Topics: Unix File and I/O
(USP Chapters 4 and 5)
(Optional SGG[9ed] Chapters 11, 12.1-5, 13)

CS 3733 Operating Systems

Instructor: Dr. Turgay Korkmaz

Department Computer Science

The University of Texas at San Antonio

Office: NPB 3.330
Phone: (210) 458-7346
Fax: (210) 458-4437
e-mail: korkmaz@cs.utsa.edu
web: www.cs.utsa.edu/~korkmaz

These slides are prepared based on the materials provided by Drs. Robbins, Zhu, and Liu. Thanks to all.

Outline

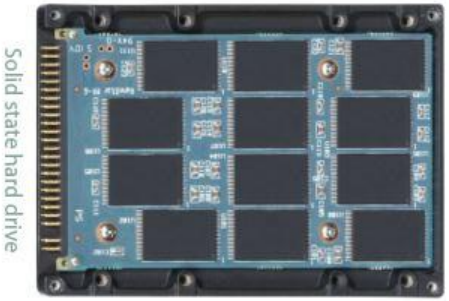
- Basics of File Systems
- Directory and Unix File System:
  - Inodes
  - Directory operations
  - Links of Files: Hard vs. Symbolic
- UNIX I/O System Calls: open, close, read, write, ioctl
- File Representations:
  - FDT, SFT, inode table
  - Fork and inheritance,
  - Filters and redirection
- FILE pointers (fopen) and buffering

Where to store information?

- How about storing information in the process address space?
- Why is this a good or bad idea?
  - Size is limited to size of virtual address space
    - ✓ May not be sufficient for airline reservations, banking, etc.
  - The data is lost when the application terminates
    - ✓ Even when computer doesn‘t crash!
  - Multiple process might want to access the same data
    - ✓ Imagine a telephone directory part of one process
- So, what can we do?

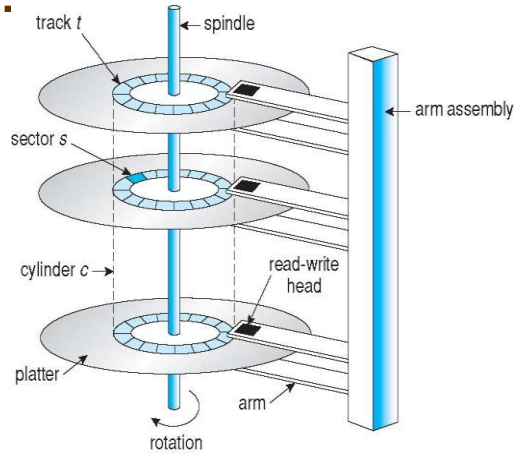
File Systems

- Criteria for long-term information storage:
  - Should be able to store very **large** amount of information
  - Information must **survive** the processes using it
  - Should provide **concurrent** access to multiple processes
- Solution:
  - Store information on **disks** in units called **files**
  - Files are persistent, and only owner can explicitly delete them
- File Systems: How the OS manages files!
  - Unix uses device **independence**.
  - I/O is done through **device drivers** that have a standard interface: open, close, read, write, ioctl



File Naming

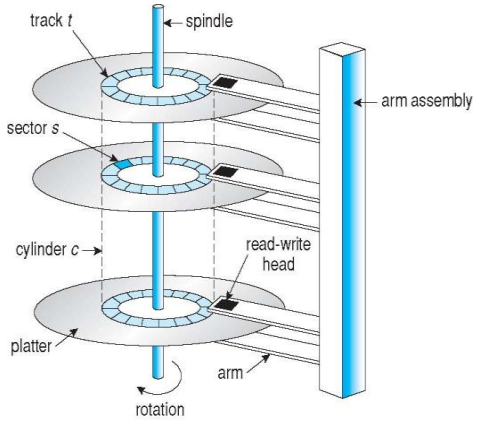
- Motivation: abstract information stored on disk
  - You do not need to remember block, sector, ..
  - We have human readable **names**



- How does it work?
  - Process creates a file, and gives it a name
    - ✓ Other processes can access the file by that name
  - Naming conventions are OS dependent
    - ✓ Usually names are alphabetic characters (as many as 255)
    - ✓ Digits and special characters are sometimes allowed
    - ✓ MS-DOS and Windows are not case sensitive, UNIX family is

File Attributes

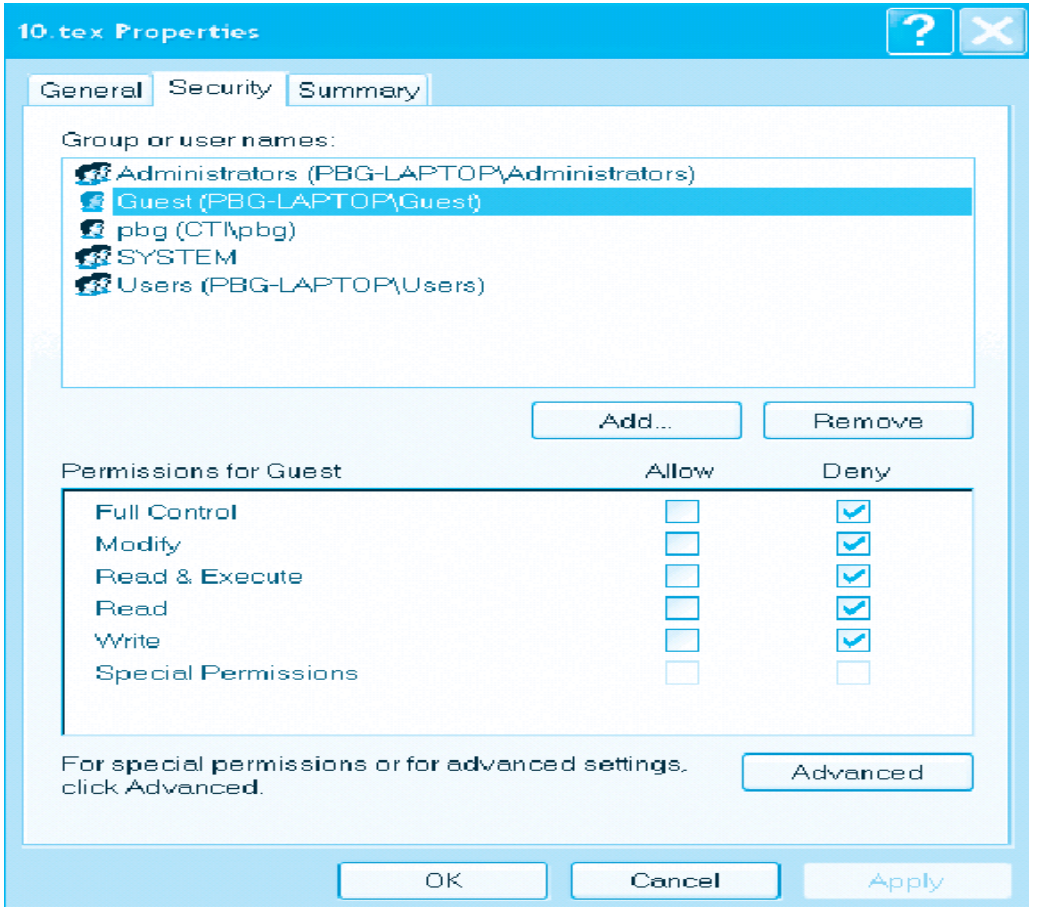
- File-specific info maintained by the OS
  - File size, modification date, creation time, etc.
  - Varies a lot across different OSes



- Some examples:
  - Name – only information kept in human-readable form
  - Identifier – unique tag (number) identifies file within file system
  - Type – needed for systems that support different types
    - ✓ What is the difference between text files and binary files?
  - Location – pointer to file location on device
  - Size – current file size
  - Protection – controls who can do reading, writing, executing
  - Time, date, and user identification – data for protection, security, and usage monitoring

