Chapter

File Systems

Yunmin Go

School of CSEE

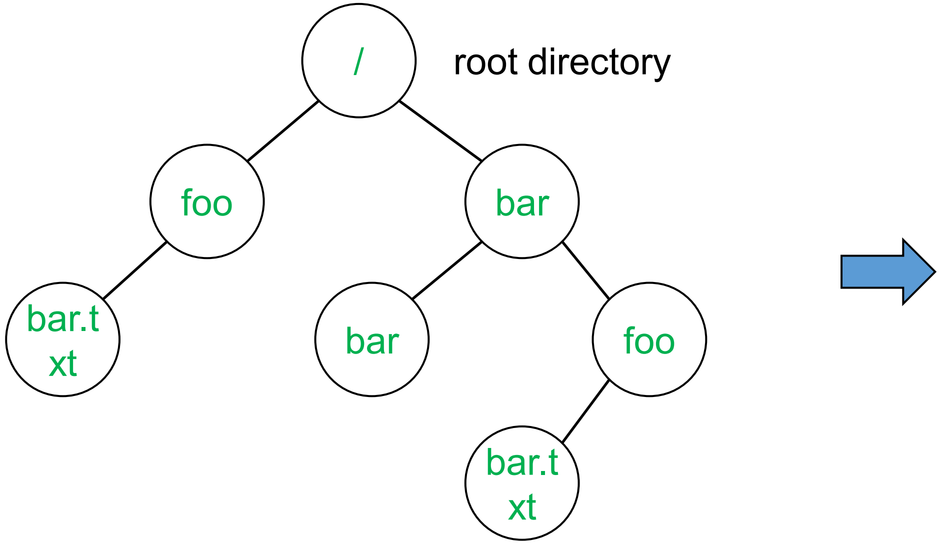
HANDONG GLOBAL UNIVERSITY

Agenda

- Files and Directories
- File System Structure & Operations
- File System Implementation

Concepts

- File
  - File is simply a linear array of bytes.
  - Each file has low-level name as ‘inode number’
- Directory
  - A file
  - A list of <user-readable filename, low-level filename> pairs



An Example Directory Tree

vaild files :
/foo/bar.txt
/bar/foo/bar.txt

vaild directory :
/
/foo
/bar
/bar/bar
/bar/foo/

File Manipulation APIs

Functions	Description
`open()`	Opens or creates a file
`close()`	Closes an open file descriptor
`read()`	Reads data from a file
`write()`	Writes data to a file
`lseek()`	Moves the file offset
`dup(), dup2()`	Duplicates file descriptors
`fsync(), fdatasync()`	Forces buffered changes to disk
`truncate(), ftruncate()`	Changes file size
`access()`	Checks file access permissions
`chmod(), fchmod()`	Changes file permissions
`chown(), fchown()`	Changes file ownership
`unlink()`	Deletes a file
`rename()`	Renames or moves a file
`stat(), fstat(), lstat()`	Gets file metadata
`readlink()`	Reads the target of a symbolic link
`symlink()`	Creates a symbolic link
`link()`	Creates a hard link

Directory Manipulation APIs

Functions	Description
`mkdir()`	Creates a new directory
`rmdir()`	Removes an empty directory
`opendir()`	Opens a directory (returns DIR*)
`readdir()`	Reads entries in a directory sequentially
`closedir()`	Closes an open directory
`chdir()`	Changes the current working directory
`getcwd()`	Gets the current working directory path
`scandir()`	Scans and optionally sorts directory entries
`dirfd()`	Gets a file descriptor from a DIR*

Open File Table

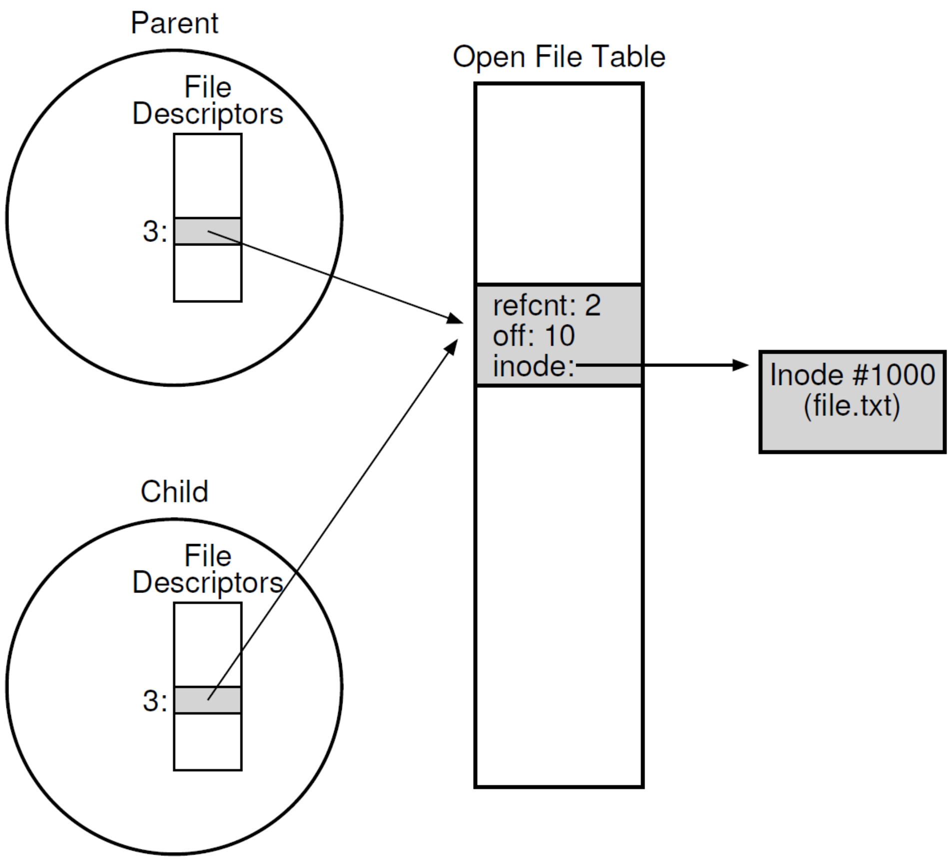
- Child process inherits the file descriptor table of the parent.

```
int main(int argc, char *argv[]) {
    int fd = open("file.txt", O_RDONLY);
    assert(fd >= 0);

    int rc = fork();
    if (rc == 0) {
        rc = lseek(fd, 10, SEEK_SET);
        printf("child: fd %d, offset %d\n", fd, rc);
    } else if (rc > 0) {
        (void) wait(NULL);
        printf("parent: fd %d, offset %d\n", fd,
                (int) lseek(fd, 0, SEEK_CUR));
    }

    return 0;
}
```

```
yunmin@yunmin:~/ch11$ ./fork-seek
child: fd 3, offset 10
parent: fd 3, offset 10
```



fsync()

- Persistency
  - `write()`: write data to the buffer. Later, save it to the storage.
  - Some applications require more than eventual guarantee. Ex) DBMS

- `fsync()`: the writes are forced immediately to disk.

```
#include <unistd.h>
int fsync(int fd);
```

  - Synchronize a file's in-core state with storage device
  - Example

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
int rc = write(fd, buffer, size);
rc = fsync(fd);
```

stat()

- `stat()`: get file status

```
#include <sys/stat.h>
int stat(const char *restrict pathname,
         struct stat *restrict statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *restrict pathname,
          struct stat *restrict statbuf);
```

  - `stat()` retrieves information about the file pointed to by pathname
  - `fstat()` is identical to `stat()`, except that the file about which information is to be retrieved is specified by the file descriptor *fd*.
  - `lstat()` is identical to `stat()`, except that if pathname is a symbolic link, then it returns information about the link itself, not the file that the link refers to.

stat()

- stat(): get file status
  - stat structure: file status

