs() in the script to return the paths of all shared libraries of a binary file whose path is passed as the first argument `$1` to the function.
- Use the previous function to get shared libraries of commands `bash, cat, echo, ls` and add them with their shared libraries to the LOFS.
- Change the root directory of the process to the mount point of the created LOFS and run the program `ex1`. Save the output of the program in a file `ex1.txt`.
- **Note:** If you `chroot` and there are no binaries like `cd`, `touch`, etc., you may want to first copy them to the chrooted location.

Exercise 1 (5/5)

- Run the same program again (DON'T change the root directory of the process). Append the output to the file `ex1.txt`.
- What is the difference between the outputs in both cases? Add your findings to the file `ex1.txt`.
- You have to put all commands that you executed in a script *ex1.sh*.
- You have to submit all files *ex1.c*, *ex1.txt*, *ex1.sh*.
- **You must explain each command line in the script and add the explanation as a comment to the same script.**
- The results should be reproducible by running the scripts *ex1.sh*
- Don't forget to add `sudo` for commands which need that permission.

Exercise 2 (1/7)(1.5 bonus points)

- The goal of this exercise is to implement a simulation in C for a simple UNIX-like file system. This file system has the following simplifying assumptions:
  - The file system resides on a disk that is 128KiB in size.
  - There is only one root directory. No subdirectories are allowed.
  - The file system supports a maximum of 16 files.
  - The maximum size of a file is 8 blocks; each block is 1KiB in size.
  - Each file has a unique name, the file name can be no longer than 16 characters.
- The layout of your 128 KiB disk is as follows:
  - The first 1KiB block is called the super block. It stores the free block list and index nodes (inode) for each file.
  - The remaining 127 (1KiB) blocks store data blocks of the files on your file system.
  - The exact structure of the super block is as follows.

Exercise 2 (2/7)(1.5 bonus points)

- The first 128 bytes stores the free block list. Each entry in this list indicates whether the corresponding block is free or in use (if the i-th byte is 0, it indicates that the block is free, else it is in use). Initially all blocks except the super block are free.
- Immediately following the free block list are the 16 i-nodes, one for each file that is allowed in the file system. Initially, all i-nodes are free. Each i-node stores the following information:

```
struct inode{
    char name[16]; //file name
    int size; // file size (in number of blocks)
    int blockPointers[8]; // direct block pointers
    int used; // 0 => inode is free; 1 => in use
};
```

- Note that each inode is 56 bytes in size (assuming sizeof(int) = 4); Since you have 16 of these, the total size occupied by the inodes is 896 bytes. The free/used block information (mentioned above) is 128 byes. So the total space used in the super block is 1024 bytes.

Exercise 2 (3/7)(1.5 bonus points)

- You need to implement the following operations for your file system.
  - **create(char name[16], int size):** create a new file with this name and with these many blocks. (We shall assume that the file size is specified at file creation time and the file does not grow or shrink from this point on)
  - **delete(char name[16]):** delete the file with this name.
  - **read(char name[16], int blockNum, char buf[1024]):** read the specified block from this file into the specified buffer; *blockNum* can range from 0 to 7.
  - **write(char name[16], int blockNum, char buf[1024]):** write the data in the buffer to the specified block in this file.
  - **ls(void):** list the names of all files in the file system and their sizes.

Exercise 2 (4/7)(1.5 bonus points)

- We will use a 128KiB file to act as the ”disk” for your file system. I have provided a program **create_fs.c** to create this file for you. You can check it in this gist.
- The given program accepts a command line argument for the file name. For example, if file name is “disk0”, then the program creates a file with the name `disk0` in the current working directory. The program also “formats” your file system by initializing all blocks in the super block to be free and marking all 16 i-nodes to be free.

