Lecture Notes for Lecture 9 of CS 5600 (Computer Systems) for the Fall, 2019 session at the Northeastern University Silicon Valley Campus.

File and Directory Systems

Philip Gust,
Clinical Instructor
Department of Computer Science

Acknowledgements:
These slide highlight content from suggested readings.

Lecture 8 Review

- In this lecture, we studied one of the primary resources managed by an operating system. A *process* is an instance of a program that has been loaded into memory and is being executed by a computer.
- We learned that a process contains the code and data that are defined by the program, as well as the running state of the program, and information about other resources being used by the program.
- We saw how a process provides a virtual computer that enables a program to operate as though it has the sole use of the computer and its resources.
- The also showed how the operating system and its kernel manage processes and mediate access to resources shared by all of them.
- The lecture concluded by looking at how processes can be viewed from the shell, and how a C program can start and run processes and send signal to its own or another process.

File and Directory Systems

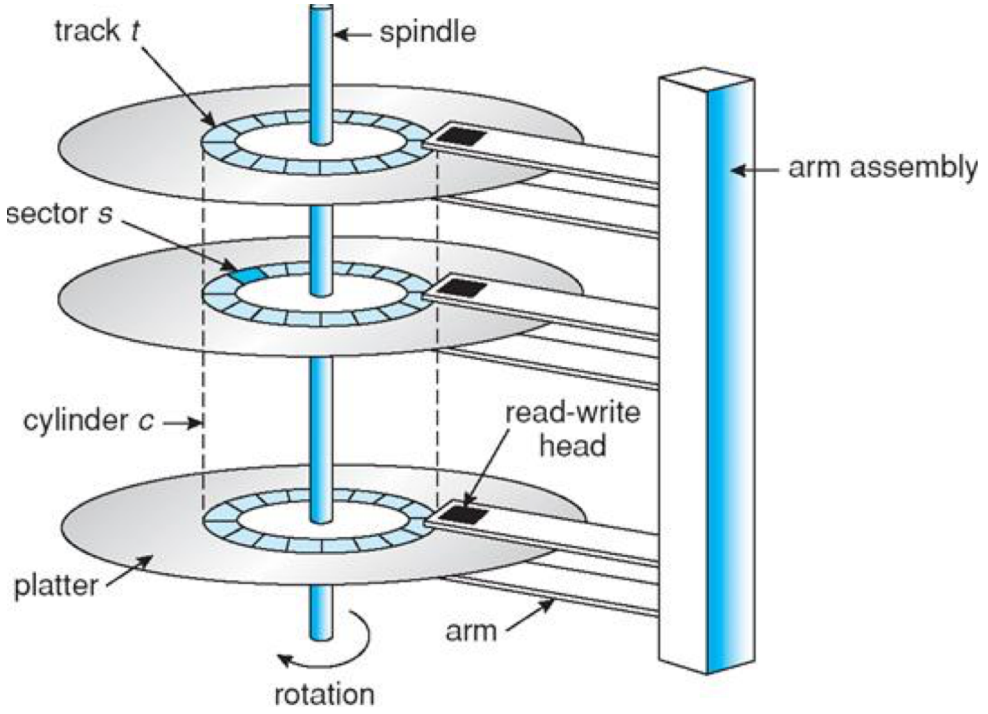
Overview

- In this lecture, we will learned about disks, which are used for long-term storage of information in a computer system, and how information is physically organized on a disk.
- We will see how the file system stores and accesses information on a disk as blocks of storage, and how it organizes information into files in a file system.
- We will also learn about a special kind of file called a directory, which names files and organizes them into a hierarchically named directory system.
- Finally, we will see how to mange files and directies in C through calls to operating system functions that interact with the file and directory system.

File and Directory Systems

The File System

- Disk is logically made up of *platters*, with concentric rings of *tracks*, and radial *sectors*. A sector of a track on a platter is called a *block*.



File and Directory Systems

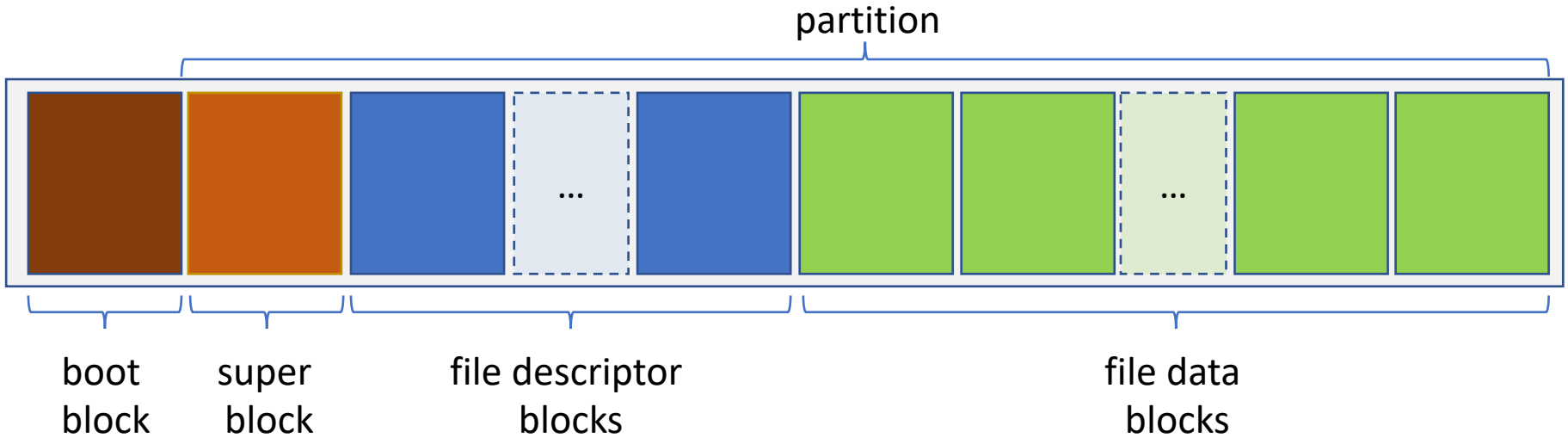
The File System

- A disk is organized as a logical collection of fixed-size blocks, typically between 512 and 4096 bytes.
- Each file is stored in one or more blocks.
- For simplicity, blocks are not split between files; leftover space in a block is unused (internal fragmentation).
- When creating or enlarging a file, the data is broken into block-size chunks and stored in new disk blocks allocated to the file from a pool of free blocks on disk.
- Reading a file reassembles the information from the blocks in the proper order.
- Access to the file system is managed by the kernel through file system *drivers* that provide standard operations on file system content.

