CS-202 Exercises on File Systems (L06 - L07)

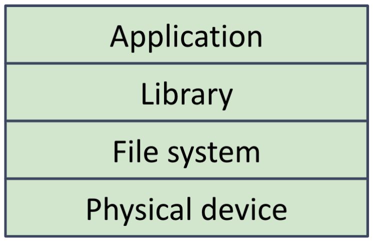
17.03.2025

This exercise set covers concepts related to file systems. We advise that you work through it sequentially, referring back to lecture slides or videos as necessary. If anything is unclear, or if you could benefit from discussing a particular concept in depth, please seek an assistant's help.

Exercise 1: File system abstractions

Fill in the blanks with terminology related to file system abstractions. The figure on the right is the file system abstraction stack, which you can refer to for hints.

- Application code uses primitives like `fopen` and `fread`, which are implemented by Library code. These primitives take as input and return objects of type `FILE *`.
- Library code uses syscalls (e.g., `open`, `read`, `lseek`). These take as input and return file descriptors, which are numerical values used to index open file metadata stored in a table. There is one such table for each process.



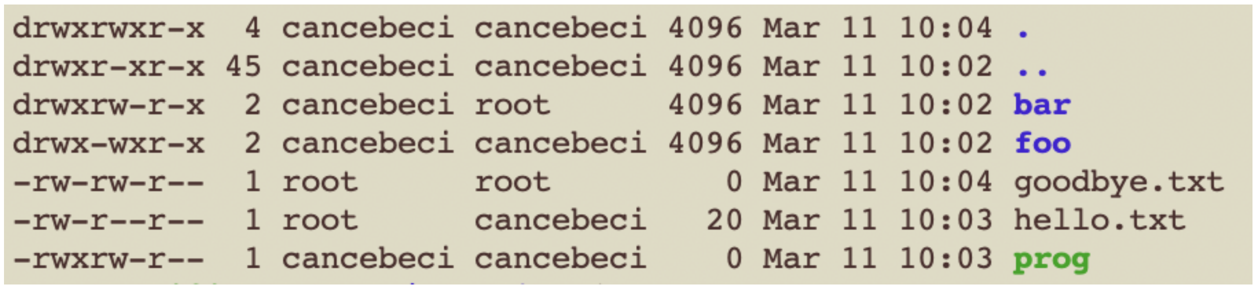
Exercise 2: File system basics

Mark the following sentences about inodes as true (T) or false (F).

(T) Inodes store metadata related to files.
(T) There may be many inodes in a file system storing information related to the same file.
(F) Among other things, an inode stores the name of the file it is related to.
(T) Multiple filenames can map to the same inode.
(F) A process can insert entries directly into a directory inode using a write syscall. (Hint: Look again at slides 32-35 that discuss the file-system interface. According to those slides, how does a process add a file to a directory?)

Exercise 3: Permission bits

Below, you are given the current directory for a process, run by user 'cancebeci', and the file descriptor table for the process. You can use the `man` command on your terminal to look up the semantics of specific syscalls and flags.



```
drwxrwxr-x  4 cancebeci cancebeci 4096 Mar 11 10:04 .
drwxr-xr-x 45 cancebeci cancebeci 4096 Mar 11 10:02 ..
drwxrw-r-x  2 cancebeci root      4096 Mar 11 10:02 bar
drwx-wxr-x  2 cancebeci cancebeci 4096 Mar 11 10:02 foo
-rw-rw-r--  1 root      root         0 Mar 11 10:04 goodbye.txt
-rw-r--r--  1 root      cancebeci   20 Mar 11 10:03 hello.txt
-rwxrw-r--  1 cancebeci cancebeci    0 Mar 11 10:03 prog
```

0	STDIN
1	STDOUT
2	STDERR
3	Inode: 55, offset: 32, mode: read
4	closed
5	Inode: 22, offset: 4918, mode: read/write

Indicate whether each syscall will succeed or fail. Assume buf is a large enough buffer and the open files are large enough that reads never reach the end of a file.

- `open("./out.txt",  O_RDWR | O_TRUNC, S_IRWXU);` will fail
- `open("./out.txt", O_CREAT | O_RDWR | O_TRUNC, S_IRWXU);` will succeed
- `open("goodbye.txt", O_RDWR | O_TRUNC, 0);` will fail
- `open("hello.txt", O_RDWR | O_TRUNC, 0);` will fail
- `open("hello.txt", O_RDONLY | O_TRUNC, 0);` will succeed
- `close(5);` will succeed
- `close(4);` will fail
- `read(3, buf, 10);` will succeed
- `write(3, buf, 10);` will fail
- `write(5, buf, 10);` will succeed

Exercise 4: Path resolution (inode walk)

The figure below depicts a disk organization consisting of 16 blocks (numbered 0-15), where the inode for the root directory ("/") is at inode 1. Block 0 is the superblock. Blocks 1-3 store the inode table, each block storing 4 inodes (block 1 stores inodes 0-4, block 2 stores inodes 4-7 and block 3 stores inodes 8-12.)

