BUCKNELL UNIVERSITY
Computer Science

CSCI 315 Operating Systems Design

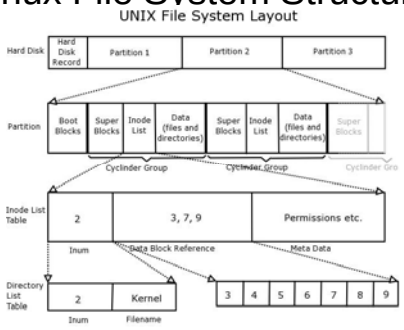
File Systems Implementation -- Part 2

Notice: The slides for this lecture have been largely based on Professor Perrone's notes. Revised by Xiannong Meng.

inode in Linux/Unix

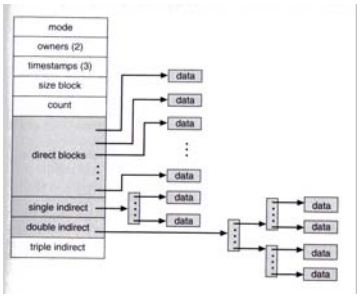
- File control blocks in Linux are called inode (has nothing to do with Apple) meaning *index node*.
  - http://www.tldp.org/LDP/tlk/ds/ds.html
- Each file in Linux has a unique control block (inode)
  - "ls -i" shows the inode number of the files
  - "stat file_name" shows inode number and other information

Linux File System Structure



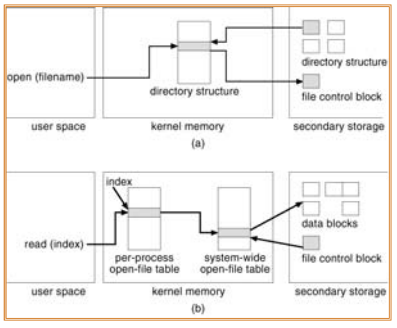
http://www.learnlinux.org.za/courses/build/internals/ch08s04.html

Linux inode Structure



http://www.csie.ntu.edu.tw/~pangfeng/System%20Programming/Lecture_Note_2.htm

In-Memory File System Structures



file open

file read

Per-process open-file table part of PCB

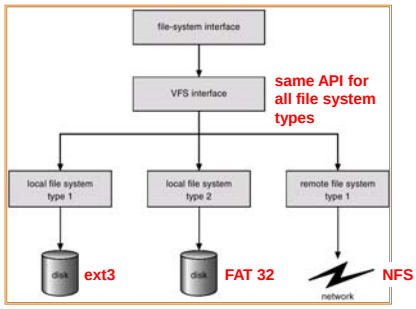
An Example

- Consider 'open("hello.txt", O_RDONLY)' in
  - http://www.eg.bucknell.edu/~cs315/2013-fall/code-examples/files/file-stream.c
- Where does the file "hello.txt" reside?
- On a remote computer!
  - unixspace.eg.bucknell.edu
  - Use "df" to find out
- Implication?
- File system has to work with networked files

Virtual File Systems

- **Virtual File Systems (VFS)** provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.

Schematic View of Virtual File System



Virtual File System Implementation

- For example, Linux has four object types:
  - *inode, file, superblock, dentry*
- VFS defines set of operations on the objects that must be implemented, inode -> vnode
  - Every object has a pointer to a function table
    - Function table has addresses of routines to implement that function on that object
    - For example:
    - **`int open()`** -- Open a file
    - **`int close()`** -- Close an already-open file
    - **`ssize t read() --`** Read from a file
    - **`ssize t write()`** -- Write to a file
    - **`int mmap()`** -- Memory-map a file

Directory Implementation

The directory is a **symbol table** that maps file names to file control block which has pointers that lead to the blocks comprising a file.

- **Linear list** of file names with pointer to the data blocks:
  - simple to program, but…
  - time-consuming to execute.
- **Hash Table:**
  - decreases directory search time,
  - *collisions* – situations where two file names hash to the same location,
  - fixed size.

Linux dirent.h

```
struct dirent {
        ino_t          d_ino;       /* inode number */
        off_t          d_off;       /* offset to the next dirent */
        unsigned short d_reclen;    /* length of this record */
        unsigned char  d_type;      /* type of file; not
                                       supported by all file
                                       system types */
        char           d_name[256]; /* filename */
};
```


Code Example to Access Directories

```
dirp = opendir(dname);
if (dirp != NULL) { // it is a directory
   printf("directory : %s\n",dname);
   for (dp = readdir(dirp); NULL != dp; dp = readdir(dirp)) {
      printf("%s\n", dp->d_name);
      print_ftype(dp->d_type);
}
closedir (dirp);  }
```

```
[xmeng@polaris files]$ ./a.out ./
directory : ./
name: [.] type: [directory]
name: [..] type: [directory]
name: [file-test.c] type: [regular]
name: [list_dir.c] type: [regular]
name: [hello.txt] type: [regular]
name: [file-stream.c] type: [regular]
name: [a.out] type: [regular]
name: [test] type: [directory]
name: [list_dir.c~] type: [regular]
[xmeng@polaris files]$
```

http://www.eg.bucknell.edu/~cs315/2013-fall/code-examples/files/list_dir.c

Allocation Methods

An **allocation method** refers to how disk blocks are allocated for files. We'll discuss three options:

- Contiguous allocation,
- Linked allocation,
- Indexed allocation.

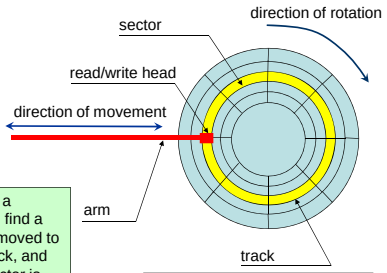
Disk Structure

Points to consider:

Sector sizes (number of bits per sector) are fixed in most disks, which means the data density is lower on outside tracks.

Newer formats, e.g., *zone-bit-recording*, uses variable size sectors so sectors have similar data density.

The disk rotates at a constant speed. To find a block, the head is moved to the appropriate track, and then the correct sector is found as the disk rotates.



Organization of a disk surface

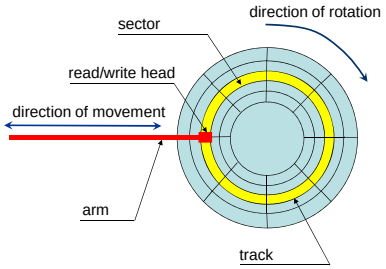
Disk Structure

The disk rotation is given in rotations per minute (RPM).

The time to find a track is proportional to the distance the head must travel.

The average time to find a sector within a track is roughly **half the time for a full rotation**.

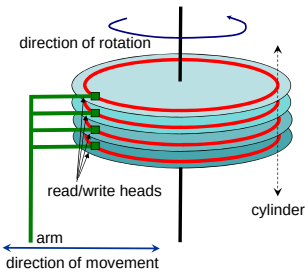
Question: If the time to move from track i to track $(i+1)$ is given by δ, assuming that the disk head is at track 0 (all the way out), could you calculate the time to get to sector 4 in track 5?



Organization of a disk surface

Disk Structure

Multi-surface disk



A cylinder is the collection of all the same tracks across all the multiple disk surfaces.

There is a time associated with turning heads on and off so that a different surface can be accessed. We call this overhead the **head-switching** time.

The time to move the arm to read another cylinder is due to the mechanics of the arm. It is certainly much large than the head-switching time, which is due to electronics only.

Question: How should one organize data across multiple surfaces to minimize access overhead?