CS-323: OS Exercise Week 12 (Fall 2023)

December 6, 2023

Last name, first name: ______________________________

1 Files and Directories

(a) What is the inode number?

For historical reasons, low-level *name* of a file is often referred to as its inode number. It is an identifier of some kind which is visible to OS. (Also to the user, if you run `ls -i file_name` in the terminal for example)

(b) What are the default file descriptors of a process in UNIX-like systems?

Standard input, standard output and standard error; file descriptor 0, 1 and 2 respectively.

(c) Write the output of following program. Remember that the file descriptor table is copied as-is to the child process.

```c
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("child1: offset %d\n", off);
        pid2 = fork(); //another child
    } else if (pid1 > 0) {
        (void) wait(NULL);
        printf("parent: offset %d\n", (int) lseek(fd, 20, SEEK_CUR));
    }
    if (pid2==0){
        // sleep(2); // try to force ordering
        int fd2 = dup(fd);
        printf("child2: offset %d\n", (int) lseek(fd2, 0, SEEK_SET));
    }
return 0;
}
```

The `fork()` system call will copy the file descriptor table (between parent and child process), and all the open files in both will point to the same open file structures. The `dup()` system call basically duplicates the file descriptor that it takes to the first available empty slot in the file descriptor table.

SEEK_SET flag sets the offset to given offset value.
SEEK_CUR flag sets the offset to current offset plus the given offset value.

Name: ____________________________

SEEK_END flag also exists which sets offset to size of the file plus given offset value.
According to the definitions, solution (without sleep) is:

child1: offset 10
child2: offset 0
parent: offset 20
child2: 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 (that we can attempt to force with sleep) is:

child1: offset 10
parent: offset 30
child2: offset 0
child2: offset 0

If we (try to) enforce ordering, e.g., through sleep as mentioned in the code, we get the desired behavior: 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 first does not matter.

(d) How many entries were created in the open file table for the previous program? What is the maximum reference count?

All five file descriptors in the program, other than stdin/stdout/stderr, point to the same open file structure so refer to the same underlying file. The five file descriptors are due to the 3 processes and the two calls to dup. Max reference count is 5 references to the same file.

(e) What is the system call to enforce writing immediately? Why is it useful?

`fsync()`. Suppose there is a database engine is running on the machine. Considering the speed of the transactions and importance of the consistency of database states, a write operation (read as well but less critical) should be executed a soon as possible. Just think that you received money from your parents for Christmas but it is not visible in your bank account immediately or worse, lost due to a crash!

(f) Explain the information returned in each column by the `ls` command:

```
prompt> ls -al
total 8
drwxr-x--- 2 sanidhya profs 4096 Apr 30 16:17 ./
drwxr-x--- 26 sanidhya profs 4096 Apr 30 16:17 ../
```

file permissions (drwxr-x—), d is for directory (f for file l for link etc)
file permissions (drwxr-x—) group by 3 characters (after the first indicator d,f,l,.). Each of 3 characters are set for **file owner**, **group user** and **everyone else** respectively. **r** for read, **w** for write, **x** for executable permission.
So, the user **`sanidhya`** has read/write/execute and the group **`profs`** has read/execute permissions. Other users have no permissions. number of (hard) links (2 for first, 26 for second)
owner name (sanidhya)
owner group (profs)
file size in bytes
time of last modification (Apr 30 16:17)
file/directory name.

Name: ____________________________

2 File System Implementation

(a) What kind of data structures are utilized by file systems to organize its data and metadata?

Files are array/blocks of bytes. Directories are specific type of files with list of tuples of (name-inode) mappings but in the end they are also array/blocks of bytes. Metadata of a file is an object/struct with fixed size and with predefined attributes. Those can be stored in array-like, list, or in a tree-based structures. Access rights/capabilities of accessors (user, process etc.) can be listed in a table with specific flags or those flags can be built into metadata attributes. To inform the OS consistently, we need to keep track of which files are in use (in memory) with open file table which is another array like table. There's also caches present in the OS for speeding up path lookups and caching recently read file blocks. As always, there's a lot going on!

(b) Suppose the block size is 4KB and an inode size is 128 bytes. How many inodes can a block hold? How many inodes an inode bitmap can keep track of?

A block can hold 4096 / 128 = 32 inodes. The inode bitmap is one block can track 32K objects since it is 4KB x 8 = 32Kbits.

(c) Disks are not byte addressable, but rather consist of a large number of addressable sectors of 512 (or more) bytes with atomic read/write features. Consider a file system partition layout, where block size is 4KB and an inode size is 256 bytes, with 5 inode blocks, one bitmap block for both data and inodes and of course a superblock. Calculate the sector where inode 40 resides.

Such calculations are actually used by device drivers to locate actual sectors, and then of course such information can be used for scheduling. In a standard layout, superblock is located in the beginning of the filesystem partition, and then bitmaps come. Afterwards, inodes are stored and finally the actual data portion starts. The following figure is taken from the book.

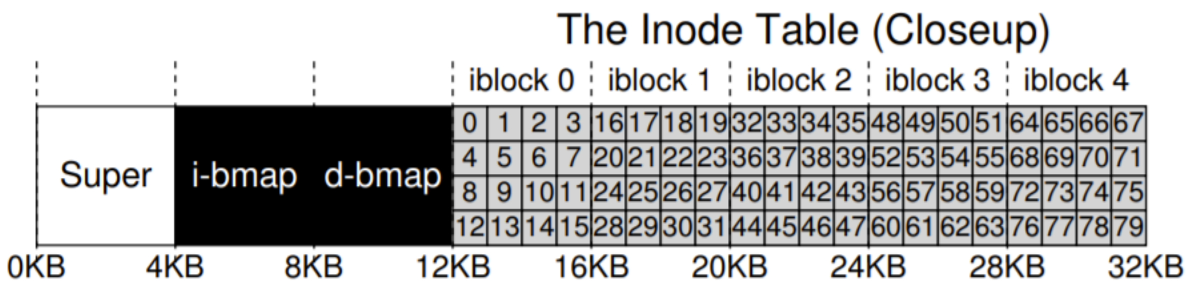

Figure 1: FS Partition Layout. [OSTEP]

As you can see, inodes start from 12KB assuming that file system partition offset is 0. Offset of 40^{th} inode is 40 x 256 (size of inode) = 10240 bytes after the inode blocks are started. There are 12KB / 512B = 24 sectors until inodes start. And, 10240B / 512B = 20 sectors for the offset of inode 40. So the answer is the sector 24 + 20 = 44.

(d) To support very big files, file systems designers developed multilevel indirect pointers (disk addresses of data blocks belonging the same file) stored in inodes for each file. Assuming 4KB blocks and 4-byte address spaces, how much data can a double indirect pointer index? If a file has 8 direct blocks, 1 indirect pointer and 1 double indirect pointer, what is the maximum file size?

Address pointers of data blocks belonging to a file are kept in the inode struct. Those pointers are called direct pointers and generally present in fixed number in an inode (assuming that small files are most common). Indirect pointers are those that point to a block that contains more pointers instead of pointing to the actual data block. Double indirect pointers basically means two level of such indirect pointers. One block can hold 4KB / 4 = 1024 block pointers. So, a double indirect pointer could address 1024 x 1024 x 4KB = 4GB of data.

The maximum size would be achieved by using all of the pointers: (8 direct + 1024 indirect + 1024^2 double indirect) x 4KB, which is just a tiny bit more than 4GB.