File and Directory Systems

The File System

- Blocks are organized into *partitions* that represent file systems on disk. A computer can have one or more partitions that can be on the same or different disk devices.
- We will learn about the role of each kind of block and how there are organized next. Here is a typical disk layout with a single partition.



File and Directory Systems

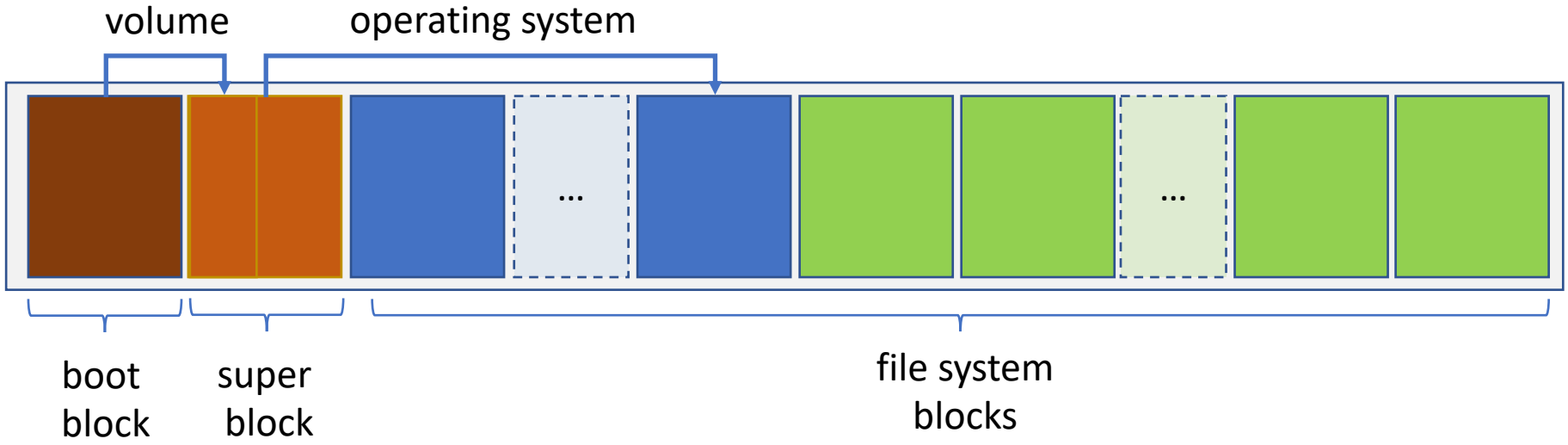
The File System Boot Block

- A disk device is divided into groups of blocks.
- **Boot Block**: Contains a *master boot record* with code that is automatically loaded into memory and run to boot the machine if this is a bootable device. Otherwise block is allocated but not used.
- Since the standard block size is 512 bytes, the entire boot loader has to fit into this space. The contents of the MBR are:
  - First stage boot loader (≤ 440 bytes)
  - Disk signature (4 bytes)
  - Disk partition table, which identifies distinct regions of the disk (16 bytes per partition × 4 partitions)
- The MBR code scans through its partition table for the device and loads the *volume boot record* (VBR) from the super block for that partition.

File and Directory Systems

The File System Boot Block

- When the computer is started, it reads a fixed-size boot block on the primary or boot partition. It contains a small executable program that loads and executes a volume loader to load the operating system.
- The volume super block address stored in the volume table of the boot block is used by this program. The volume super-block loader mounts the file system and loads the operating system.



File and Directory Systems

The File System Volume Blocks

- Groups of blocks for each partition include:
- **Super Block**: Information about the file system parameters. The OS also reads this block while *mounting* the file system.
- **File Descriptor Blocks**: Fixed set of blocks that hold information about files and directories for this partition.
- **File Data Blocks**: Rest of the blocks used to hold actual file data in the file system for this partition. Blocks are allocated as required.

File and Directory Systems

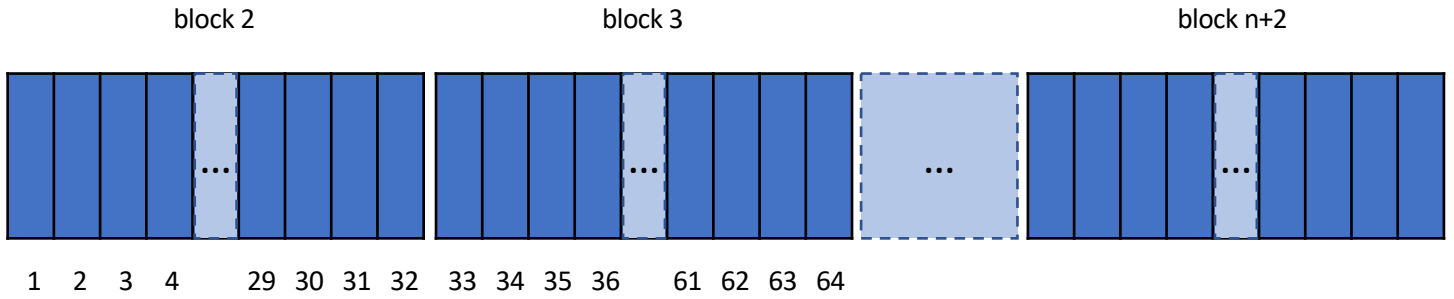
The File System Superblock

- The volume superblock parameters that provide sufficient information for the operating system to mount and manage the file system. Information stored in the volume super block, includes:
  - Type of the file system (eg. ext2, ext3, ..)
  - Block size
  - Size of file system
  - Size of file descriptor area
  - Free block list pointer or location of bitmap
  - Location of file descriptor of root directory
  - Other metadata such as permissions and times
- The VBR also has a volume boot loader, a secondary loader that mounts the file system and loads the operating system.
- The superblock is so important that most file systems keep backups.