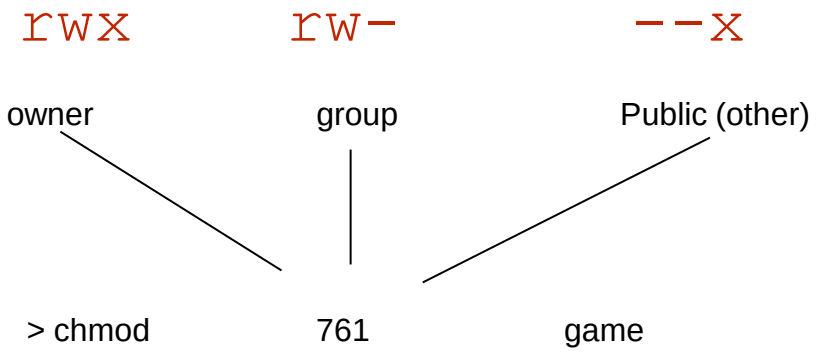
File Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List



Linux example

- Mode of access: read, write, execute `rwx`
- Three classes of users:
  - Owner:
  - Group:
  - Public (other):




```
7: 111
6: 110
1: 001
```

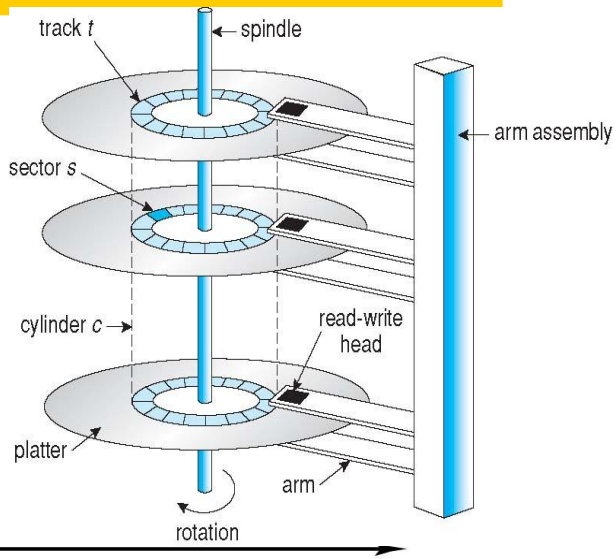
How would you give `r-x` rights to group?

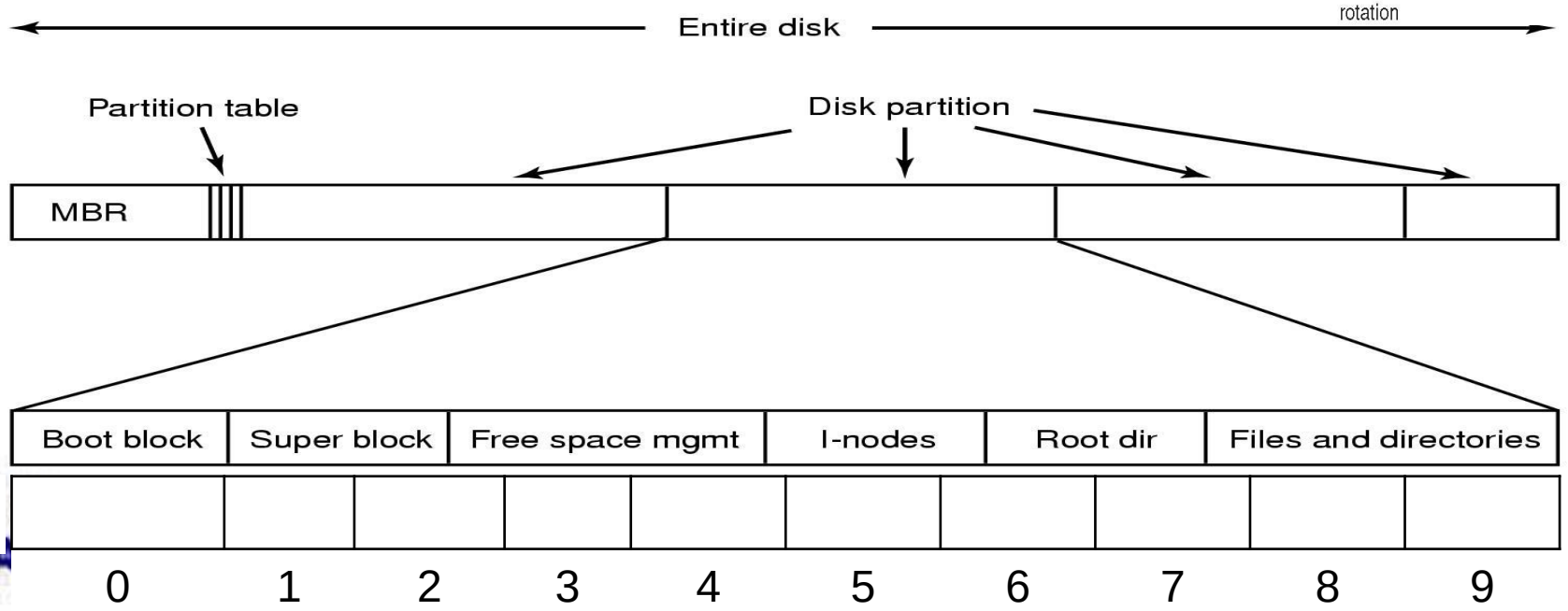
Outline

- Basics of File Systems
- Directory and Unix File System:
  - inodes
  - Directory operations (mostly self study)
  - Links of Files: Hard vs. Symbolic
- UNIX I/O System Calls: open, close, read, write, ioctl
- File Representations:
  - FDT, SFT, inode table
  - Fork and inheritance,
  - Filters and redirection
- FILE pointers (fopen) and buffering

