File Systems: The Implementation

Azza Abouzied

From Last Class: Interface Decisions

Interface considerations

1. **Structure:** what does the file look like? and in turn what operations would it support?
2. **File Naming:** how do you refer to a file?
3. **File Typing:** how do you encode type? Should you?
4. **Block Access:** How do you expose the bytes of file? Sequentially? Directly?
5. **Organization:** Directories? Links?
6. **Hardware Abstraction:** Device-based mount points?
7. **Special Files:** Would you include more than files?
8. **Access Control:** How do you represent user privileges?

The Implementation

A Collection of ...

On-disk Data Structures:

1. to track **free** blocks
2. to track **blocks that hold a file's** contents
3. to represent **the tree of named directories and files**

Other functions of an FS

1. **Crash recovery.**

2. Different processes operate on the file system at the same time

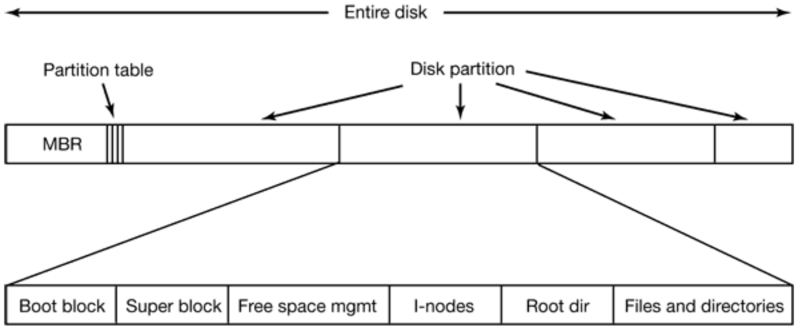
   1. coordinate access to maintain invariants.
   2. provide a measure of consistency

3. **Caching.**

Accessing a disk is orders of magnitude slower than accessing memory: maintain an in-memory cache of popular blocks.

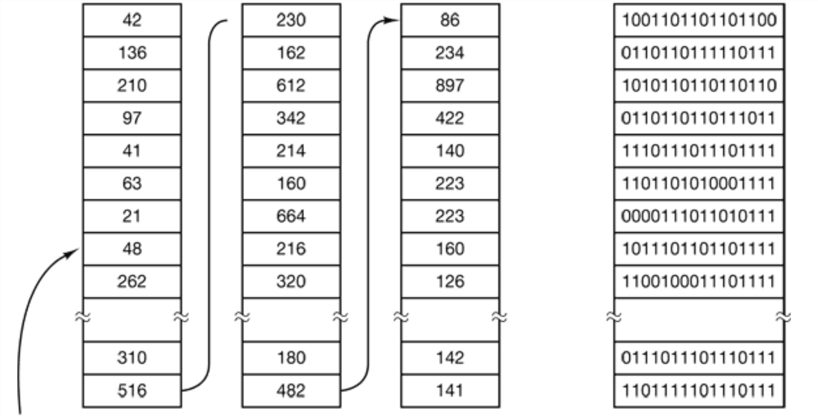
On Disk Data Structures: Free Blocks

Disk Layout



Entire disk
Partition table
Disk partition
MBR
Boot block
Super block
Free space mgmt
I-nodes
Root dir
Files and directories

Free Blocks



A 1-KB disk block can hold 256 32-bit disk block numbers

A bitmap

On Disk Data Structures: Blocks of a file

Blocks of a file

Option A: Contiguous Allocation

Each file is a sequence of consecutive blocks. Only need to store first and last blocks.

Pros & Cons:

1. We need to know file size ahead of time
2. We get fast sequential reads and easy random access
3. Deleting or shrinking files causes fragmentation
4. Hard to grow files

What is this good for?

Blocks of a file

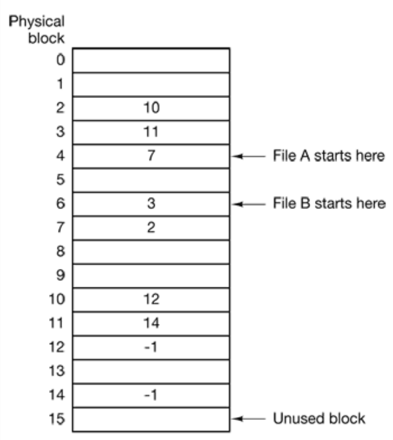
Option B: Linked Lists
File header points to first block. Every block points to the next

Pros & Cons:

1. Growth problem resolved but
2. Doesn't work well for random access
3. Blocks end up with weird sizes ($2^{12} - 4 = 4092$ bytes)

Option C: File Allocation Table FAT

Separate linked lists from files



Pros & Cons:

1. If FAT resides in memory random access is okay
2. Size of FAT depends on disk size! 20GB disk requires 80MB of RAM for FAT ... can be pageable

Blocks of a file

Option D: Index Nodes (inodes)

Each file is associated with an inode, which is identified by an integer number, often referred to as an i-number or inode number and contains an index (array of pointers) to the blocks of a file.