```
#include <sys/stat.h>
struct stat {
    dev_t      st_dev;      /* ID of device containing file */
    ino_t      st_ino;      /* Inode number */
    mode_t     st_mode;     /* File type and mode */
    nlink_t    st_nlink;    /* Number of hard links */
    uid_t      st_uid;      /* User ID of owner */
    gid_t      st_gid;      /* Group ID of owner */
    dev_t      st_rdev;     /* Device ID (if special file) */
    off_t      st_size;     /* Total size, in bytes */
    blksize_t  st_blksize;  /* Block size for filesystem I/O */
    blkcnt_t   st_blocks;   /* Number of 512 B blocks allocated */

    struct timespec  st_atim;  /* Time of last access */
    struct timespec  st_mtim;  /* Time of last modification */
    struct timespec  st_ctim;  /* Time of last status change */
};
```


https://man7.org/linux/man-pages/man3/stat.3type.html

stat()

```c
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <time.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n",
                argv[0]);
        return 1;
    }

    const char *filename = argv[1];
    struct stat file_stat;

    if (stat(filename, &file_stat) == -1) {
        perror("stat");
        return 1;
    }
```

```c
    // File status
    printf("File: %s\n", filename);
    printf("Size: %ld bytes\n", file_stat.st_size);
    printf("Inode: %ld\n", file_stat.st_ino);
    printf("Permissions: %o\n", file_stat.st_mode & 0777);
    printf("Links: %ld\n", file_stat.st_nlink);
    printf("Owner UID: %d\n", file_stat.st_uid);
    printf("Group GID: %d\n", file_stat.st_gid);
    printf("Last accessed: %s", ctime(&file_stat.st_atime));
    printf("Last modified: %s", ctime(&file_stat.st_mtime));
    printf("Last status change: %s", ctime(&file_stat.st_ctime));

    return 0;
}
```

```
yunmin@yunmin:~/ch11$ ./filestat file
File: file
Size: 6 bytes
Inode: 811949
Permissions: 664
Links: 1
Owner UID: 1000
Group GID: 1000
Last accessed: Sat Jun  7 23:08:34 2025
Last modified: Sat Jun  7 23:08:34 2025
Last status change: Sat Jun  7 23:08:34 2025
yunmin@yunmin:~/ch11$ stat file
  File: file
  Size: 6               Blocks: 8          IO Block: 4096   regular file
Device: 8,2     Inode: 811949      Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/  yunmin)   Gid: ( 1000/  yunmin)
Access: 2025-06-07 23:08:34.326295363 +0900
Modify: 2025-06-07 23:08:34.326295363 +0900
Change: 2025-06-07 23:08:34.326295363 +0900
 Birth: 2025-06-07 23:08:34.326295363 +0900
```

stat()

- The `stat.st_mode` field contains the file type and mode.
  - Mask values for file type

```
S_IFMT     0170000   bit mask for the file type bit field

S_IFSOCK   0140000   socket
S_IFLNK    0120000   symbolic link
S_IFREG    0100000   regular file
S_IFBLK    0060000   block device
S_IFDIR    0040000   directory
S_IFCHR    0020000   character device
S_IFIFO    0010000   FIFO
```

```
stat(pathname, &sb);
if ((sb.st_mode & S_IFMT) == S_IFREG) {
    /* Handle regular file */
}
```

```
stat(pathname, &sb);
if (S_ISREG(sb.st_mode)) {
    /* Handle regular file */
}
if (S_ISDIR(sb.st_mode)) {
    /* Handle directory */
}
```

https://man7.org/linux/man-pages/man7/inode.7.html

stat()

- The `stat.st_mode` field contains the file type and mode.
  - Mask values for file mode

```
S_ISUID     04000   set-user-ID bit (see execve)
S_ISGID     02000   set-group-ID bit
S_ISVTX     01000   sticky bit

S_IRWXU     00700   owner has read, write, and execute
                              permission
S_IRUSR     00400   owner has read permission
S_IWUSR     00200   owner has write permission
S_IXUSR     00100   owner has execute permission

S_IRWXG     00070   group has read, write, and execute
                    permission
S_IRGRP     00040   group has read permission
S_IWGRP     00020   group has write permission
S_IXGRP     00010   group has execute permission

S_IRWXO     00007   others (not in group) have read, write,
                    and execute permission
S_IROTH     00004   others have read permission
S_IWOTH     00002   others have write permission
S_IXOTH     00001   others have execute permission
```

```
yunmin@yunmin:~/ch11$ ls -al
total 52
drwxrwxr-x  2 yunmin yunmin  4096 Jun  9 06:33 .
drwxr-x--- 32 yunmin yunmin  4096 Jun  7 23:08 ..
-rwxrwxr-x  1 yunmin yunmin 16208 Jun  7 23:16 filestat
-rw-rw-r--  1 yunmin yunmin   997 Jun  7 23:16 filestat.c
-rw-rw-r--  1 yunmin yunmin     6 Jun  9 06:32 file.txt
-rwxrwxr-x  1 yunmin yunmin 16224 Jun  9 06:33 fork-seek
-rw-rw-r--  1 yunmin yunmin   485 Jun  9 06:33 fork-seek.c
```

https://man7.org/linux/man-pages/man7/inode.7.html

opendir() & readdir()

- opendir(): open a directory
- readdir(): read a directory
  - Directory is a file, but with a specific structure.
  - When reading a directory, we use specific system call other than read().

DIR *: a pointer to the directory stream

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir("."); /* open current directory */
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) { /* read one directory entry */
        printf("%d %s\n", (int) d->d_ino, d->d_name);
    }
    closedir(dp); /*close current directory */
    return 0;
 }
```

https://man7.org/linux/man-pages/man3/opendir.3.html
https://man7.org/linux/man-pages/man3/readdir.3.html

opendir() & readdir()

- `struct dirent`: structure of the directory entry

```
struct dirent {
    ino_t          d_ino;       /* Inode number */
    off_t          d_off;       /* Current location in directory stream */
    unsigned short d_reclen;    /* Length of this record */
    unsigned char  d_type;      /* Type of file; not supported
                                   by all filesystem types */
    char           d_name[256]; /* Null-terminated filename */
};
```

link()

- `link()`: make a new name for a file (i.e., hard link)
  - Create hard link named file2.

```
yunmin@yunmin:~/ch11$ echo hello > file
yunmin@yunmin:~/ch11$ cat file
hello
yunmin@yunmin:~/ch11$ ln file file2
yunmin@yunmin:~/ch11$ ls -l
total 28
-rw-rw-r-- 2 yunmin yunmin     6 Jun  8 00:14 file
-rw-rw-r-- 2 yunmin yunmin     6 Jun  8 00:14 file2
-rwxrwxr-x 1 yunmin yunmin 16208 Jun  7 23:16 filestat
-rw-rw-r-- 1 yunmin yunmin   997 Jun  7 23:16 filestat.c
yunmin@yunmin:~/ch11$ cat file
hello
yunmin@yunmin:~/ch11$ cat file2
hello
yunmin@yunmin:~/ch11$ ls -i file file2
811752 file  811752 file2
```

Create a hard link, link `file` to `file2`

Two files have same inode number, but two human name (file, file2)

  - After creating a hard link to file, old and new files have no difference.
  - Thus, to remove a file, we call `unlink()`.

unlink()

- `unlink()`: delete a name and possibly the file it refers to

```
#include <unistd.h>
int unlink(const char *pathname);
```