File and Directory Systems

The File System File Descriptor Blocks (i-nodes)

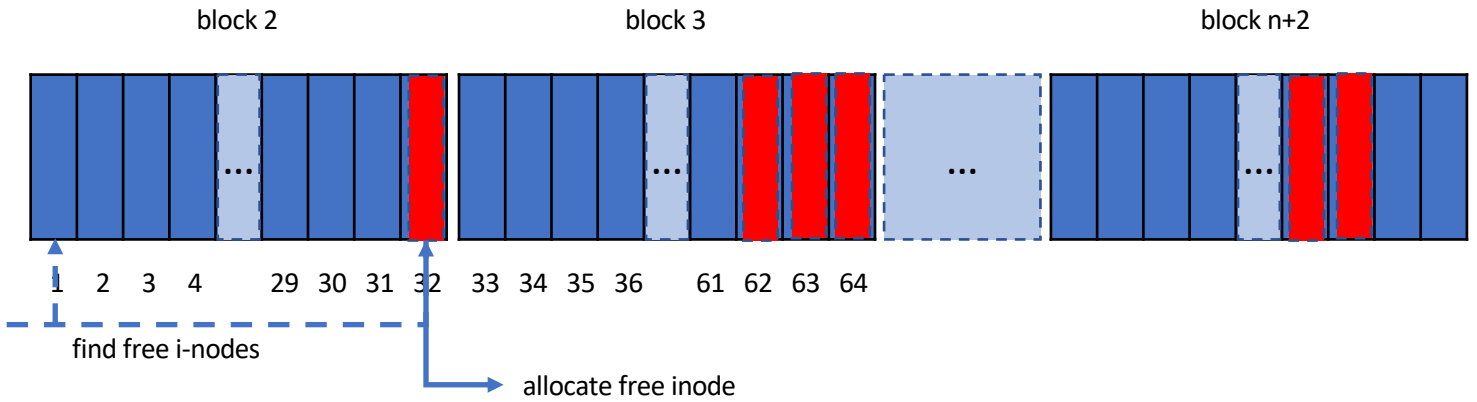
- The file descriptor blocks are a contiguous set of blocks that contain information about individual files and directories. On POSIX systems, these are called *i-nodes* (index nodes).
- Multiple i-nodes are packed into a block, and contiguous blocks are allocated when the partition is created. Each i-node is identified by an *i-number* (index number).
- For a typical i-node size of 128 bytes and 4096 bytes/block, there would be 32 i-nodes per block. For typically 1 i-node for every 16384 bytes (16kb) there would be ~61035156 i-nodes for a 1Tb partition.



File and Directory Systems

The File System File Descriptor Blocks (i-nodes)

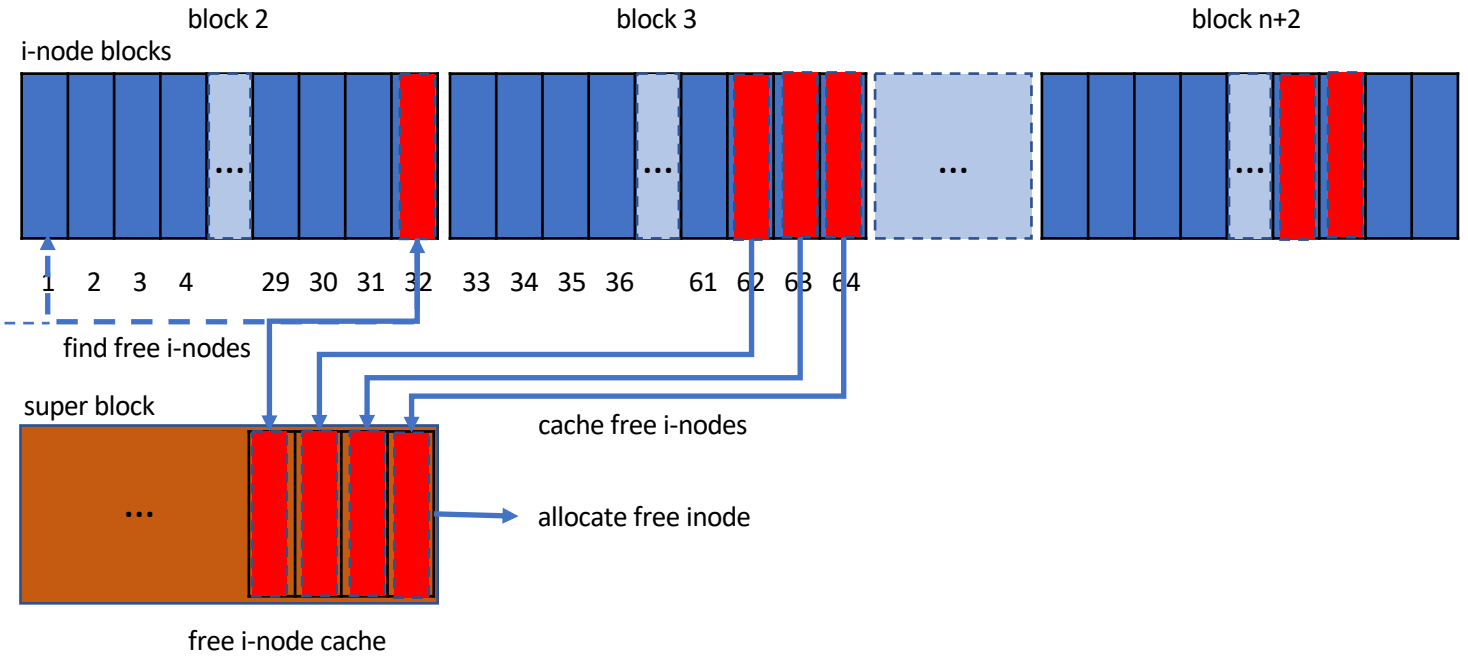
- Identifying a free i-node can be done in one of several ways:
  a) Traversing the i-node array to find an i-node marked free each time one is needed. This takes no more space, but requires searching i-node blocks when a file is created, and clearing the i-node fields to identify it as free.