4.1: *Write down the sequence of blocks that will be read when the user asks to read the first character of "/hello/world.txt", and identify the character that will be read.*

0	1	2	3	4	5	6	7
	[... (inode 1: Location: 12 …), (inode 2: Location: 14 …)]	[….]	[… (inode 8: Location: 7 …), …]		“This block stores a text file … ”	“00101010 010101010 010101110 01011…”	“its.me”: inode 4 “world.txt”: inode 2 “baz”: inode 12
8	9	10	11	12	13	14	15
	“etc”: inode 4 “pwd”: inode 5	“This block stores another text file … ”		“foo”: inode 3 “bar”: inode 7 “hello”: inode 8		“!\nabcdefg …”	

- Block 1, to read inode 1 and find out where the directory “/” is stored.
- Block 12, to look up the inode number for “/hello”.
- Block 3, to read inode 8 and find out where the directory “/hello” is stored.
- Block 7, to lookup the inode number for “/hello/world.txt”
- Block 1, to read inode 2 and find out where the file “/hello/world.txt” is stored.
- Block 14, to read the first byte, which is “!”.

***4.2:** Which of these reads will actually trigger an interaction with the disk? Which of them may be served by a block cache?*

- Block 1, when read for the second time, can be served by the block cache.

Exercise 5: File allocation: contiguous, linked

Below you are given the contents of the root directory in a file system.

```
root
    ├── phantom_thread.mov
    └── secrets
        └── never_cursed.txt
```

The file system is stored on a disk consisting of 24 blocks of size 4KB. Block 0 is the superblock, blocks 1-7 store inodes, and 8-23 are data blocks. The size of phantom_thread.mov is 12KB, and the size of never_cursed.txt is 8KB.

***5.1.1:** Assume the file system is using contiguous allocation. Draw what the organization of the disk may look like. Clearly label each block with its contents (Which file, directory do they store? Do they include any metadata?). You can assume that directories are smaller than 4KB.*

0	1	2	3	4	5	6	7

	inodes	inodes	inodes	inodes	inodes	inodes	inodes
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23

Solution:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
`root`	`secrets`	`phantom_thread.mov`	`phantom_thread.mov`	`phantom_thread.mov`	`never_cursed.txt`	`never_cursed.txt`	
16	17	18	19	20	21	22	23

5.1.2: *What happens if the user needs to append 4KB of data to phantom_thread.mov?*
This is not possible with contiguous allocation. We would need to allocate a new file 16KB file and copy the existing 12KB over before appending the new 4KB.

5.2.1: *Assume the file system is using linked blocks for file allocation. Draw what the organization of the disk may look like. Clearly label each block with its contents (Which file, directory do they store? Do they include any metadata?). You can assume that directories are smaller than 4KB.*

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
`root`	`secrets`	`phantom_thread.mov, Next block:11`	`phantom_thread.mov Next block: 12`	`phantom_thread.mov Next block: 13`	`phantom_thread.mov No next block.`	`never_cursed.txt Next block: 15`	`never_cursed.txt Next block: 16`
16	17	18	19	20	21	22	23

Never_cursed.txt No next block.							

5.2.2: *Assume the user asked to append 4KB of data to phantom_thread.mov, and then append 4KB of data to never_cursed.txt. Draw the organization of the disks after both operations are completed.*

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
root	secrets	phantom_thread.mov, Next block:11	phantom_thread.mov Next block: 12	phantom_thread.mov Next block: 13	phantom_thread.mov Next block:17	Never_cursed.txt Next block: 15	Never_cursed.txt Next block: 16
16	17	18	19	20	21	22	23
Never_cursed.txt Next block: 18	Phantom_thread.mov, No next block.	Never_cursed.txt, No next block.					

Exercise 6: File allocation: multi-level indexing

Consider the multi-level indexing approach discussed in Lecture 7. Below is a brief recap and some problem parameters.

- Each disk block is 4KB
- Each inode contains 15 pointers.
    - 12 of these directly point to data blocks.
    - The 13th is a single-indirect pointer. It points to a block, which contains pointers to data blocks
    - The 14th is a double-indirect pointer and the 15th is a triple-indirect pointer.
- Each “pointer” is 4 bytes. (By pointer, here we mean a disk block index, not C pointers - which are typically 8 bytes).