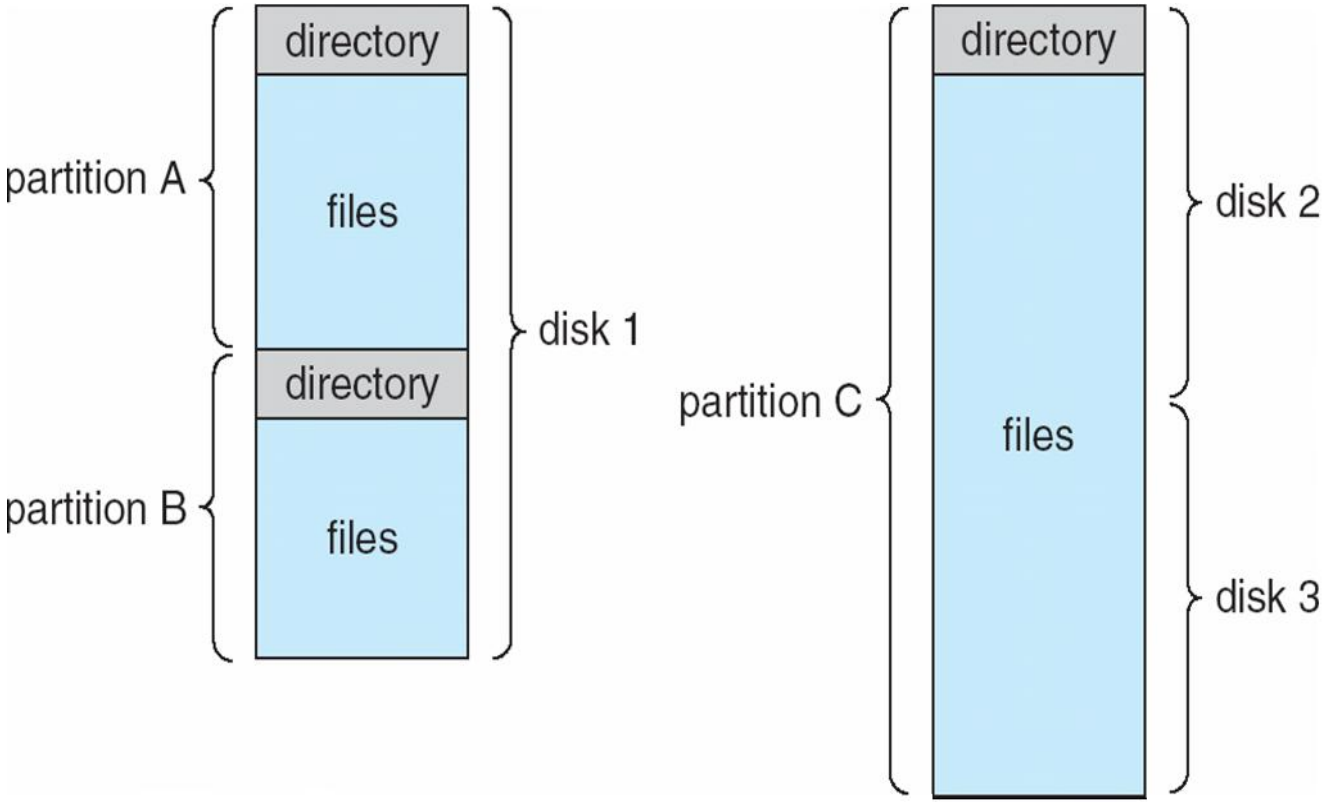
File System Layout: Unix

- File System is stored on disks
  - Disk is divided into 1 or more partitions
  - Sector 0 of disk called **Master Boot Record (MBR)**
  - End of MBR has partition table (start & end address of partitions)
- First block of each partition has **boot block**
  - Loaded by MBR and executed on boot





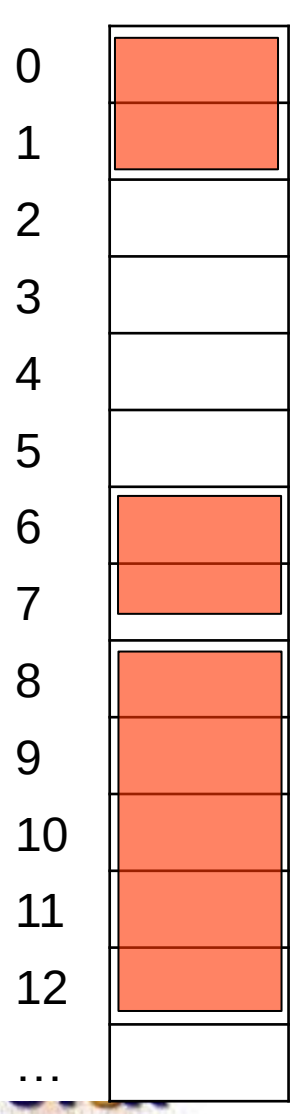
A Typical File-System Organization

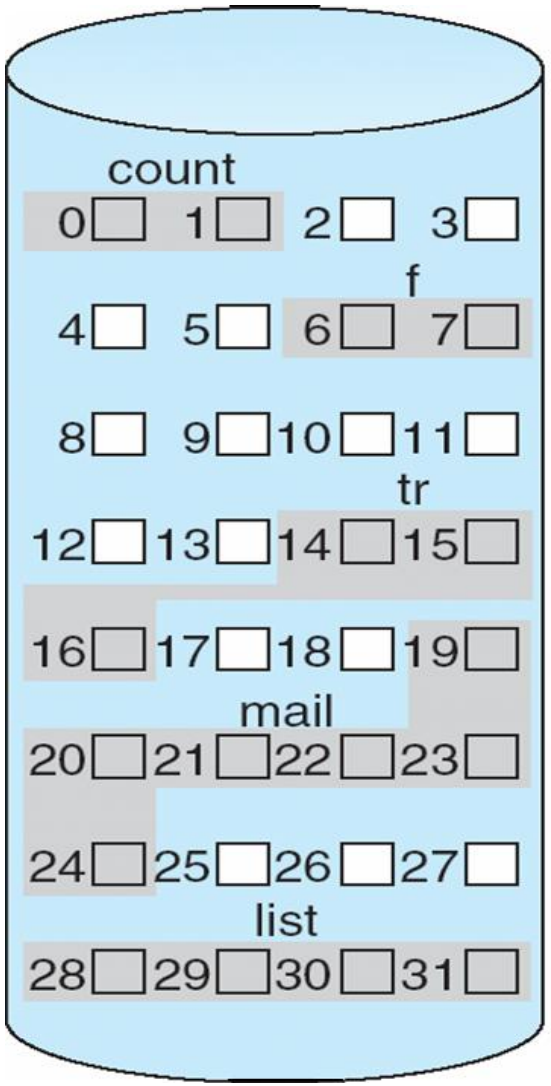


Operations Performed on Directory

- Search for a file
- **Create a file**
- **Delete a file**
- List a directory
- Rename a file
- Traverse the file system

Contiguous Allocation of Disk Space





directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

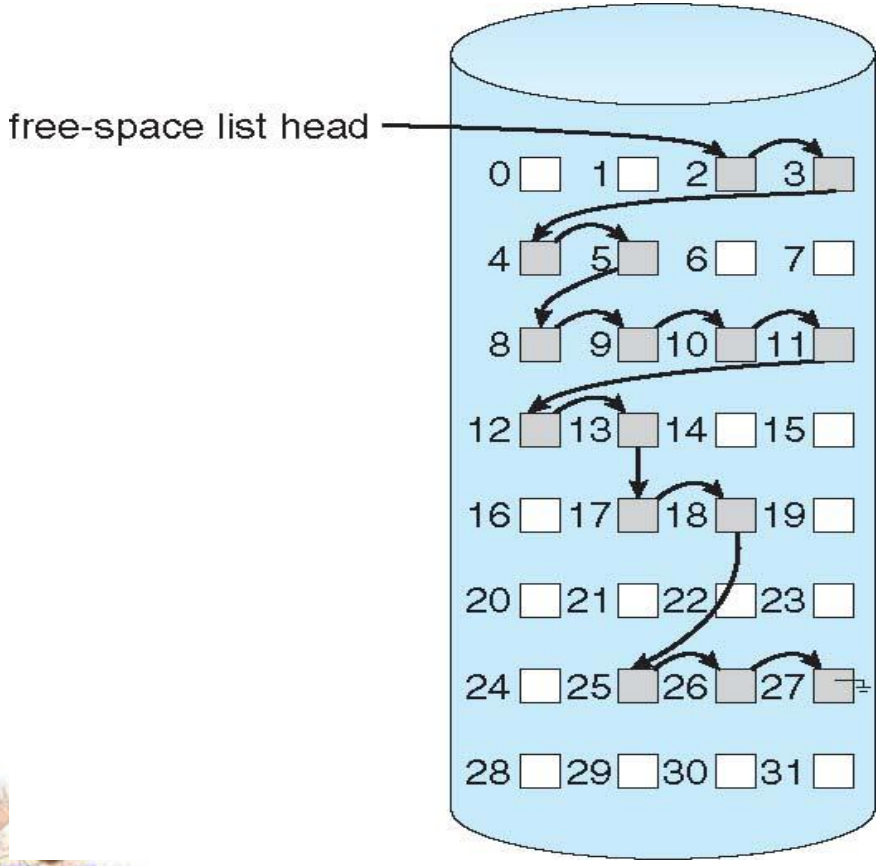
Other types of allocations:

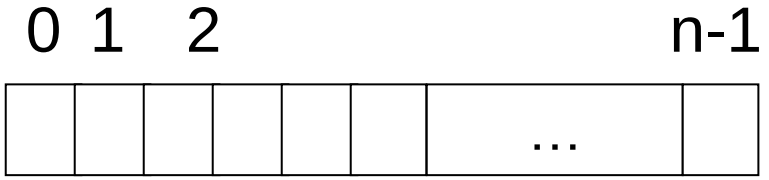
- Linked List Allocation
- **Indexed Allocation (inode)**

Where should we put a new file x requiring 5 blocks?