File and Directory Systems

The File System File Descriptor Blocks (i-nodes)

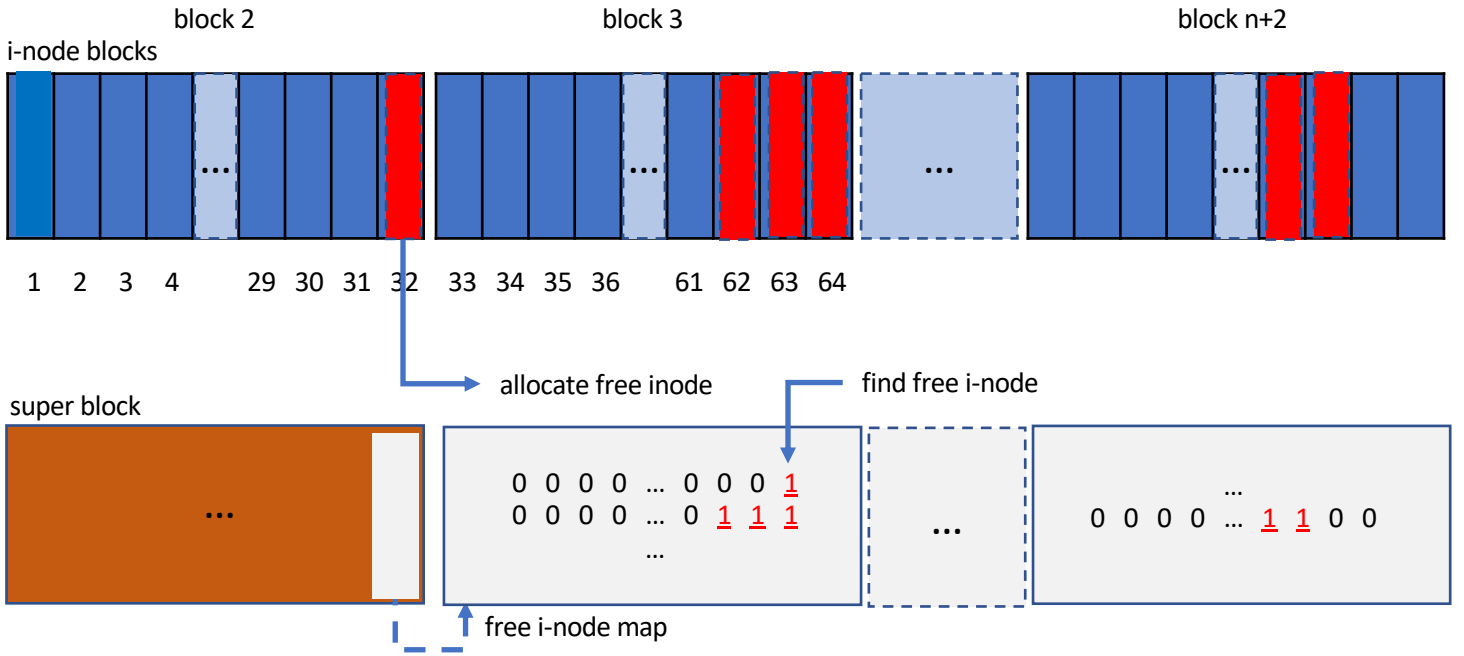
- Identifying a free i-node can be done in one of several ways:
  b) Caching a short list of free i-nodes in the super block, and rescanning when a new i-node is needed and the cache is empty. This takes advantage of the fact that there may be several free i-nodes in the same or adjacent blocks.



File and Directory Systems

The File System File Descriptor Blocks (i-nodes)

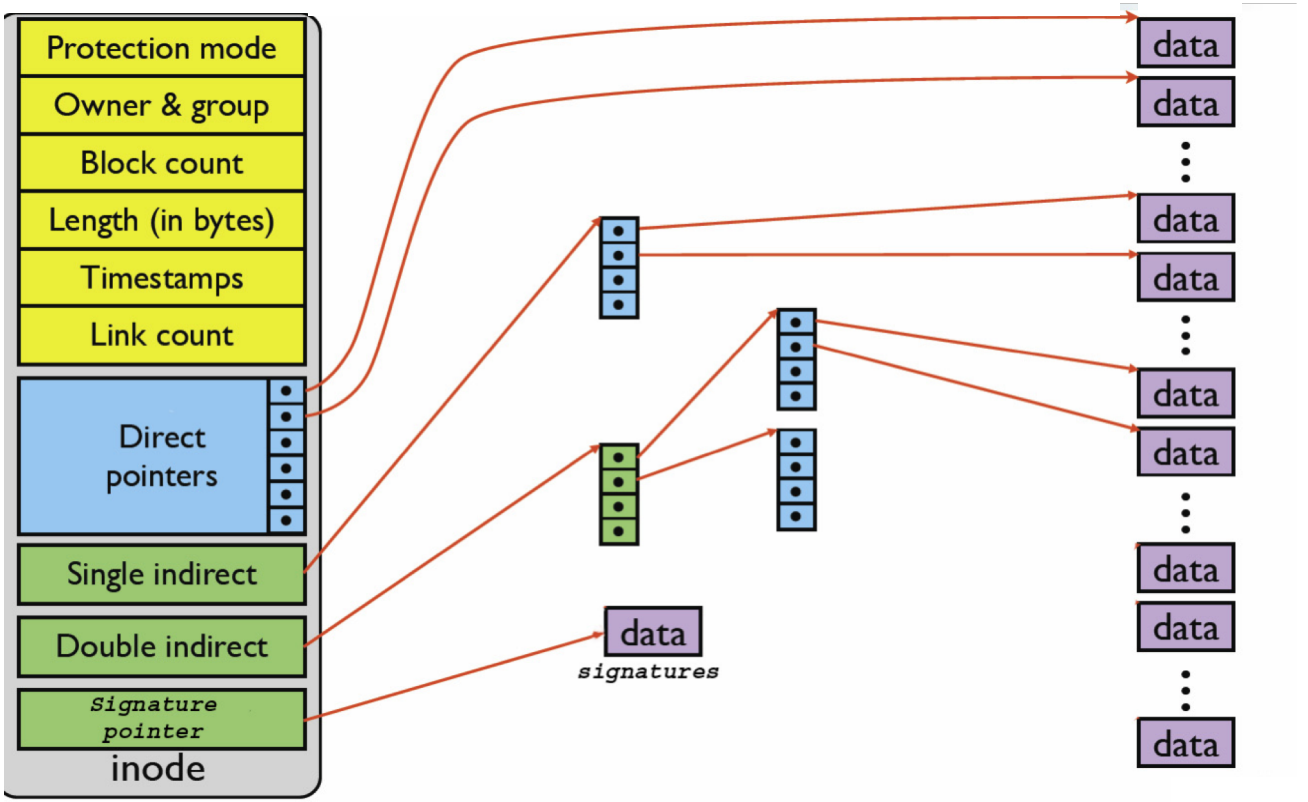
- Identifying a free i-node can be done in one of several ways:
  - c) Maintaining a bitmap of free i-nodes in separate contiguous blocks. If the bit for an i-node is 0, the i-node is free or 1 for in-use. For a 1Tb file partition with 4096 bytes per block and 61035156 i-nodes, that would consume ~1863 blocks. The exact map block and bit can be computed in constant time.