Answer the following questions.

6.1: *What is the maximum file size that can be represented by an inode?*

- Direct pointers point to 12 x 4KB blocks, representing 48KB of data.
- The single indirect pointer points to an indirect block, which can store 4KB/4 bytes = 1K block pointers. So it represents 1K x 4KB = 4MB of data.

- The double-indirect pointer points to 1K single-indirect pointers, so it represents 1K x 4MB = 4 GB of data.
- Similarly, the triple-indirect pointer represents 4TB of data.
- The maximum file size is the total, which is 4TB + 4GB + 4MB + 48 KB.

6.2: *How many block accesses does it take to read a data byte, including the initial access to the inode? You can assume that the inode walk has already been done (i.e., we already know the number of the inode related to the file). The answer depends on which data block the byte resides in.*

- Which one requires more block accesses, reading the first byte of the first block, or the last byte of the last block (assuming the file's size is the maximum you computed in 7.1)
  - To read the first byte of the first block, we need to access the inode to read the direct pointer, then access the data block it points to. In contrast, to read the last byte of the last block, we need to traverse the indirection tree (i.e., read the triple-indirect pointer from the inode, then access the triple-indirect block, then access the double-indirect block, then the single-indirect block, and finally the data block.)
- What is the minimum number of block accesses?
  - By the reasoning above, the minimal case is reading through a direct pointer, which entails two block accesses.
- What is the maximum number of block accesses?
  - Reading through the triple-indirect pointer takes five block accesses.
- Assuming the maximum file size, if all bytes are accessed uniformly randomly, what is the average number of block accesses?
  - There are
    - 1K x 1K x 1K (2^30) blocks that require 5 accesses
    - 1K x 1K (2^20) block that require 4
    - 1K (2^10) blocks that require 3
    - And 12 blocks that require 2.
  - The answer is (2^30 x 5 + 2^20 x 4 + 2^10 x 3 + 12*x 3) / (2^30 + 2^20 + 2^10 + 12) = 4.9990

6.3: *If we were to change the approach slightly, such that an inode contains 11 direct pointers to data blocks instead of 12, and in exchange, it contains an additional quadruple-indirect pointer (four levels of indirection), what would the maximum file size be? What is the trade-off here?*

- Following the rationale from 7.1, the maximum file size would be 4PB + 4TB + 4GB + 4MB + 44 KB, where 1 PB = 1K TB.
- The tradeoff is that increasing the maximum file comes at the cost of increasing the average number of block accesses required to read a data byte (assuming maximum file size). With this setup, repeating the calculation above would yield an average close to 6. This approach also makes things slightly worse for smaller files. For instance reading a 48KB file originally only used direct pointers, but now it uses single-indirect pointers for the last 4KB.

Exercise 7: File system API

Below is a C program using the file system system call API. Show the contents of the file descriptor table of the process running this program after each syscall.

```c
int main(int argc, char *argv[]) {
    int fd = open("file.txt", O_RDONLY);
    char buf[10];
    read(fd, buf, 5);
    lseek(fd, 0, SEEK_SET);
    close(fd);
    return 0;
}
```

After open:

0	STDIN
1	STDOUT
2	STDERR
3	Inode: <some inode number>, offset: 0, mode: read

After read:

0	STDIN
1	STDOUT
2	STDERR
3	Inode: <some inode number>, offset: 5, mode: read

After seek:

0	STDIN
1	STDOUT
2	STDERR
3	Inode: <some inode number>, offset: 0, mode: read

After close:

0	STDIN
1	STDOUT

2	STDERR
3	closed

Exercise 8 (Advanced): File system API + fork

Write the output of the following program. Remember that when a process forks, the file descriptor table is copied as-is to the child process.

```
int main(int argc, char *argv[]) {
    int pid1 = 0; int pid2 = 0;
    int fd = open("file.txt", O_RDONLY);
    assert(fd >= 0);
    pid1 = fork();
    if (pid1 == 0) {
        int off = lseek(fd, 10, SEEK_SET);
        printf("statement1: offset %d\n", off);
        pid2 = fork();
    } else if (pid1 > 0) {
        (void) wait(NULL);
        printf("statement2: offset %d\n", (int) lseek(fd, 20, SEEK_CUR));
    }
    if (pid2 == 0) {
        printf("statement3: offset %d\n", (int) lseek(fd, 0, SEEK_SET));
    }
    return 0;
}
```

The fork() system call will copy the file descriptor table and so open files descriptors in both the parent and the child will point to the same open file structures.

- lseek(fd, [off], SEEK_SET) sets the offset to [offs].
- lseek(fd, [offs], SEEK_CUR) sets the offset to current offset plus the given offset value.