  - `unlink()` deletes a name from the filesystem. ('rm' calls `unlink()`)
  - If that name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse.
  - If the name was the last link to a file but any processes still have the file open, the file will remain in existence until the last file descriptor referring to it is closed.
  - If the name referred to a symbolic link, the link is removed.

unlink()

- What `unlink()` is doing ?
  - Check reference count within the inode number.
  - Remove link between human-readable name and inode number.
  - Decrease reference count.
    - When only it reaches zero, It delete a file (free the inode and related blocks)

```
$ echo hello > file               /* create file*/
$ stat file
... Inode: 811752 Links: 1 ... /* Link count is 1 */
$ ln file file2                   /* Hard link file2 */
$ stat file
... Inode: 811752 Links: 2 ... /* Link count is 2 */
$ stat file2
... Inode: 811752 Links: 2 ... /* Link count is 2 */
$ ln file2 file3                  /* Hard link file3 */
$ stat file
... Inode: 811752 Links: 3 ... /* Link count is 3 */
$ rm file                         /* Remove file */
$ stat file2
... Inode: 811752 Links: 2 ... /* Link count is 2 */
$ rm file2                        /* Remove file2 */
$ stat file3
... Inode: 811752 Links: 1 ... /* Link count is 1 */
$ rm file3
```

Symbolic Links

- Symbolic link
  - Special file that contains path to the source directory.
  - Hard link cannot create to a directory.
  - Hard link cannot create to a file to other partition.

```
yunmin@yunmin:~/ch11$ echo hello > file
yunmin@yunmin:~/ch11$ ln -s file file2
yunmin@yunmin:~/ch11$ ls -l
total 24
-rw-rw-r-- 1 yunmin yunmin     6 Jun  8 23:50 file
lrwxrwxrwx 1 yunmin yunmin     4 Jun  8 23:51 file2 -> file
-rwxrwxr-x 1 yunmin yunmin 16208 Jun  7 23:16 filestat
-rw-rw-r-- 1 yunmin yunmin   997 Jun  7 23:16 filestat.c
yunmin@yunmin:~/ch11$ ls -i file file2
811752 file  811906 file2
yunmin@yunmin:~/ch11$ cat file2
hello
yunmin@yunmin:~/ch11$ rm file
yunmin@yunmin:~/ch11$ cat file2
cat: file2: No such file or directory
```

Option -s: Create a symbolic link

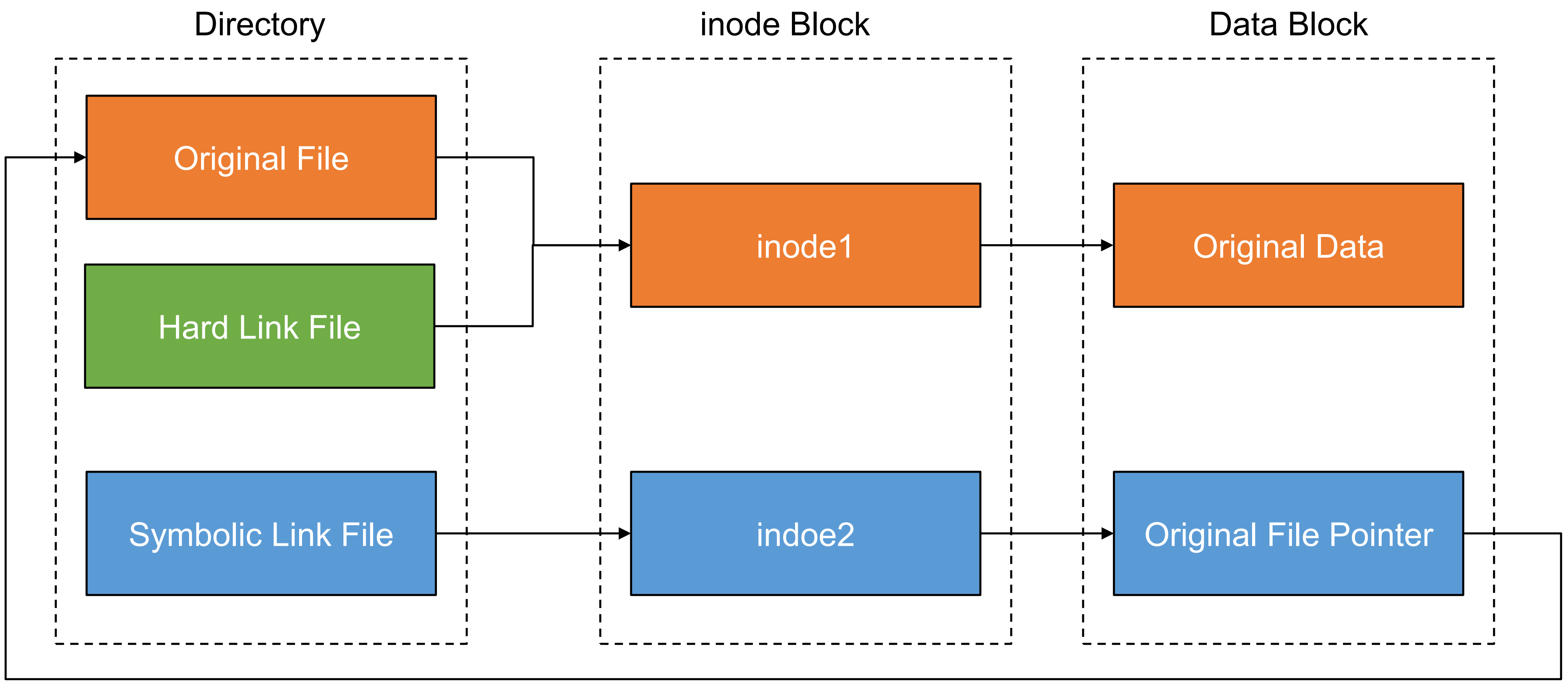
Regular file

Symbolic link

Different inodes

Symbolic link is subject to the dangling reference.