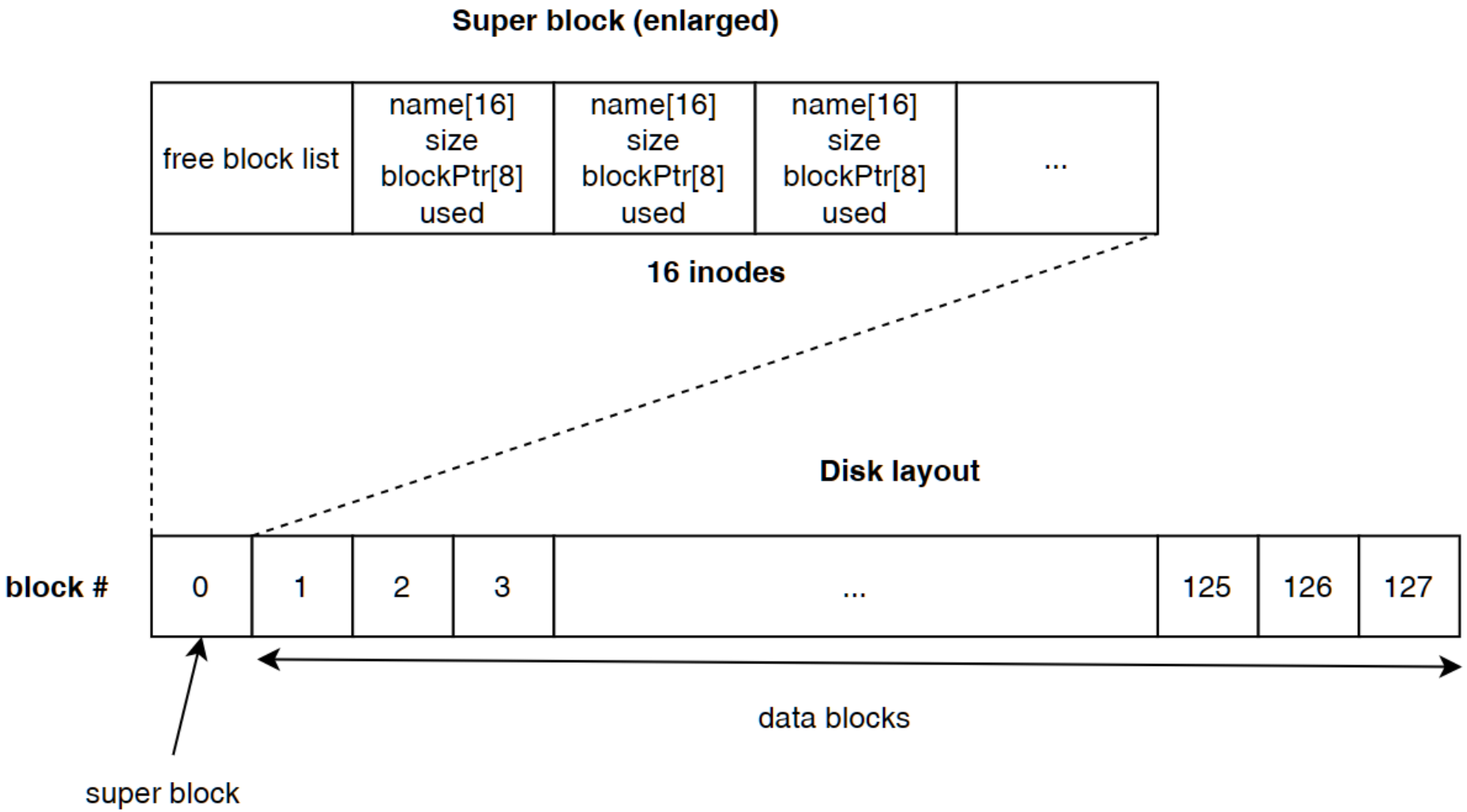
Exercise 2 (5/7)(1.5 bonus points)

- You program should take input from an input file and perform actions specified in the file, while printing out the result of each action. The format of the input file is as follows.

```
diskName  // name of the file that emulates the disk
C fileName Size //create a file of this size (# blocks)
D fileNAme  // delete this file
L   // list all files on disk and their sizes
R fileName blockNum // read this block from this file
W fileName blockNum // write to this block in the file
```

- Note that in write operation, you can write dummy content to fill in the file block (1KiB).
- Note that your file system must be persistent. If you shutdown your program and then restart it at a later time, all files on your file system must be intact.

Exercise 2 (6/7)(1.5 bonus points)



Exercise 2 (7/7)(1.5 bonus points)

- You can find a sample of the input file in this gist. You can also find there a high-level pseudo code template for the implementation.
- The implementation of this simulation need to be written to **ex2.c** file.
- Write a script **ex2.sh** to create the file system (**create_fs.c**) and process the input (**ex2.c**) from the input file.
- Submit only **ex2.c** and **ex2.sh**.

End of lab 11 (CS)