1. Why is it better than FAT?
2. What is the drawback for short files?
3. What is the drawback for big files?

Blocks of a file

Option D: Index Nodes (inodes)
Each file is associated with an inode, which is identified by an integer number, often referred to as an i-number or inode number and contains an index (array of pointers) to the blocks of a file.

1. Why is it better than FAT? Don't need to load inodes for closed files.
2. What is the drawback for short files? Most inodes will be empty.
3. What is the drawback for big files? Inode size is limited

How to get the inode number and contents of a file

```
ls -i <file/dir> //gets inode number
stat -x <file/dir> //gets the contents of the inode
```

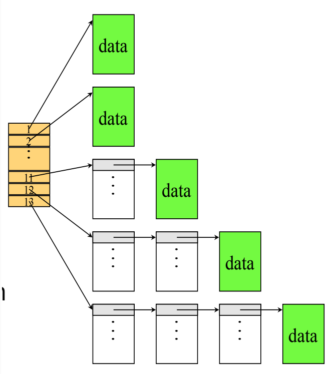
The Unix inode structure

1. 10 direct pointers
2. 11: 1-level indirection
3. 12: 2-level indirection
4. 13: 3-level indirection
5. **What is the maximum file size limit with 1 KB blocks?**

$$1024 * (10 + 1024/4 + (1024/4)^2$$

$$+(1024/4)^3) = 17GB$$

6. What if we have 8KB blocks?
7. In earlier unix versions, we had a sneaky problem due to where we store inodes.



What else is an inode?

1. File size
2. Device ID
3. File type
4. Protection bits, setuid (“set user ID upon execution”) and setgid (“set group ID upon execution”) bits
5. Link count: for **hard links** to the file
6. UID: file owner
7. GID: group ID of owner
8. Accessed and Modified timestamps of data and inode
9. The pointers
10. **What about filename?**

On Disk Data Structures: directories

Named files/directories

How do we map a filename to an inode?

Directory: Table mapping names to inodes/FAT entries/other directories

Read a byte from /nyuados/lab5

1. Read inode and first data block of /
2. Then inode and first data block of nyuados
3. Then inode and first data block of lab5

Write to a file

1. Read Inodes of the directories and directory file
2. Read/create the inode of the file
3. Write back the directory and the file

How to minimize all these IOs?

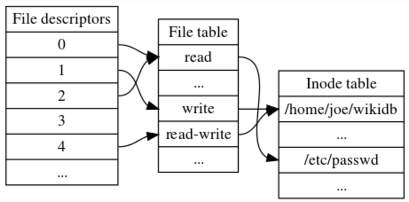
What happens when you open a file?

```
#include <stdio.h>
#include <errno.h>
int main (void) {
  FILE *fp;
  fp = fopen ("test.txt","w");
  if (fp == NULL) {
    printf ("File not created, errno = %d\n", errno);
    return 1;
  }
  ...
  fclose (fp);
  return 0;
}
```

You can check the file descriptors with ls -l /proc/PID/fd

File Descriptors

1. Each PCB has a pointer to a **file descriptor table.**
2. These index into a system-wide **file table**: a table of all files. You can see all opened files with lsof -u user. The table records the access mode for each file.
3. The **file table** indexes into a third table: the inode table that describes the actual underlying files.



The Original Unix Layout

The UNIX physical disk layout

The features:

1. Block size was 512 bytes: why such a small size?
2. Inodes on outermost cylinder
3. Data block inside
4. Linked List for Free blocks

The issues

1. Large index. Why?
2. Fixed possible number of files. Why?
3. Inodes far from data blocks. Do we always read data blocks when we read inodes?
4. Inodes for a directory not close together. What does this hurt?
5. Sequential access hurt: poor bandwidth 20KB/s. Why?

Questions?