Hard Link vs. Symbolic Link

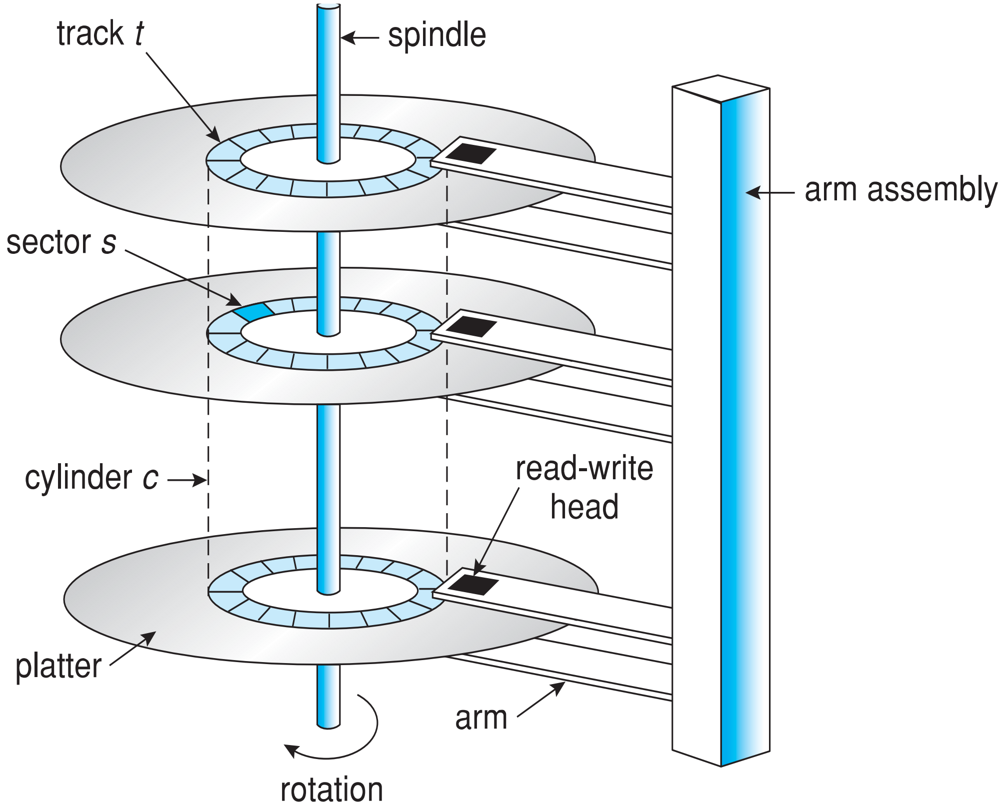


Agenda

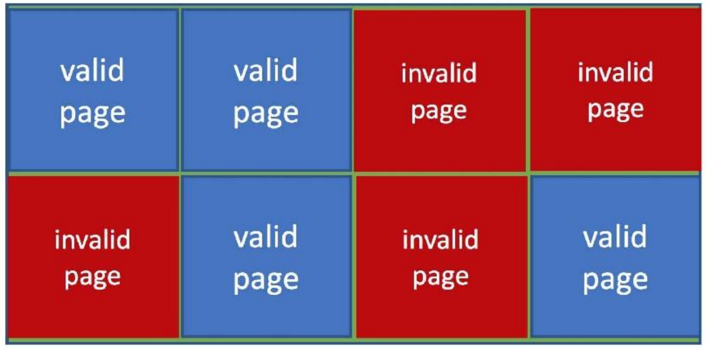
- Files and Directories
- File System Structure & Operations
- File System Implementation

Secondary Storages

- Bulk of secondary storage for modern computers is **hard disk drives** (**HDDs**) and **nonvolatile memory** (**NVM**) devices



A 3.5-inch SSD circuit board



NAND block with valid and invalid pages

Disk Drives Address Mapping

- Disk drives are addressed as large one-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer
  - Low-level formatting creates **logical blocks** on physical media
- The one-dimensional array of logical blocks is mapped onto the sectors or pages of the device
  - Sector 0 is the first sector of the first track on the outermost cylinder on HDD
    - The mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders, from outermost to innermost.
  - For NVM the mapping is from a tuple of chip, block, and page to an array of logical blocks
  - A logical block address (LBA) is easier for algorithms to use than a sector, cylinder, head tuple or chip, block, page tuple

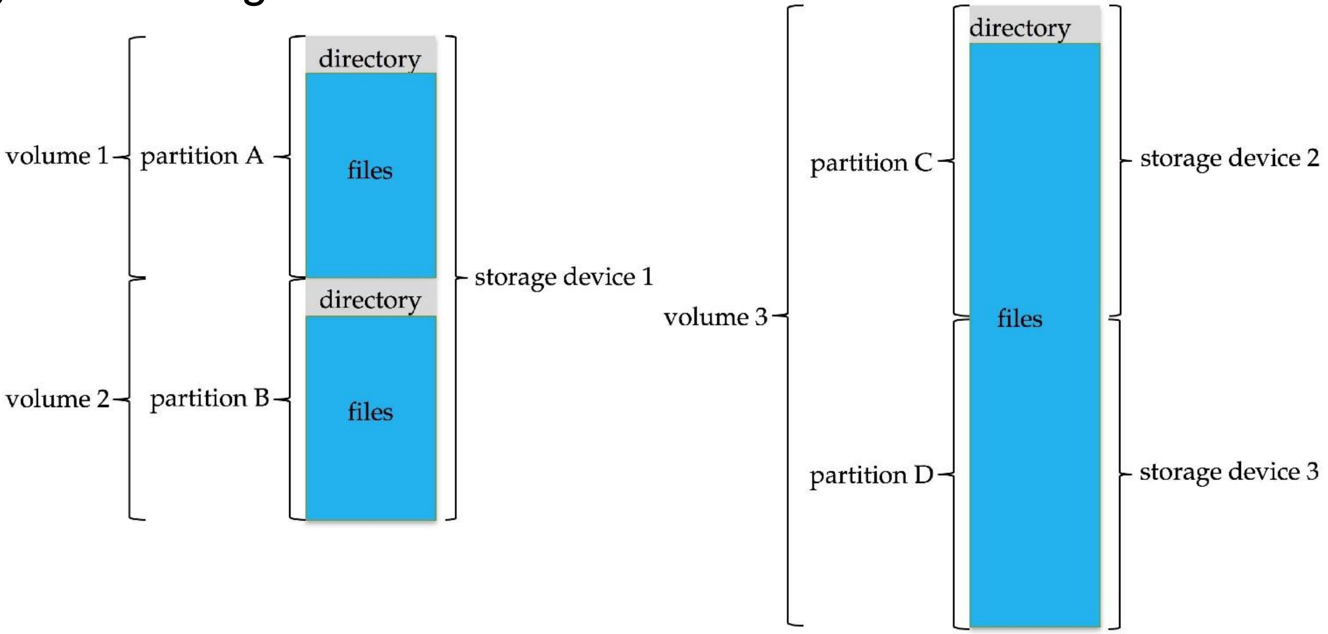
Disk Structure

- Disk can be subdivided into **partitions**
- Disk or partition can be used **raw** – without a file system, or **formatted** with a file system
- Partitions also known as minidisks, slices
- Entity containing file system known as a **volume**
- Each volume containing file system also tracks that file system's info in **device directory** or **volume table of contents**
- As well as **general-purpose file systems** there are many **special-purpose file systems**, frequently all within the same operating system or computer

Storage Device Organization

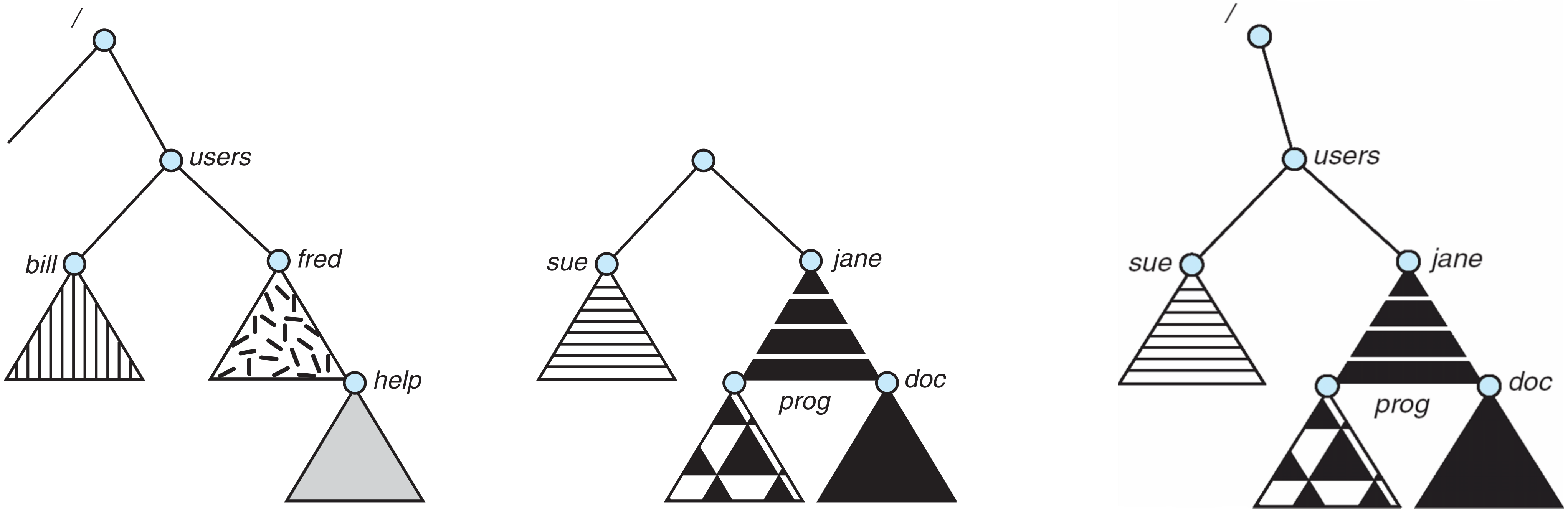
- General-purpose computers can have multiple storage devices
  - Devices can be sliced into partitions, which hold volumes
  - Volumes can span multiple partitions
  - Each volume usually formatted into a file system
  - # of file systems varies, typically dozens available to choose from