Managing Free Disk Space

- Two approaches to keep track of free disk blocks
  - **Linked list** and **bitmap** approach



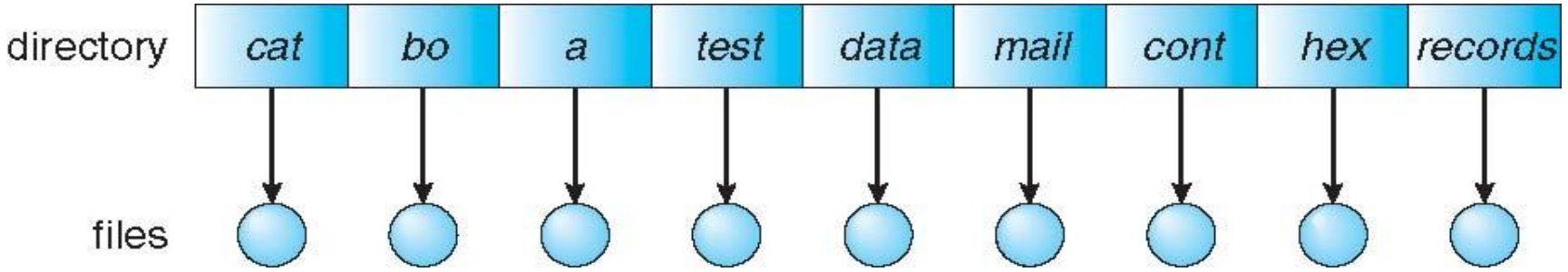


$$bit[i] = \begin{cases} 0 \Rightarrow block[i] \text{ free} \\ 1 \Rightarrow block[i] \text{ occupied} \end{cases}$$

Can you write a function/program to find **contiguous** space for a file x that requires 5 blocks? First fit, best fit, worst fit?

Single-Level Directory

- A single directory for all users

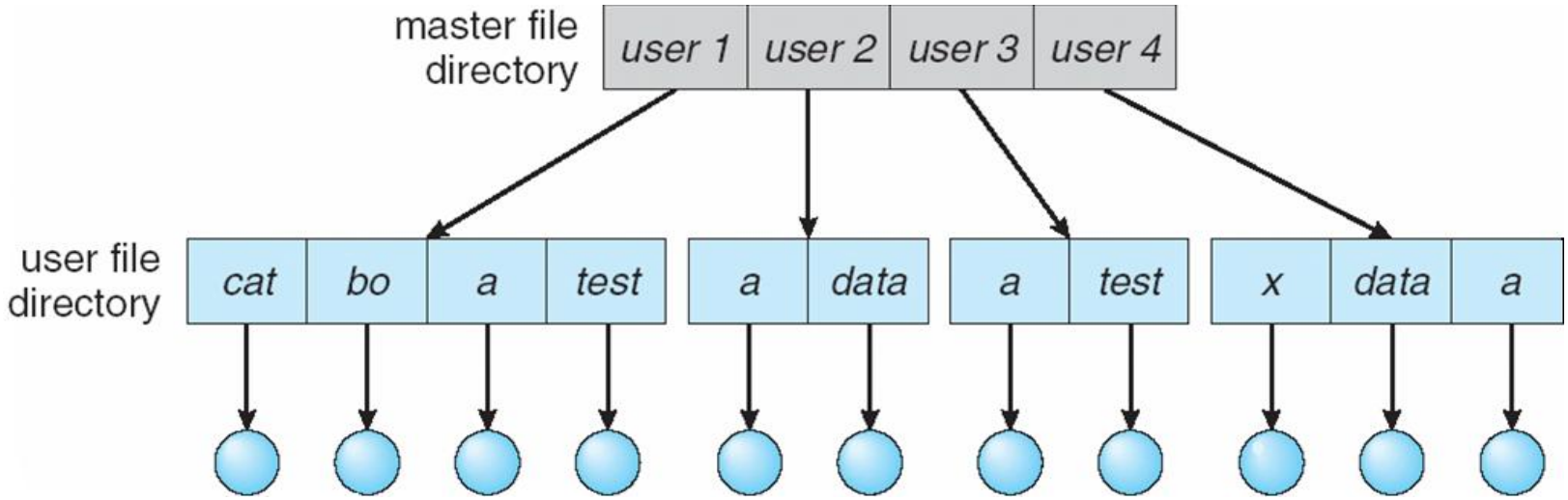


Naming problem

Grouping problem

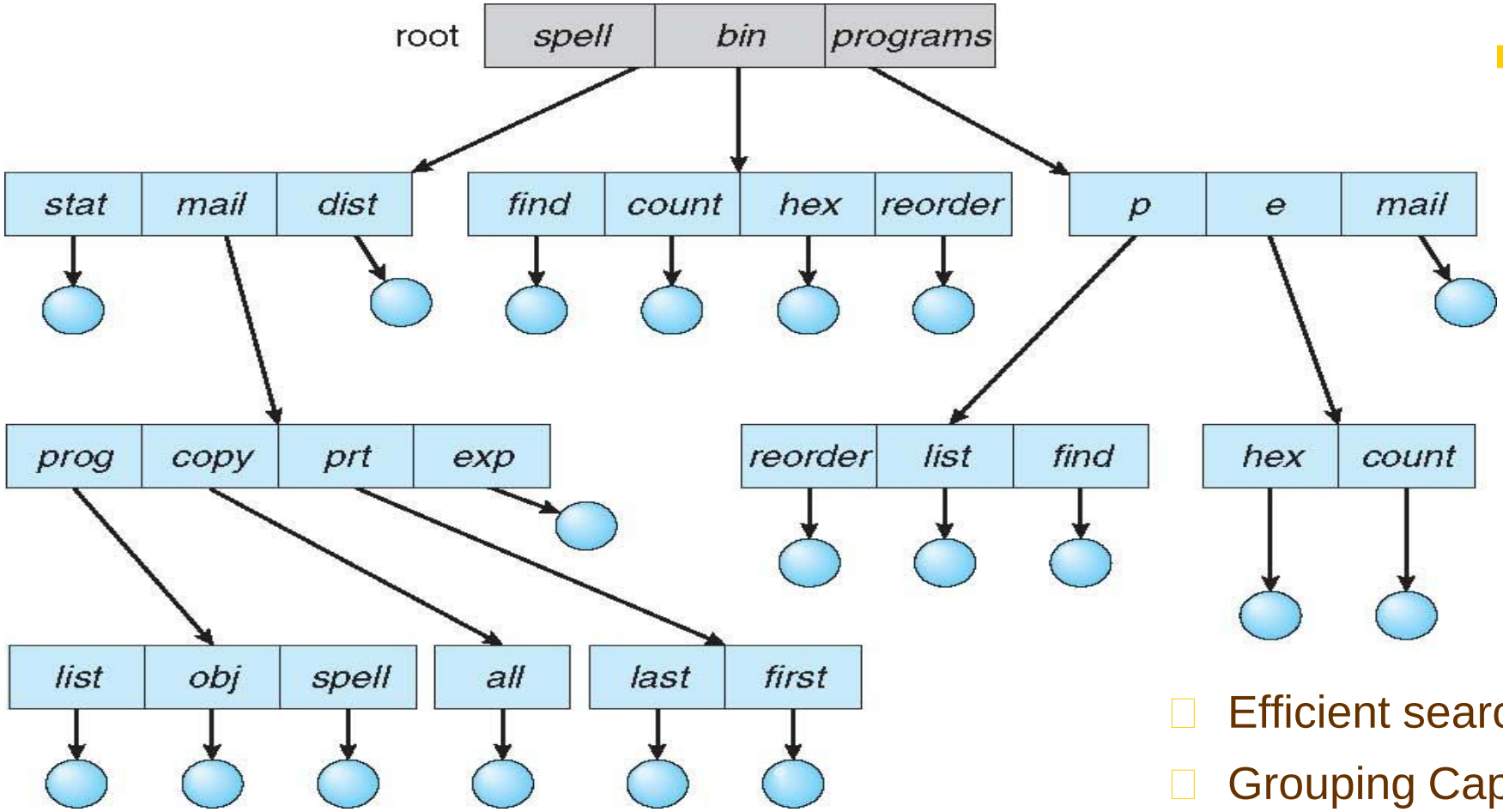
Two-Level Directory

- Separate directory for each user



- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

Tree-Structured Directories



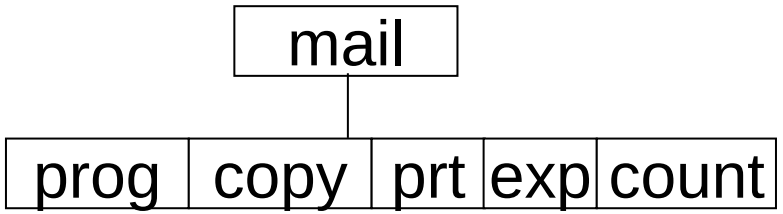
- **Absolute** or **relative** path name
  - Starting from root: `/spell/mail/prog/spell`
  - Depending on Current directory: `../copy/all`

- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - **`cd /spell/mail`**
  - **`cd prog`**
  - **`pwd`**

Tree-Structured Directories (Cont.)