File and Directory Systems

The File System File Descriptor Blocks (i-nodes)

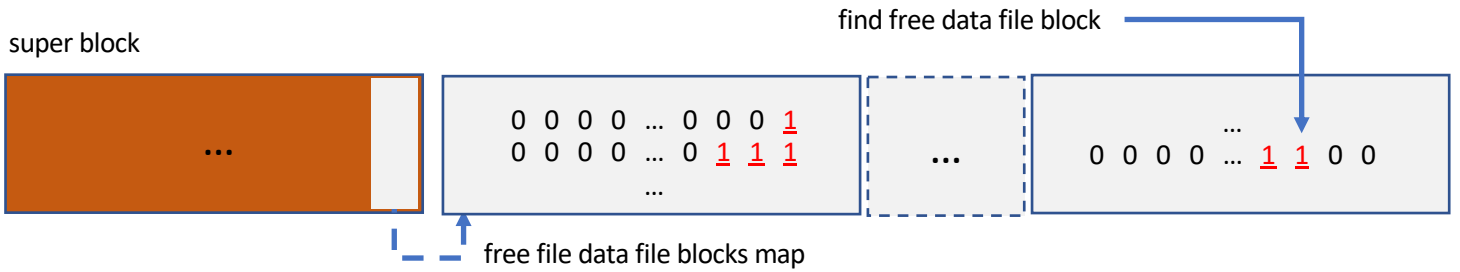
- An i-node structure represents information about a file or directory on disk. Double- or even triple-indirection supports larger files.



File and Directory Systems

The File System File Data Blocks

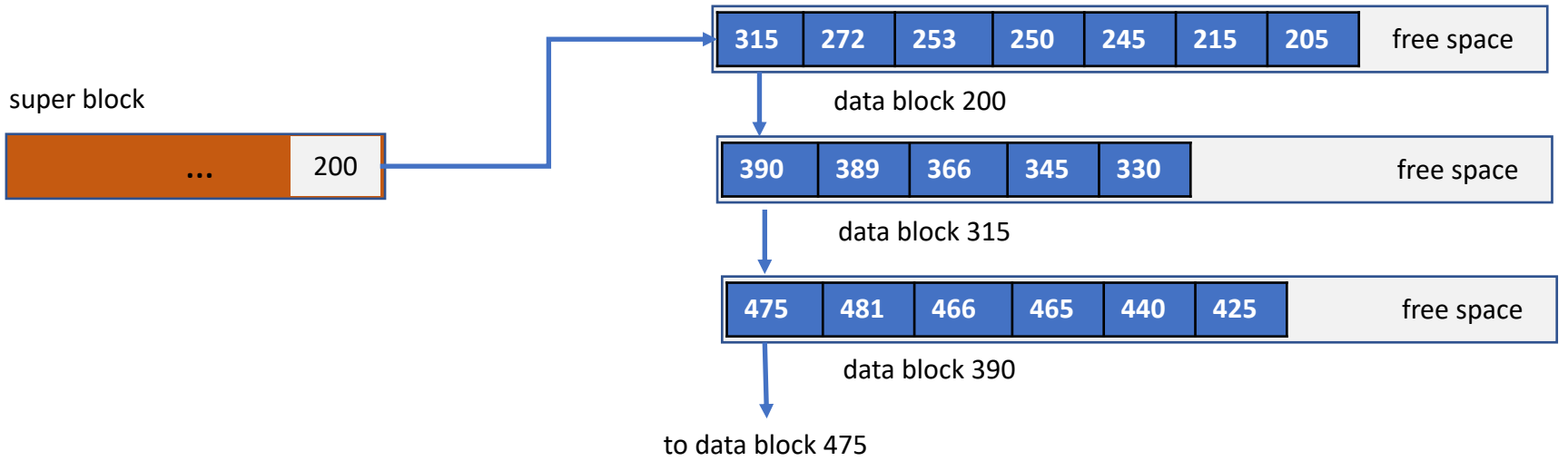
- There are several strategies for managing the free blocks.
  a) Maintain a bitmap of free data file blocks in separate contiguous blocks. If the bit for a block is 0, the block is free or 1 for in-use.
     - For a 1Tb file partition with 4096 bytes per block and 61035156 i-nodes, that would consume ~33554432 blocks.
     - The exact map block and bit can be computed in constant time. However, the technique is wasteful of space, especially for larger disks.



File and Directory Systems

The File System File Data Blocks

- There are several strategies for managing the free blocks.
  b) Use free blocks as a linked list pointed to by the super block. The first free block contains a list of other free block numbers. The last block number points to another block of free block numbers.



File and Directory Systems

The File System File Ownership and Permissions

- The i-node protection mode field includes information about ownership and type for a file.
- A file is owned by a user (userID) and also associated with a user group that includes other other users (groupID).
- Files have three access permissions: for *user* (u), for the *group* (g), and for *others* (o).
- Permissions have three bits for *read* (r), *write* (w), and *execute* (x), shown either as three octal digits or symbolically
  - 755: rwx r-x r-x (user: rxw, group: r-x, other: r-x)
  - 644: rw- r-- r-- (user: rw-, group: r--, other: r--)
- For files, *execute* indicates a runnable program; for directories it indicates ability to list files in the directory.