According to the definitions, one possible solution is:

```
statement1: offset 10
statement3: offset 0
statement2: offset 20
statement3: offset 0
```

All processes access the same file structure, since it's only the pointers in the fd table that are copied.The parent's seek pointer starts at 0, then the first child moves it to 10 and the second child moves it to 0. Both the parent and child2 run the final if block.

Due to the race condition, the 2nd child may run before the parent runs. If child2 runs before the parent, then we get the output above.

The second solution is:

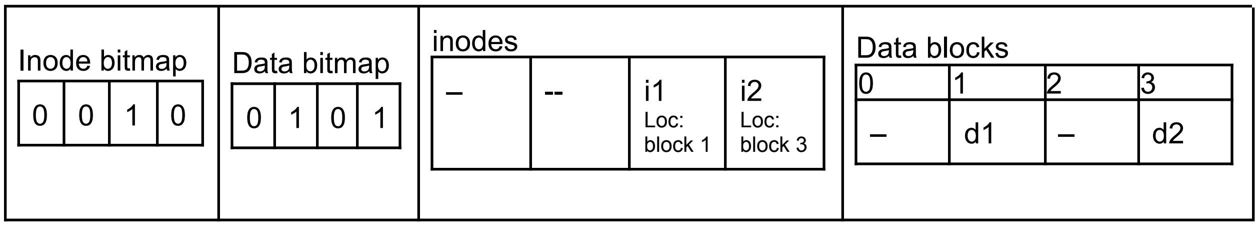
statement1: offset 10
statement2: offset 30
statement3: offset 0
statement3: offset 0

Here the child1 moves the offset to 10, the parent adds 20 and moves it to 30, then it will be moved
back to zero twice. Whether the parent or child2 runs the final if block first does not matter.

Exercise 9 (Advanced): Crash consistency

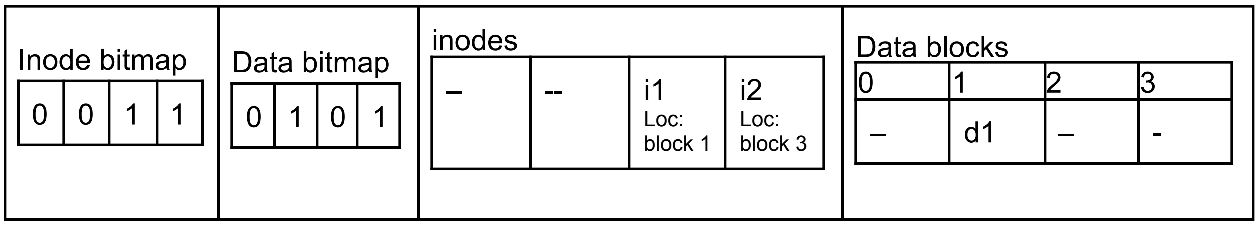
Below, you are given two disk states. Both contain inconsistencies introduced by a crash. For each example, identify the inconsistency and propose a fix.

9.1:



- The Inode bitmap says inode 3 (0-indexed) is free, but in fact the inode is used and it points to a valid data block.
    - Fix: If we can realize independently of the inode bitmap and data bitmap that i2 and d2 are used, we could update the inode bitmap to reflect this. Otherwise, i2 and d2 will be lost and we will assume that those blocks contain garbage.

9.2:



- The metadata structures are consistent but the data block 3 has not been written. Reading the data block pointed by i2 will return garbage.
    - Fix: we can not restore the file. If we can realize that the data block contains garbage, we can free the file by deallocating the inode.

Exercise 10 (Optional): Redirecting STDOUT

Below is a C program that prints “Hello World!”. Add a few lines of code at the beginning of the main function, such that this program writes “Hello World!” to a new file, hello.txt, instead of printing to the terminal. Do not change the existing code; the goal is to get the exact same printf statement to write to a file. (Hint: what happens if you close STDOUT? Try writing a program and running it to understand this better.)

```
#include <stdio.h>

int main() {
    // add code here

    printf("Hello World!");
    return 0;
}
```

There are two important things one needs to realize for this exercise. First, “printf” writes through the 0th file descriptor, which refers to the standard output stream when the process is created. Second, “open” always allocates the lowest available file descriptor, so calling it after closing stdout will reassign the 0th descriptor to the newly-opened file. Then, “printf” will write to the file instead of STDOUT.

```
// close the 0th file descriptor, which refers to STDOUT by default.
close(0);
// open will allocate the 0th file descriptor for “hello.txt”, since 0 is the lowest
available file descriptor.
open(“hello.txt”, O_CREAT | O_RDWR | O_TRUNC, S_IRWXU);
```