Typical storage device organization:



File System Mounting

- A file system must be **mounted** before it can be accessed
- A unmounted file system is mounted at a **mount point**



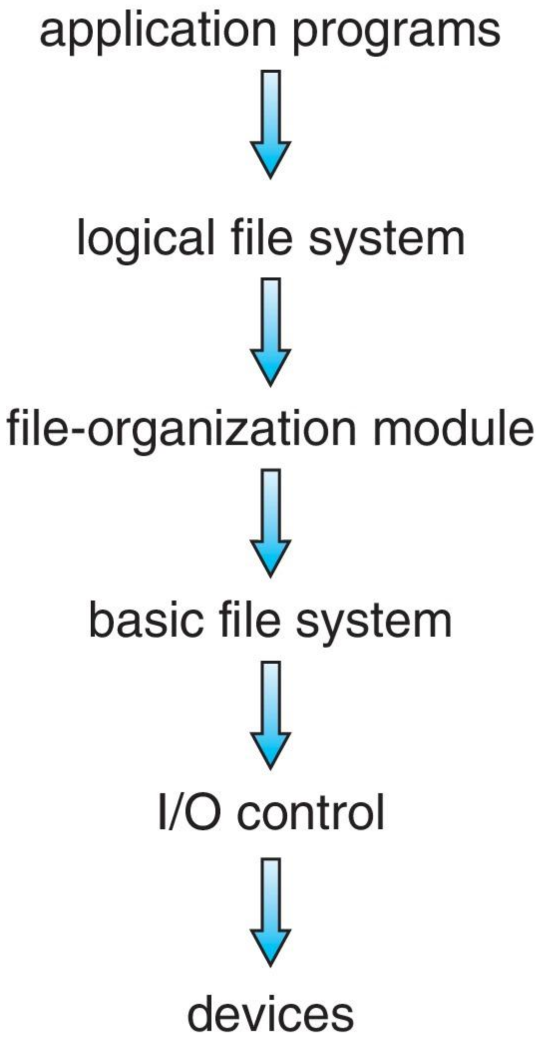
Existing system   Unmounted volume   Volume mounted at `/users`

File System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- **File system** resides on secondary storage (disks)
  - Provides user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes or 4096 bytes)
- **File control block** (**FCB**): storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers

File System Layers

- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance, translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Logical layers can be implemented by any coding method according to OS designer
- Many file systems, sometimes many within an operating system
  - Unix: UFS(Unix File System), FFS
  - Linux: **extended file system** ext3 and ext4
  - ZFS, GoogleFS, FUSE, etc

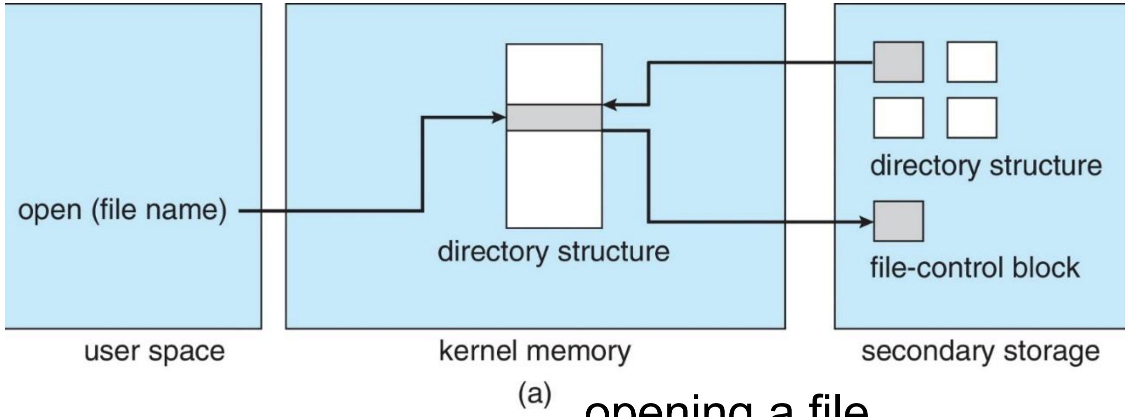


File System Operations

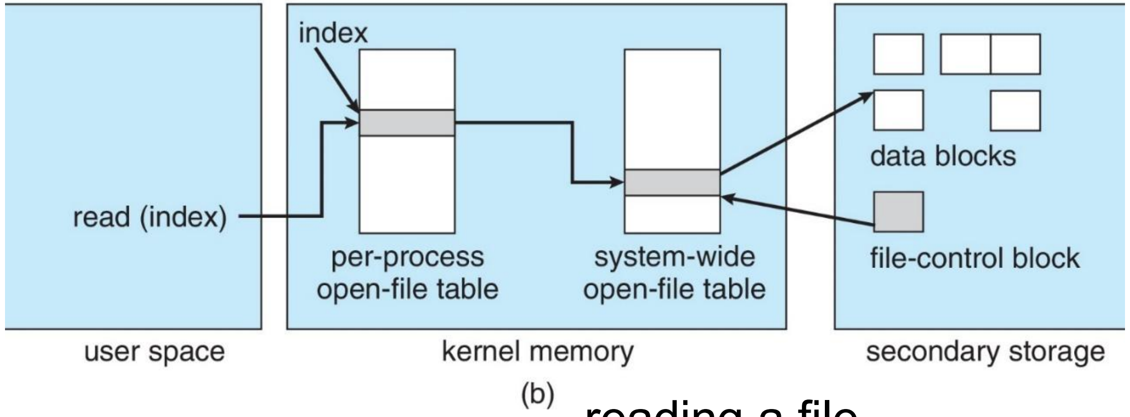
- **Boot control block** contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- **Volume control block** (**superblock, master file table**) contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- **Directory structure** (per file system) is used to organize the files
  - File names and associated inode numbers (master file table)

In-Memory File System Structures

- **Mount table** storing file system mounts, mount points, file system types
- **system-wide open-file table** contains a copy of the FCB of each file and other info
- **per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info



opening a file



reading a file

Agenda

- Files and Directories
- File System Structure & Operations
- File System Implementation