File and Directory Systems

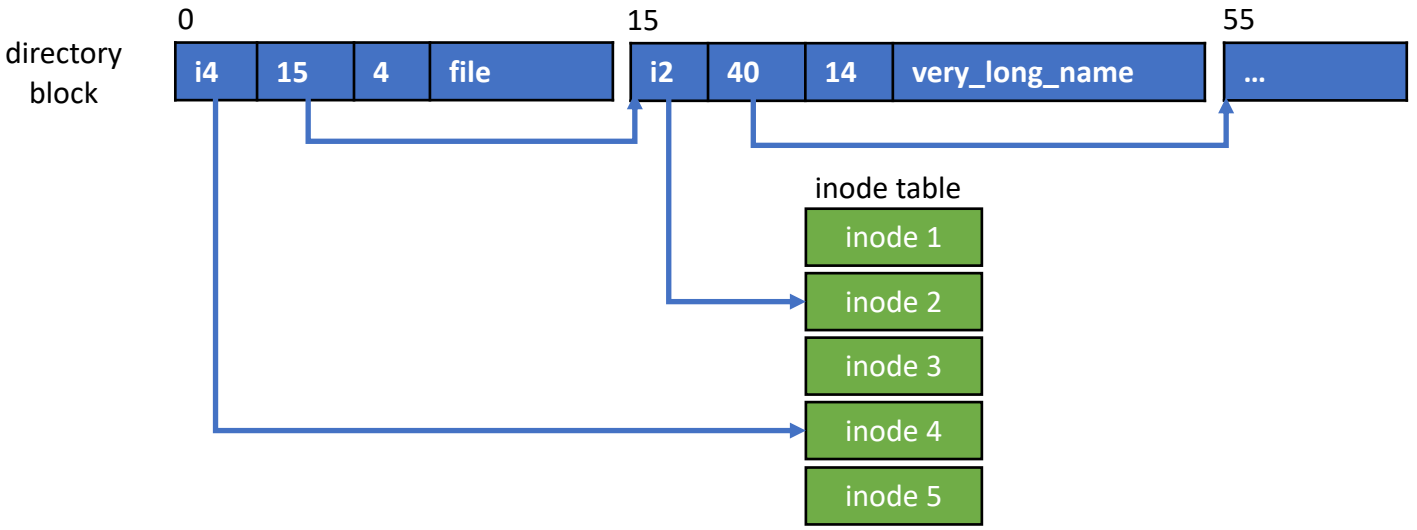
The File System File Protection Mode

- The file protection mode field also includes information about the type of file. This information enables the operating system to determine what operations can be performed on it.
- Two of the file types are:
  - *Regular file*: uninterpreted bytes that can contain any type of data.
  - *Directory*: a special file with a list of other files and directories that are contained in the directory.
- We will learn about other file types later on. For now, we will focus on the directory file and how it is represented.

File and Directory Systems

The Directory System

- A directory is a structured file that maps file names to i-nodes. A directory can contain files and other directories, structured as a tree.
- A directory file has at least two items:
  - The i-node of the file
  - The name of the file
- This simple directory structure requires linear searching.

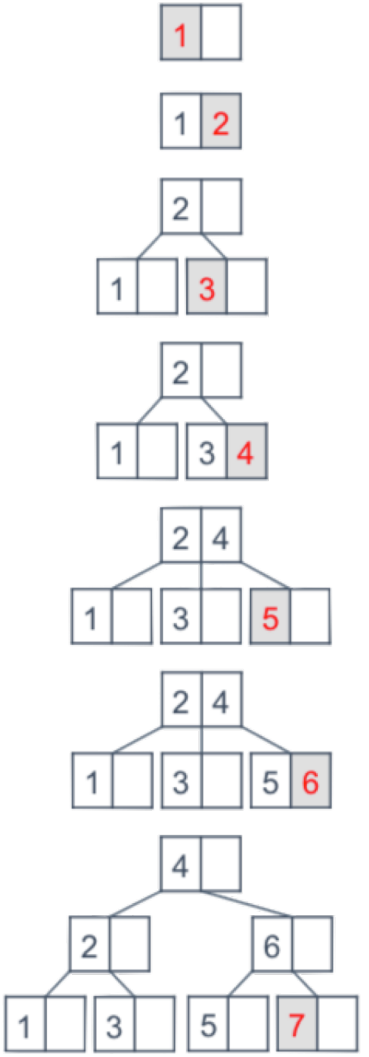


File and Directory Systems

The Directory System

- Another disk-optimized search structure for directories, the *b-tree*, is made up of large blocks, with a high branching factor, to reduce the number of block accesses needed for an operation.
- Interior and leaf nodes are identical; each contains a sorted list of key/value pairs, and (in non-leaf nodes) pointers between pairs of keys, pointing to subtrees holding keys between those two values:

1. The first value ’1’ goes in the root.
2. Since the root is not full, ‘2’ value goes here too.
3. Adding ‘3’ root is full, so we split the node. Since the block does not have a parent (it is the root) we add one, which becomes the new root
4. ’4’ fits into one of the leaf nodes where there is room
5. ’5’ does not fit, so we split the node. There is room in the parent to hold another pointer
6. ’6’ fits in the leaf node
7. ‘7’ doesn’t, so we split the leafnode, but that causes the parent node to overflow, so we split it, and have to add a new parent node which becomes the new root.