- Usually file operations are done in **current directory**
- Creating a new file
  - **`vi <file-name>`**
- Delete a file
  - **`rm <file-name>`**
- Creating a new subdirectory
  - **`mkdir <dir-name>`**
  - If we are in **`/spell/mail`**

`mkdir count`



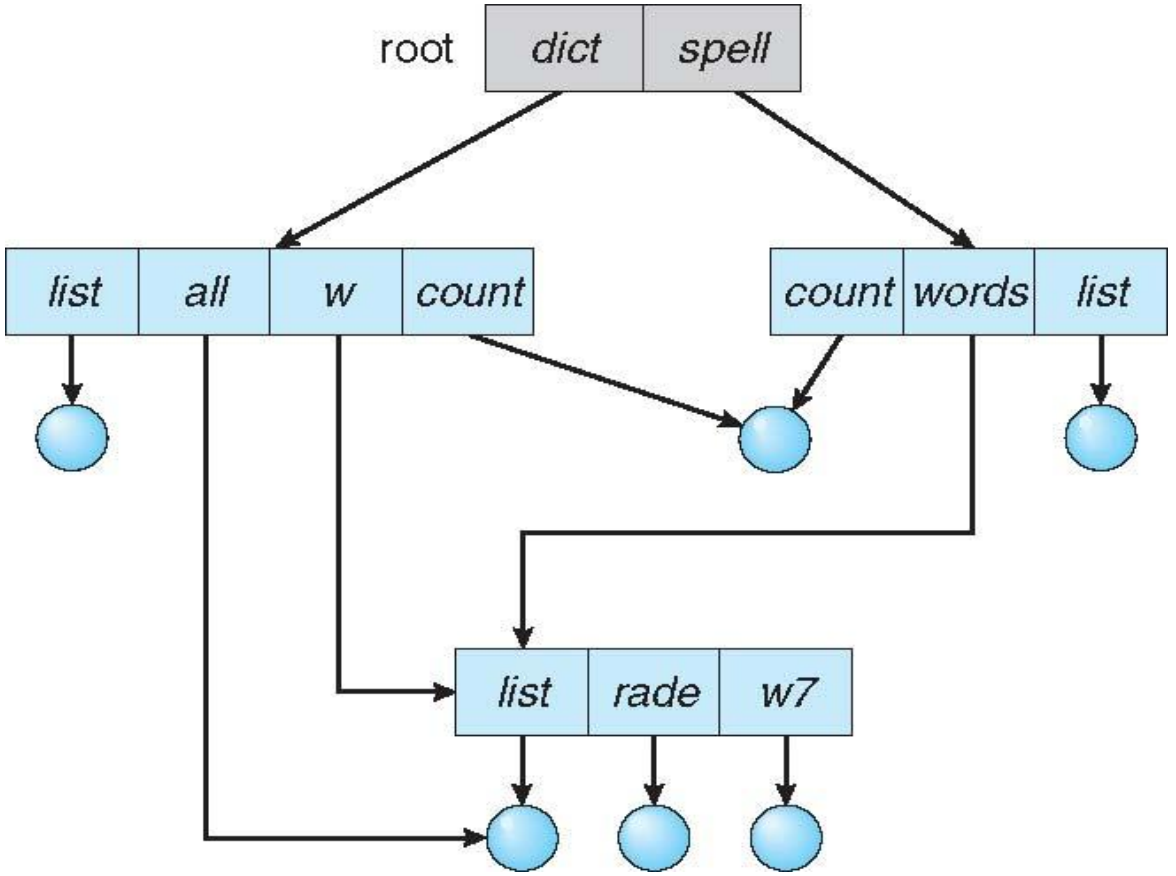
What are the other most common file operations?

- **File Administration:**
  ls, cd, rm, mkdir, rmdir, mv, cp, ln, chown, chgrp, chmod, locate, find, touch, gzip, tar
- **Access File Contents:**
  cat, less, grep, diff, head, tail
- **File Systems:**
  mount, unmount, df, du
- **More help:**
  man

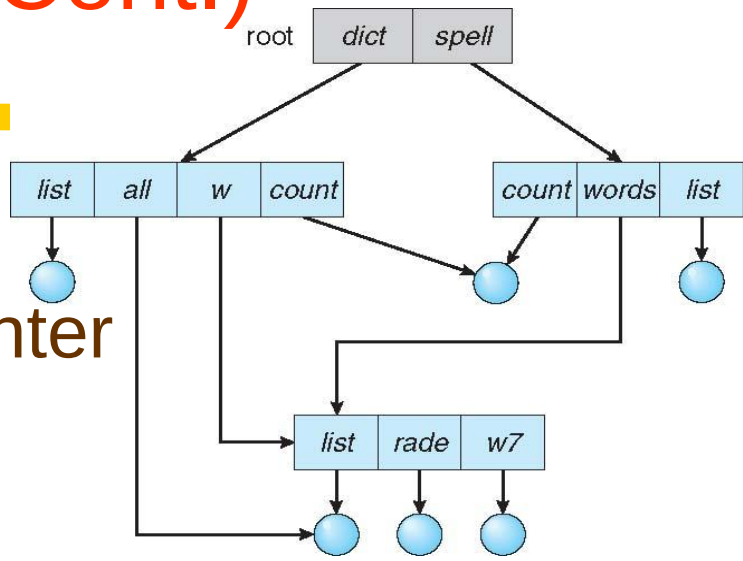
Deleting "mail" ⇒ deleting the entire subtree rooted by "mail"

Acyclic-Graph Directories

- Have shared subdirectories and files



Acyclic-Graph Directories (Cont.)