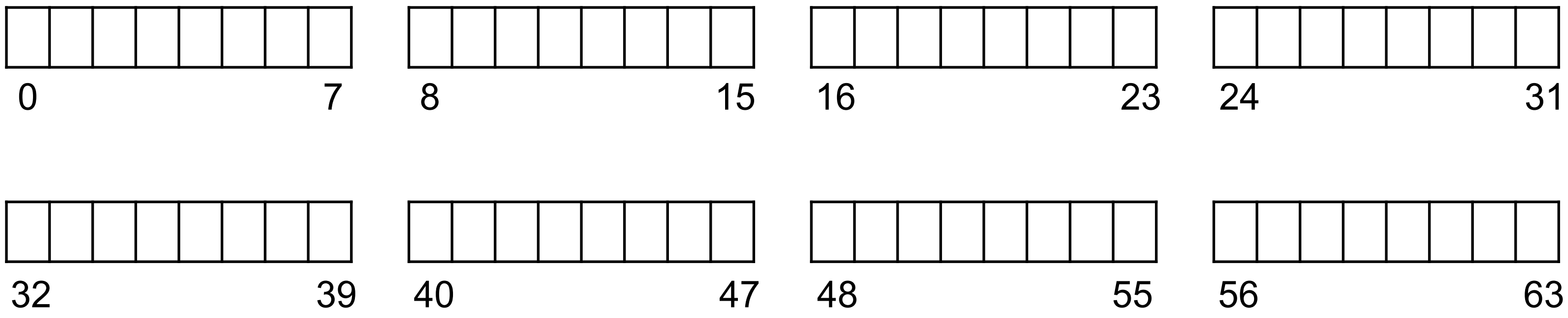
File System Implementation

- What types of data structures are utilized by the file system?
- How file system organize its data and metadata?
- Understand access methods of a file system.
  - `open()`, `read()`, `write()`, etc.

Overall Organization

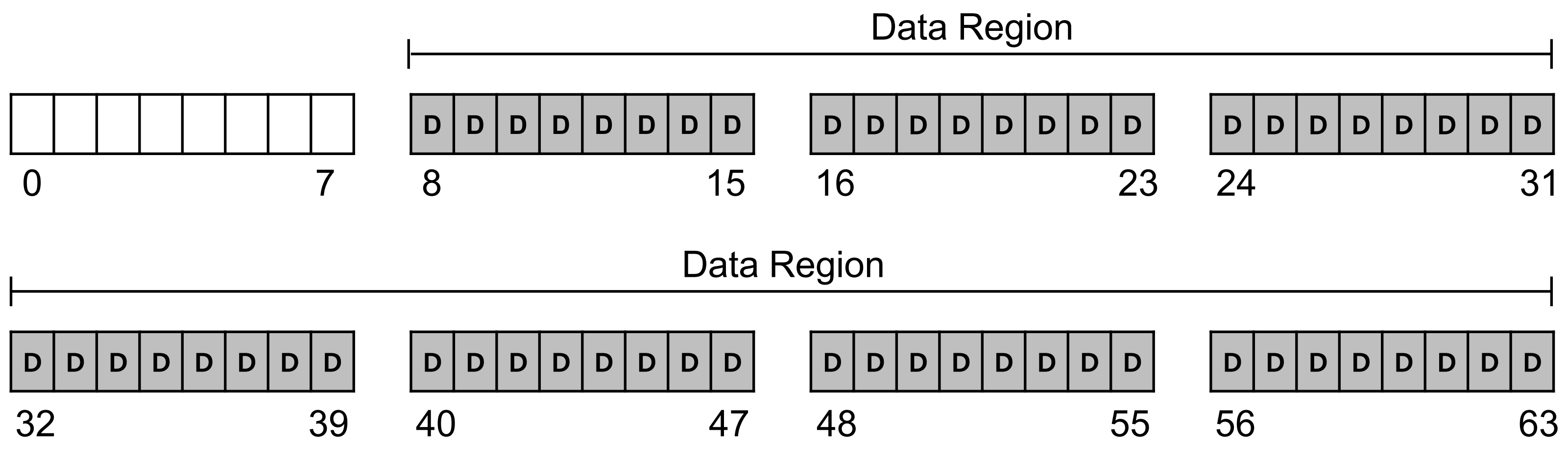
- Let’s develop the overall organization of the file system data structure.

- Divide the disk into blocks.
  - Block size is 4KB
  - The blocks are addressed from 0 to N-1.



Data Region in File System

- Reserve data region to store user data

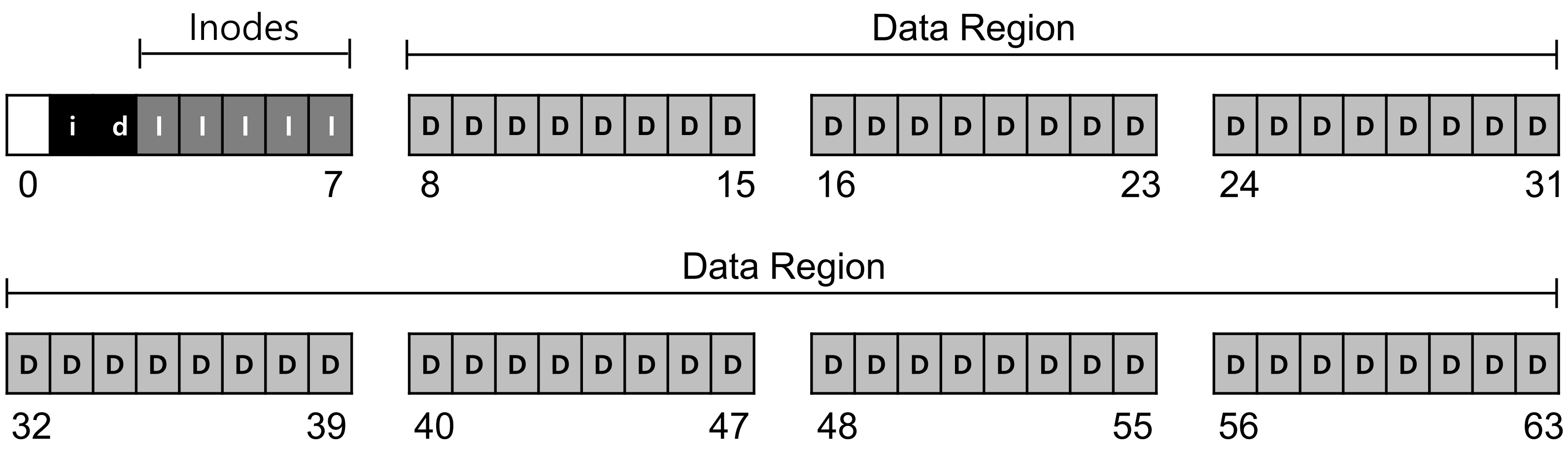


- File system has to track which data block comprise a file, the size of the file, its owner, etc.

How we store these inodes in file system?