File and Directory Systems

File and Directories

- Next we will look at how to interact i-nodes and directory entries in C
- The *stat()* system call takes a *stat* structure by reference and returns information about the file form the i-node. Here is how to get the last accessed time stored in the i-node.

```
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>
int main(void) {
    static const char* bash_str = "/bin/bash";
    struct stat s;
    if (stat(bash_str, &s) != 0) {
        perror(bash_str);
    } else {
        printf(”%s: last accessed %s\n", bash_str, ctime(&s.st_atime));
    }
}
```

File and Directory Systems

File and Directories

- A variant of the function, *fstat()*, returns the same information using the file id returned by the open() system call on the file.

```
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main(void) {
    static const char* bash_str = "/bin/bash";
    int fd = open(bash_str, O_RDONLY);  // read only
    if (fd == -1) {
        perror(bash_str);
    } else {
        struct stat s;
        if (fstat(fd, &s) != 0) {  // read information about file into s
            perror(bash_str);
        } else {
            printf("%s: last accessed %s\n", bash_str, ctime(&s.st_atime));
    }
    close(fd);  // frees entry in file table for reuse
}
```

File and Directory Systems

File and Directories

- Here is the definition of the *stat* file structure. This information comes primarily from the inode fields.

```
struct stat {
        dev_t     st_dev;       /* ID of device containing file */
        ino_t     st_ino;       /* inode number */
        mode_t    st_mode;      /* protection */
        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;   /* blocksize for file system I/O */
        blkcnt_t  st_blocks;    /* number of 512B blocks allocated */
        time_t    st_atime;     /* time of last access */
        time_t    st_mtime;     /* time of last modification */
        time_t    st_ctime;     /* time of last status change */
};
```

File and Directory Systems

File and Directories

- The stat.st_mode field can be used to determine whether the file is a directory. The *S_ISDIR()* and *S_ISREG()* macros check the bit fields:

```
#include <sys/stat.h>
#include <stdio.h>
int main(void) {
    const char* bin_str = "/usr/bin";
    struct stat s;
    if (stat(bin_str, &s) != 0) {
        perror(bin_str);
    } else {
        if (S_ISDIR(s.st_mode)) printf("%s is a directory", bin_str);
        if (S_ISREG(s.st_mode)) printf("%s is a regular file", bin_str);
    }
}
```

File and Directory Systems

File and Directories

- We can also use masks for these fields to determine whether an entry is a directory (using stat.st_mode field and S_IFMT mask).

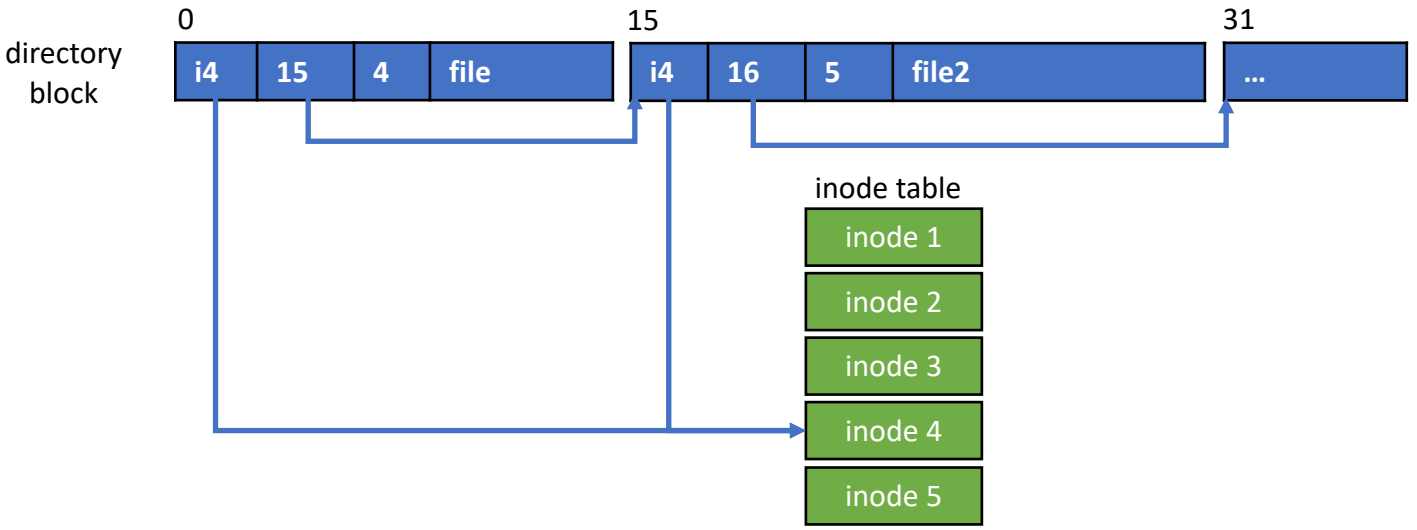
```
#include <sys/stat.h>
#include <stdio.h>

int main(void) {
    const char* bin_str = "/usr/bin";
    struct stat s;
    if (stat(name, &s) != 0) {
        perror(bin_str);
    } else {
        switch(s.st_mode & S_IFMT) {
        case S_IFDIR:
            printf("%s is a directory\n", bin_str);
            break;
        case S_IFREG:
            printf("%s is a regular file\n", bin_str);
            break;
        }
    }
}
```

File and Directory Systems

File and Directories

- A link (or “hard link) is the term for a directory entry for a file or directory.
- The directory entry associates an i-node with a name in a directory.
- A file can have multiple links in multiple directories. If there are multiple links to directories, they may no longer be trees.



File and Directory Systems

File and Directories

- The i-node includes a link count. Adding a link increments the count.
  - Use the **ln** command to add a link for a file in a directory:

    ln *existing_name.txt new_name.txt*
  - Use the C link() function to add a link for a file in a directory:

    link("*existing_name.txt*", "*new_name.txt*");
  - Opening a file also increments the link count:

    vi *existing_name.txt*

File and Directory Systems

File and Directories

- This example shows creating a link to a file and observing their link counts using the ‘-i’ option of the ‘ls’ utility. Here is the listing for the file ‘1.txt’ showing the inode (1782366) and the link count (1):

```
$ ls -il 1.txt
1782366 -rw-r--r--  1 phil  staff  618 Jan 20  2012 1.txt
```

- Now we will create a second link to the file in the same directory:

```
$ ln 1.txt 2.txt
$ ls -il 1.txt 2.txt
1782366 -rw-r--r--  2 phil  staff  618 Jan 20  2012 1.txt
1782366 -rw-r--r--  2 phil  staff  618 Jan 20  2012 2.txt
```

- Both ‘1.txt’ and ‘2.txt’ have the same inode, and the inode link count is now 2.

File and Directory Systems

File and Directories

- Deleting a link decrements the count
  - Use **rm** command to remove a link for a file in a directory:

    rm new_name.txt;
  - Use the C *remove()* function to remove a link for a file in a directory:

    remove(“new_name.txt”);
  - Closing a file using the close() system call also decrements the file link count:

    close(fd);
  - File system reclaims storage when link count reaches 0.