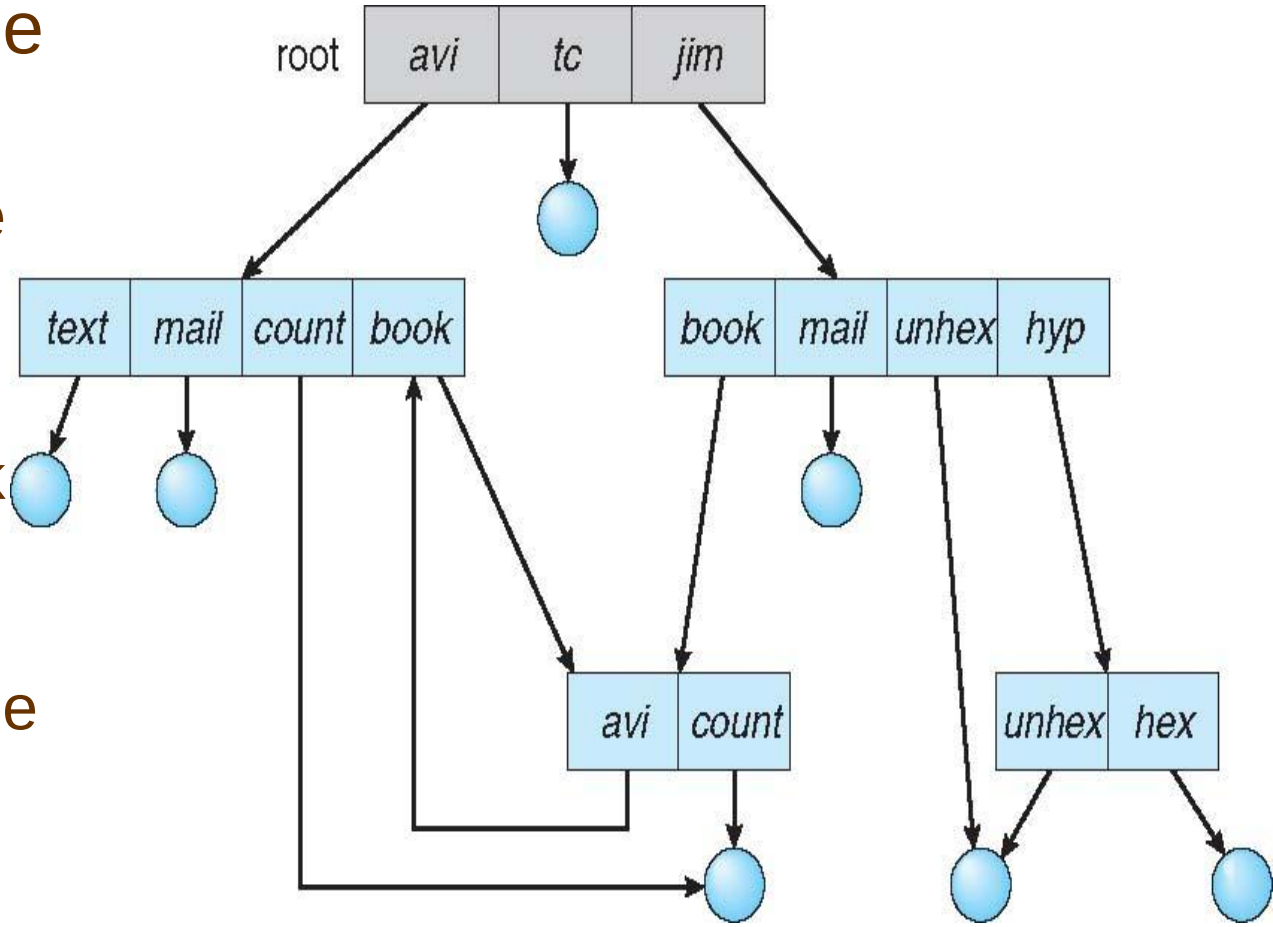


- Two different names (aliasing)
- If *dict* deletes *count* ⇒ dangling pointer

  Solutions:
  - Backpointers, so we can delete all pointers
  - **Entry-hold-count solution** (delete when zero)
- New directory entry type
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file

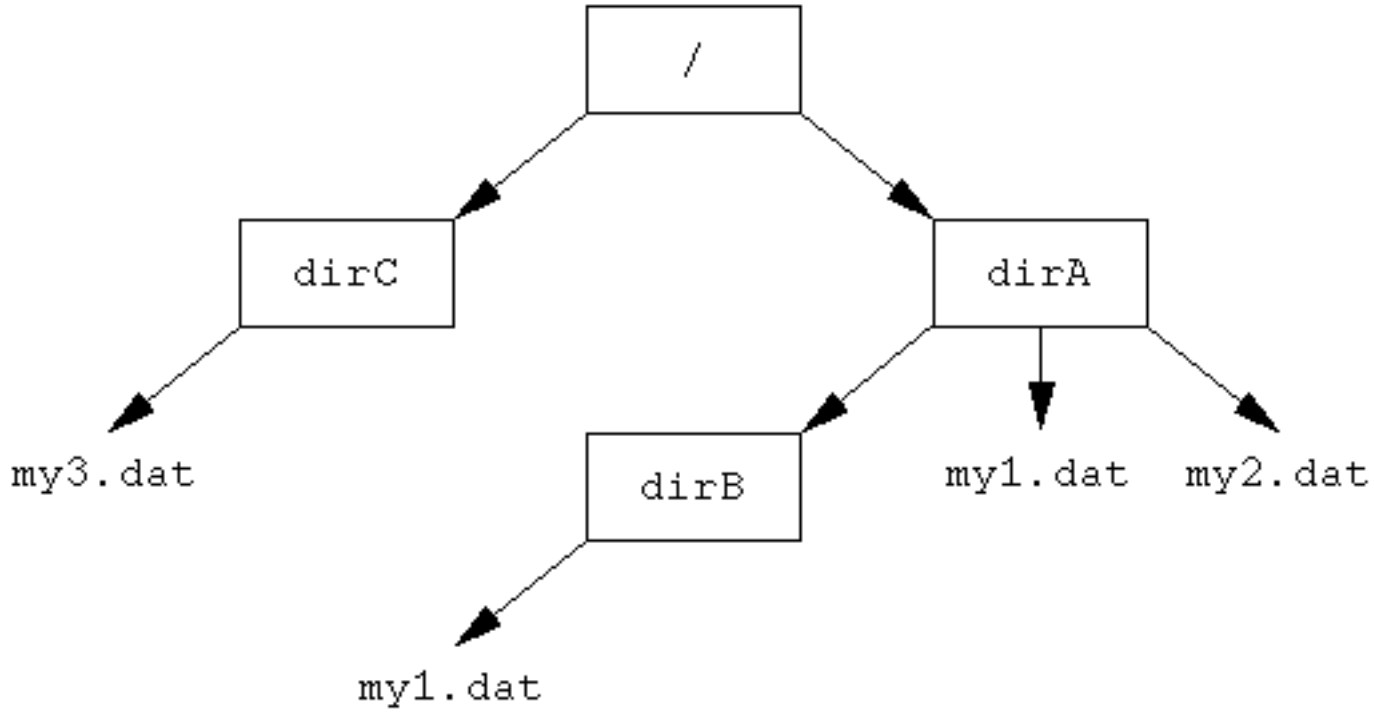
General Graph Directory

- How do we guarantee no cycles?
  - Allow only links to file not subdirectories
  - Garbage collection
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK



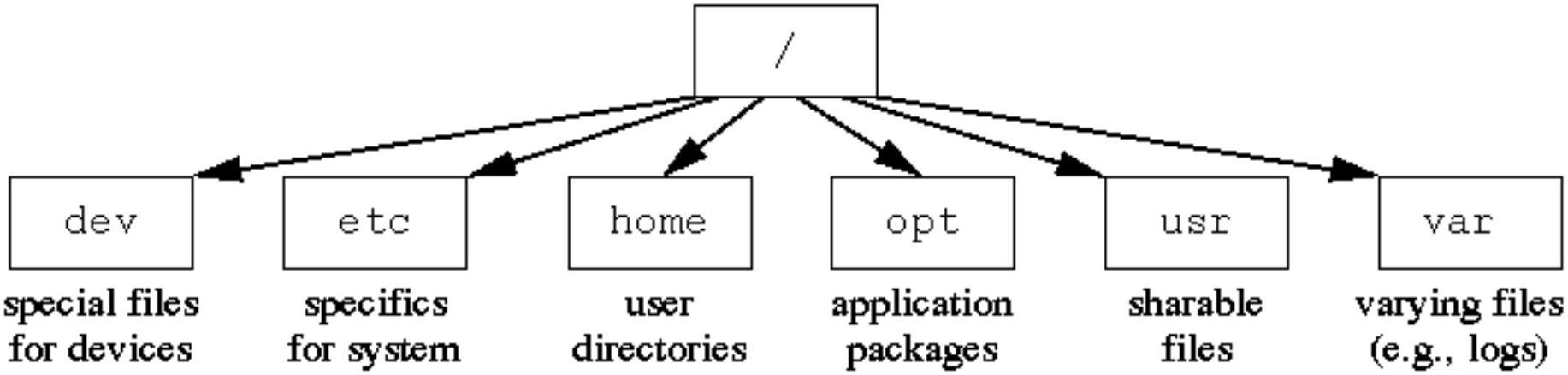
Can you write a cycle detection algorithm?