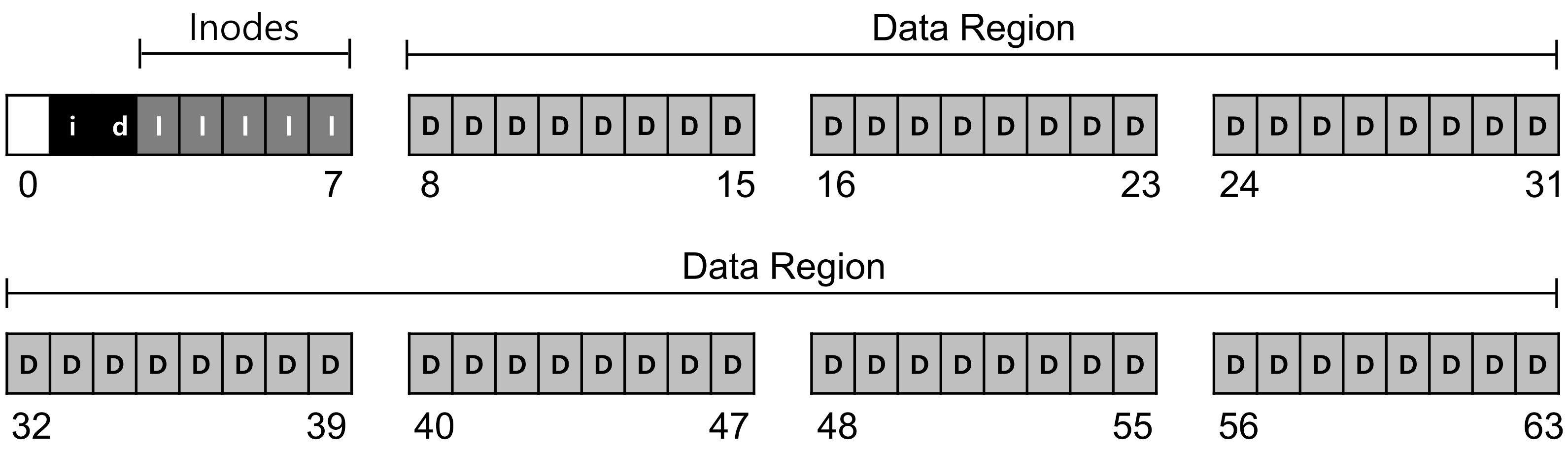
Inode Table in File System

- Reserve some space for inode table
  - This holds an array of on-disk inodes.
  - Ex) inode tables: 3 ~ 7, inode size: 256 bytes
    - 4-KB block can hold 16 inodes.
    - The file system contains 80 inodes. (maximum number of files)



Allocation Structures

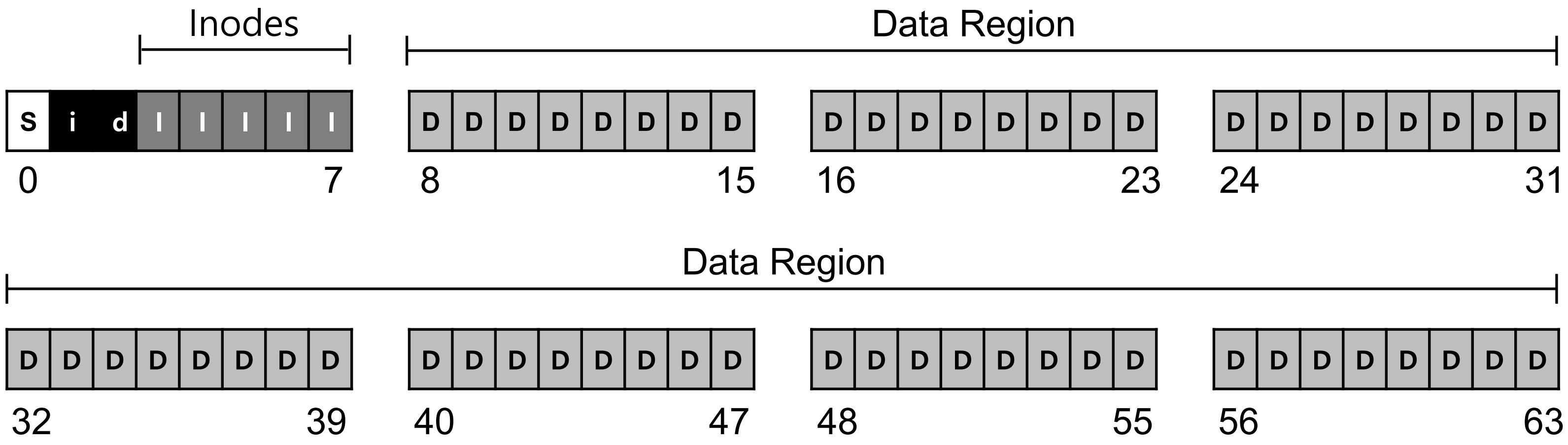
- This is to track whether inodes or data blocks are free or allocated.
- Use bitmap, each bit indicates free(0) or in-use(1)
  - inode bitmap: for inode table
  - data bitmap: for data region for data region



HANDONG GLOBAL UNIVERSITY

Super Block

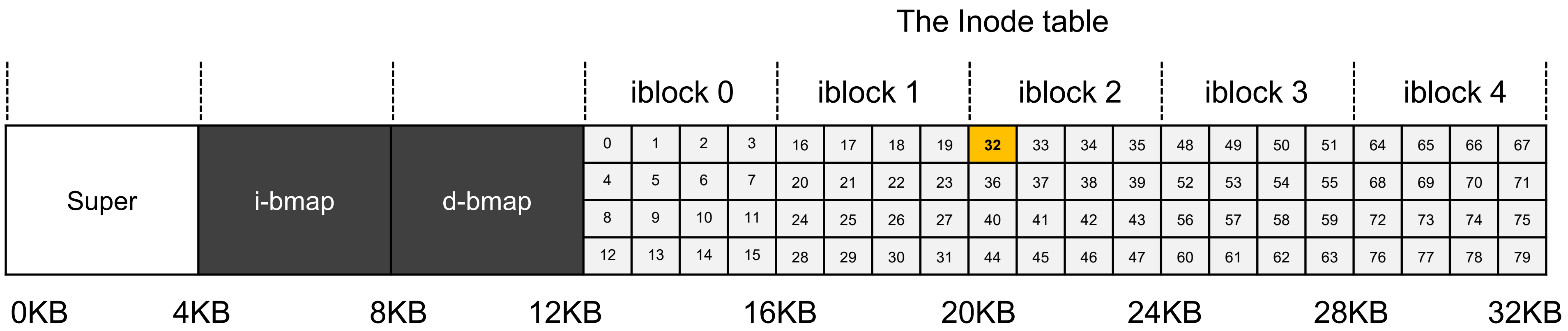
- Super block contains this information for particular file system
  - Ex) The number of inodes, begin location of inode table, etc



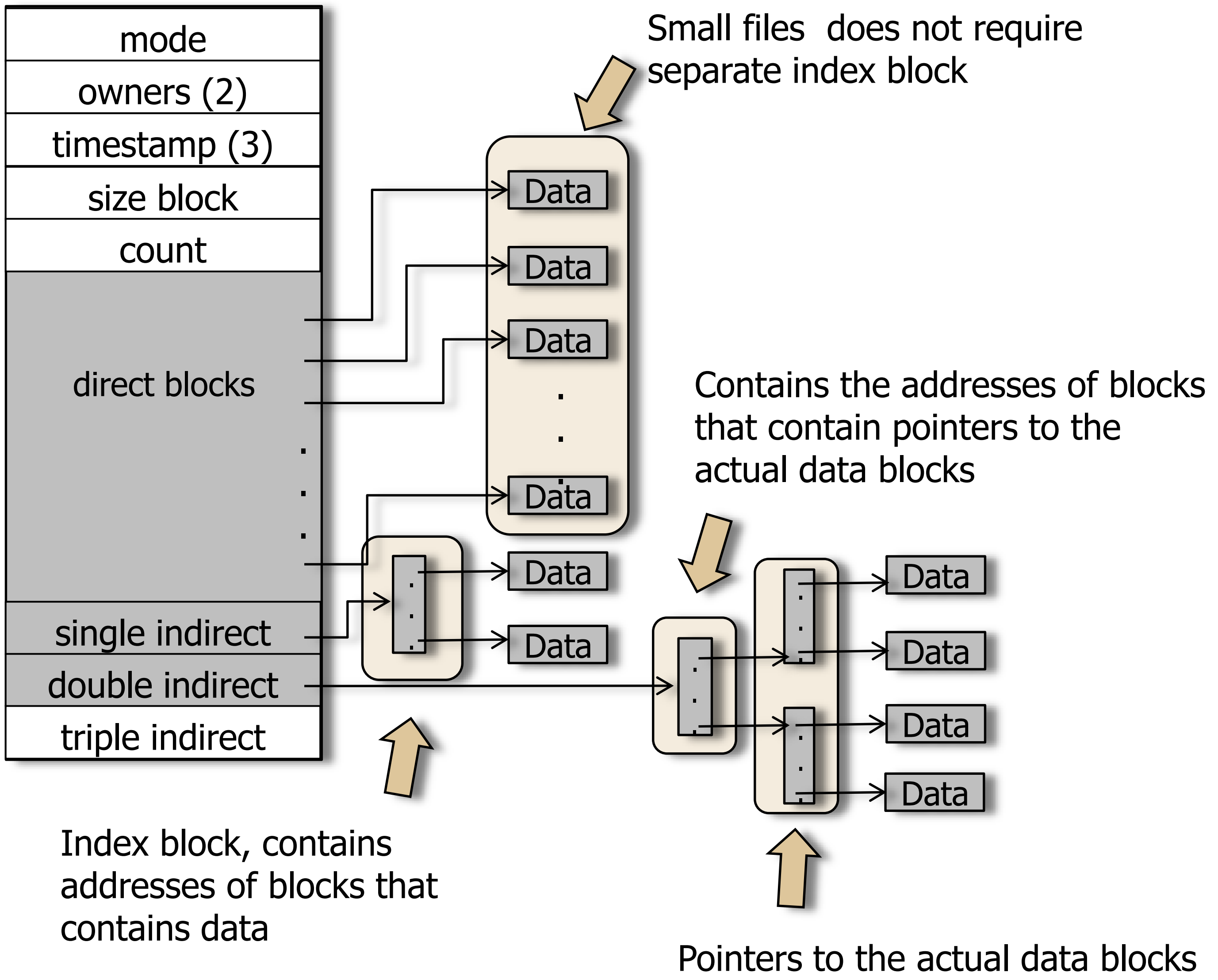
- Thus, when mounting a file system, OS will read the superblock first, to initialize various information.