File and Directory Systems

File and Directories

- This example shows deleting a link to a file and observing the link count. Here is the listing for the files ‘1.txt’ and ‘2.txt’ that have the same inode (1782366):

```
$ ln 1.txt 2.txt
$ ls -il 1.txt 2.txt
1782366 -rw-r--r--  2 phil  staff  618 Jan 20  2012 1.txt
1782366 -rw-r--r--  2 phil  staff  618 Jan 20  2012 2.txt
```

- Now we will delete ‘2.txt’:

```
$ rm 2.txt
$ ls -il 1.txt
1782366 -rw-r--r--  1 phil  staff  618 Jan 20  2012 1.txt
```

- The inode for the file ‘1.txt’ now has a link count of 1.

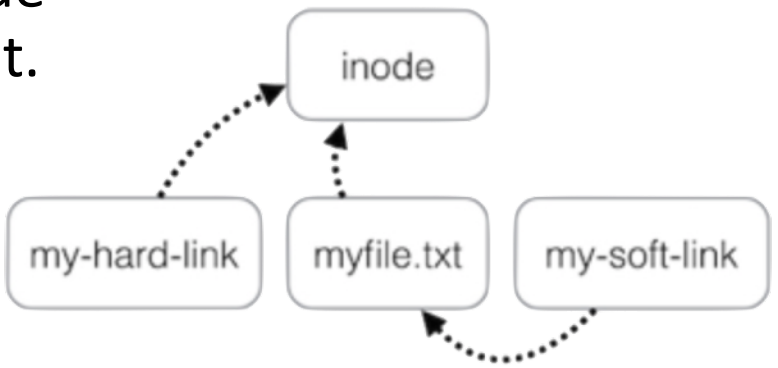
File and Directory Systems

File and Directories

- A *symbolic link* (or *soft link*) is a file that contains the name of another file.
  - Use the **ln** command to create a symbolic link:

    ln -s existing_name.txt symbolic_link.txt
  - Use *symlink*() command to create symbolic link in C:

    symlink("existing_name.txt", "symbolic_link.txt");
- A symbolic link does not involve an i-node and so does not increment the link count.
- If the file is deleted, the symbolic link refers to a file that no longer exists, and opening it results in a file not found error.



File and Directory Systems

File and Directories

- Add test for symbolic link using stat.st_mode field:

```
#include <sys/stat.h>
#include <stdio.h>
int main(void) {
    static const char* java_str = "/usr/bin/java";  // macos
    struct stat s, ls;
    if (stat(java_str, &s) != 0) {
        perror(java_str);
    } else {
        if (S_ISDIR(s.st_mode)) printf("%s is a directory\n", java_str);
        if (S_ISREG(s.st_mode)) printf("%s is a regular file\n", java_str);
        if (S_ISLNK(s.st_mode)) printf ("%s is a symbolic link\n", java_str);

        lstat(java_str, &ls); // returns info about link rather than linked file
        if (S_ISDIR(ls.st_mode)) printf("%s is a directory\n", java_str);
        if (S_ISREG(ls.st_mode)) printf("%s is a regular file\n", java_str);
        if (S_ISLNK(ls.st_mode)) {
            char link[128];
            readlink(java_str, link, 128);
            printf ("%s is a symbolic link -> %s\n", java_str, link);
        }
    }
}
```

File and Directory Systems

File and Directories

- This example shows creating a symbolic link to a file and observing their link counts. Here is the listing for the file ‘1.txt’ showing the inode (1782366) and the link count (1):

```
$ ls -il 1.txt
1782366 -rw-r--r--  1 phil  staff  618 Jan 20  2012 1.txt
```

- Now we will create a second link to the file in the same directory:

```
$ ln -s 1.txt 2.txt
$ ls -il 1.txt 2.txt
 1782366 -rw-r--r--   1 phil  staff  618 Jan 20  2012 1.txt
16456026 lrwxr-xr-x  1 phil  staff    5 Mar 18 07:09 2.txt -> 1.txt
```

- The files ‘1.txt’ and ‘2.txt’ have different inodes and their link counts are both 1.

File and Directory Systems

File and Directories

- This example shows deleting a symbolic link to a file.

  $rm 2.txt

  $ ls -il 1.txt

  1782366 -rw-r--r-- 1 phil staff 618 Jan 20 2012 1.txt

- If we had instead deleted ‘1.txt’, the symbolic link would remain an error would be reported when we try to use it.

File and Directory Systems

File and Directories

- We can list the contents of a directory in C using system calls that open the directory and iterate through its contents.
- The **opendir**(*pathname*) function opens the directory and returns a pointer to a **DIR** struct that represents the directory. This structure is defined in ‘dirent.h’.
- If the directory does not exist, NULL is returned.

File and Directory Systems

File and Directories

- Once we have a DIR struct, we can list the contents of a directory in C using system calls that iterates to get the next directory entry.
- The **readdir**(*dirp*) function returns a pointer to a **dirent** struct that represents the entry. This structure is defined in ‘dirent.h’.
- If there are no more entries in the directory, the function returns NULL.
- Once finished listing directories, resources should be freed by calling the **closedir**(*dirp*) function.

File and Directory Systems

File and Directories

- Here is the definition of a typical *dirent* file structure. This information comes primarily from the directory fields. Note that only *d_ino* and *d_name* are specfied by POSIX.

```
struct dirent {
    ino_t d_ino;                    // (POSIX) 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];              // (POSIX) filename
};
```

File and Directory Systems

File and Directories

- List the contents of a directory in C:

```
#include <stdio.h>
#include <dirent.h>
#include <stdlib.h>

int main(int argc, char * argv[]) {
    static const char* bin_str = "/bin";

    struct dirent *dp;
    DIR *dirp = opendir(bin_str);   // open directory for reading
    if (dirp == NULL) {
        perror(bin_str);
        return 1;
    }
    printf("%s:\n", bin_str);
    while ((dp = readdir(dirp)) != NULL) {  // get next entry
        printf("%9lu %s\n", (unsigned long)dp-> d_ino, dp-> d_name);
    }
    closedir(dirp);
    return 0;
}
```