UNIX Directory: An Example

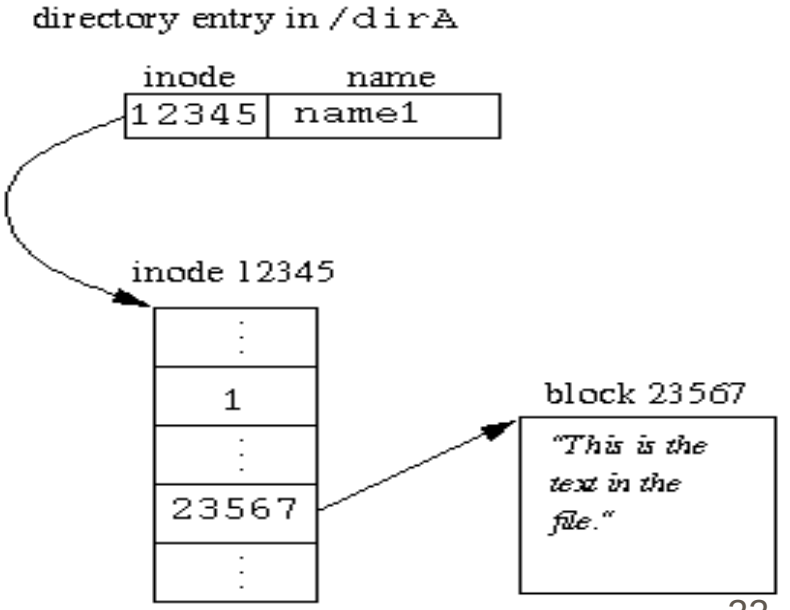


An example tree structures of a file system.

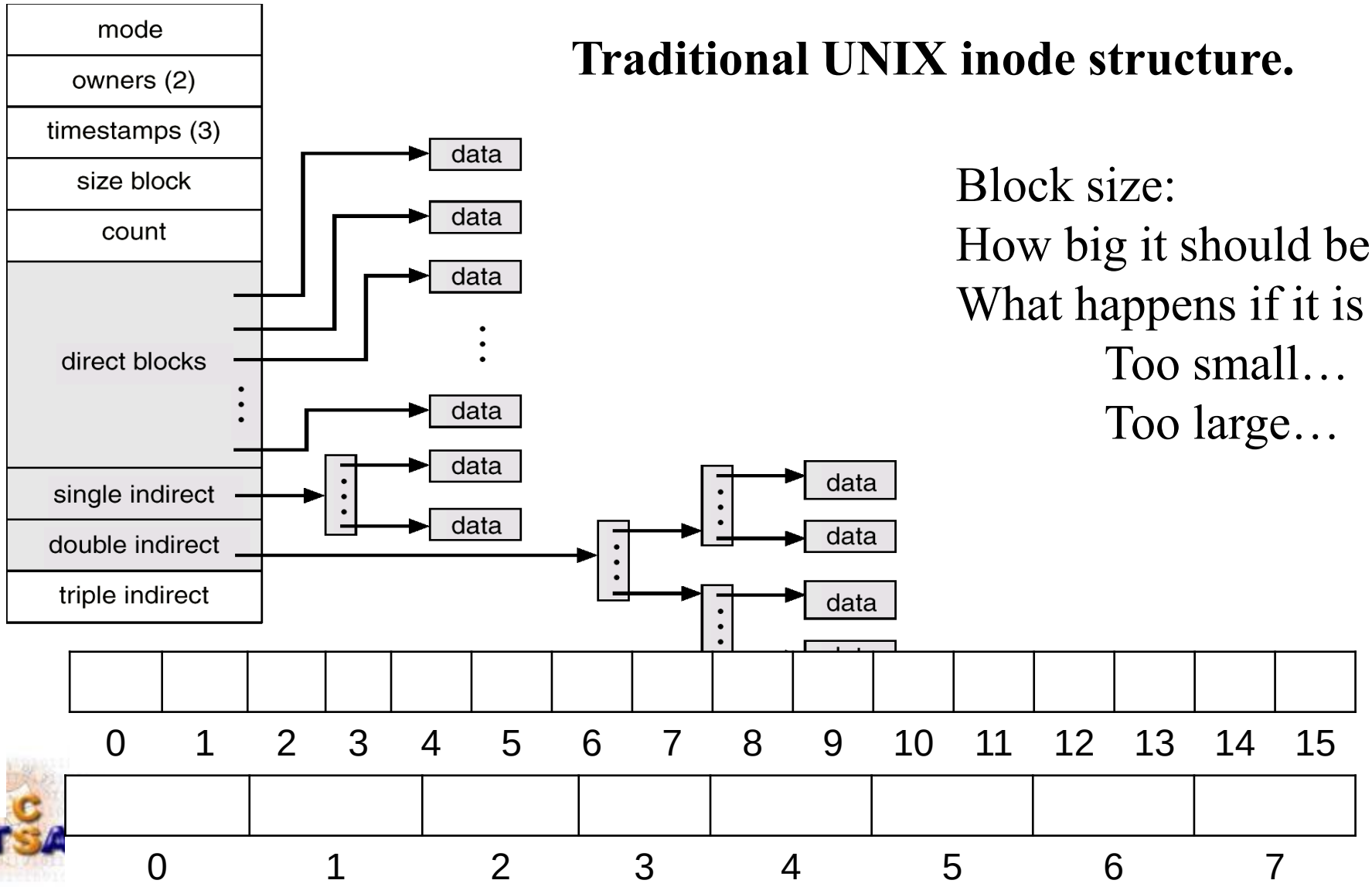
Structure of a Unix File System



A **directory entry** contains only a **name** and an **index** into a table giving information about a file. The table and the index are both referred to as an **inode**.