File Organization: The inode

- Each inode is referred to by inode number.
  - by inode number, File system calculate where the inode is on the disk.
  - Ex) inode number: 32
    - Calculate the offset into the inode region (32 x sizeof(inode) (256 bytes) = 8192 (=8KB)
    - Add start address of the inode table(12 KB) + inode region(8 KB) = 20 KB



File Structure: Indexed Allocation

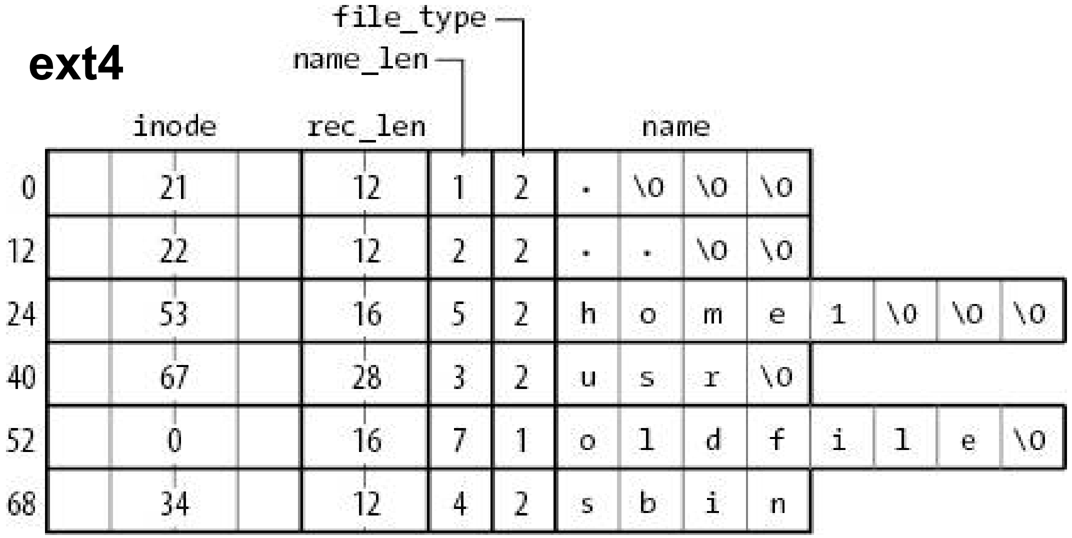


Directory Structure

- VSFS

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar

ext4



Access Path: Reading a File from Disk

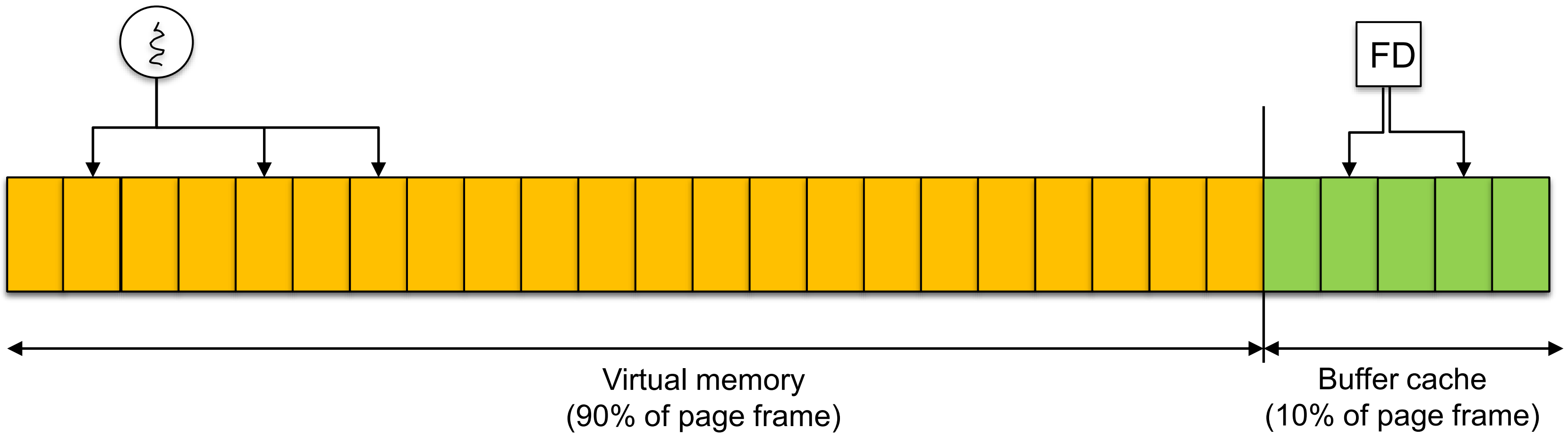
	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			**read**							
						read				
				read						
							read			
					read					
read()					**read**					
								read		
					write					
read()					**read**					
									read	
					write					
read()					**read**					
										read
					write					

Access Path: Writing a File to Disk

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)			read							
						read				
				read						
							read			
		read								
		write								
							write			
					read					
					write					
				write						
write()					read					
	read									
	write									
								write		
					write					
write()					read					
	read									
	write									
									write	
					write					
write()					read					
	read									
	write									
										write
					write					

Caching and Buffering

- Reading and writing can very IO intensive.
  - File open: two IO for each directory component and one read for the data.
- Buffer cache
  - cache the disk blocks to reduce the IO.
  - LRU replacement
  - Static partitioning: 10% of DRAM, inefficient usage



Caching and Buffering

- Page cache
  - Merge virtual memory and buffer cache
  - A physical page frame can host either a page in the process address space or a file block.
    - Process uses page table to map a virtual page to a page frame.
    - A file IO uses “address_space”(Linux) to map a file block to a physical page frame.
  - Dynamic partitioning