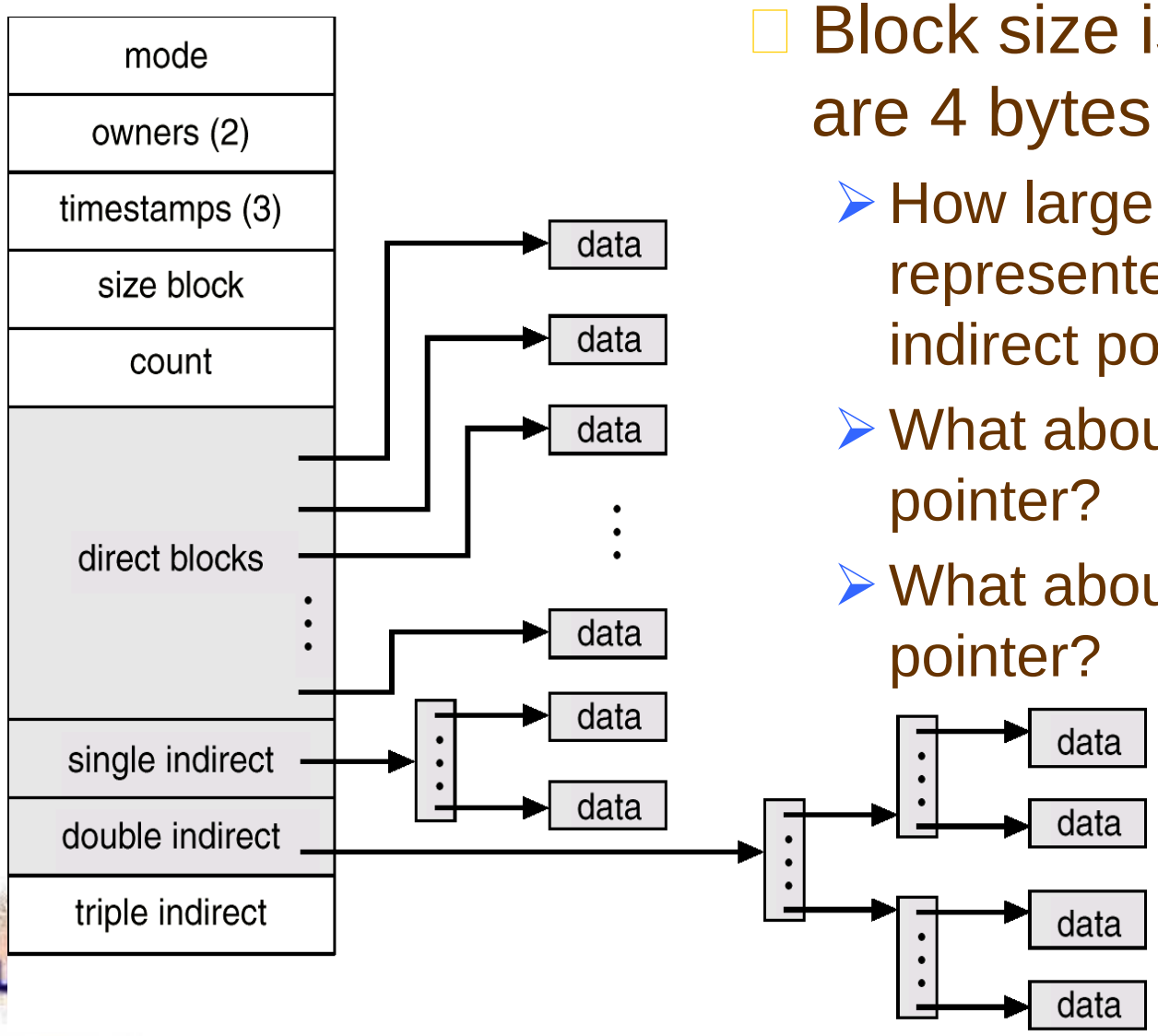
inode: Store File Information



Traditional UNIX inode structure.

Block size:
How big it should be?
What happens if it is
Too small…
Too large…

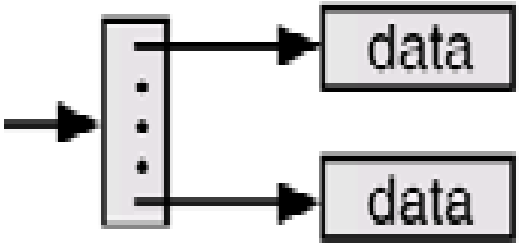
Exercise: File Size and Block Size



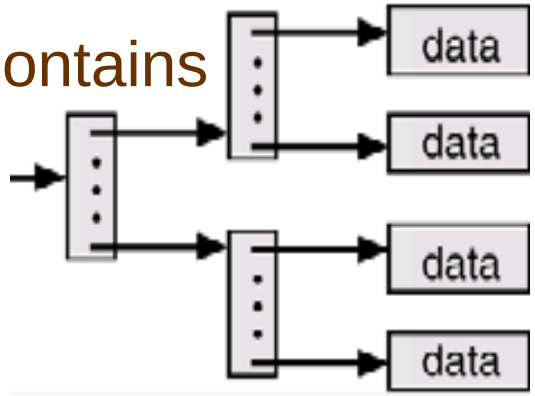
- Block size is 8K and pointers are 4 bytes
  - How large a file can be represented using only one single indirect pointer?
  - What about one double indirect pointer?
  - What about one triple indirect pointer?

File Size and Block Size (cont.)

- Block size is 8K and pointers are 4 bytes
- One single indirect pointer
  - 8K/4= 2K pointers to identify file's 2K blocks
  - File size = 2K * 8K = 16MB



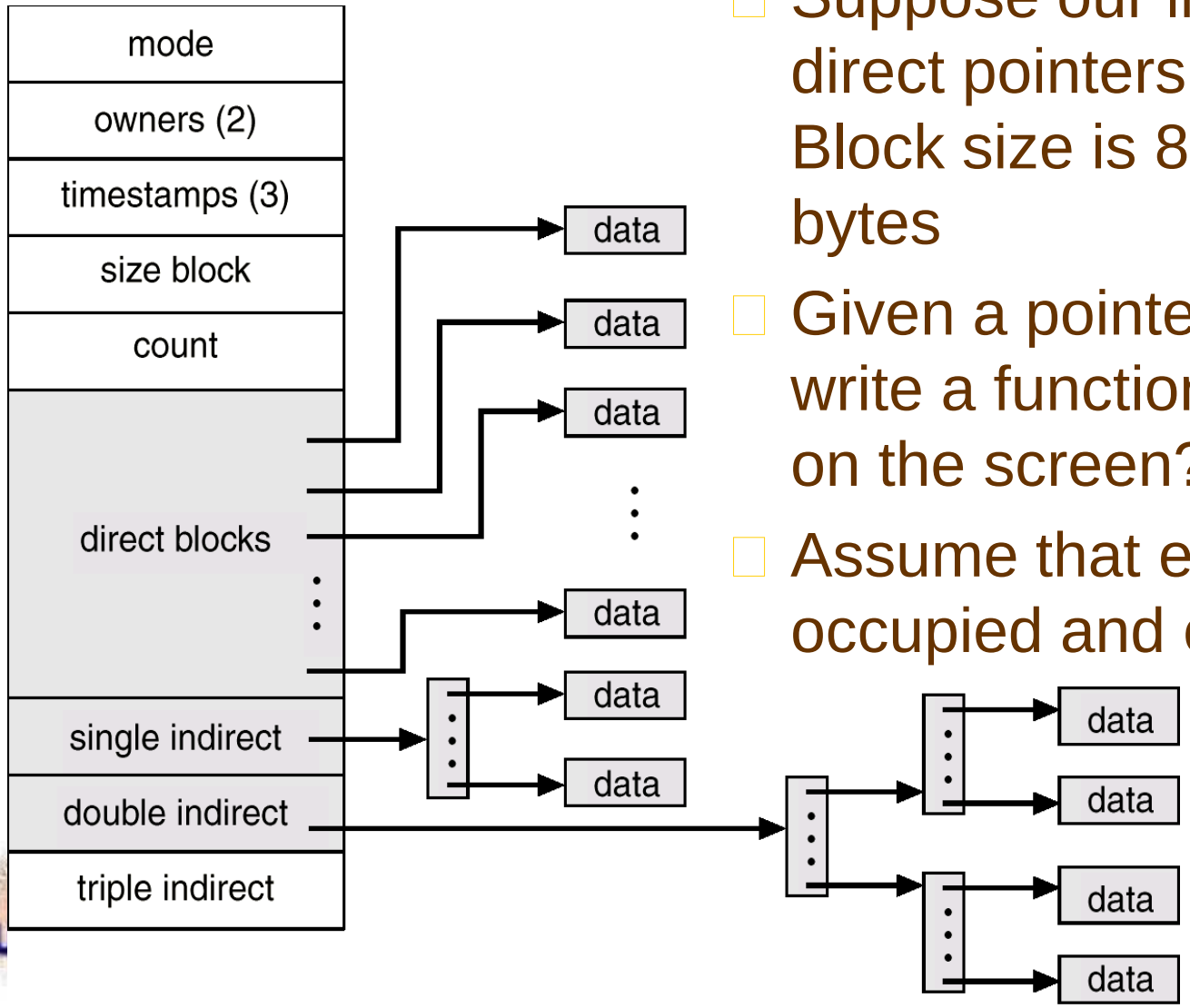
- One double indirect pointer
  - 2K pointers → 2K blocks, where each block contains 2K pointers to identify blocks for the file
  - Number of blocks of the file = 2K*2K = 4M
  - File size = 4M * 8K = 32 GB



- One triple indirect pointer
  - Number of blocks of the file = 2K*2K*2K = 8 *K*K*K
  - File size → 8 *K*K*K * 8K = 64 TB

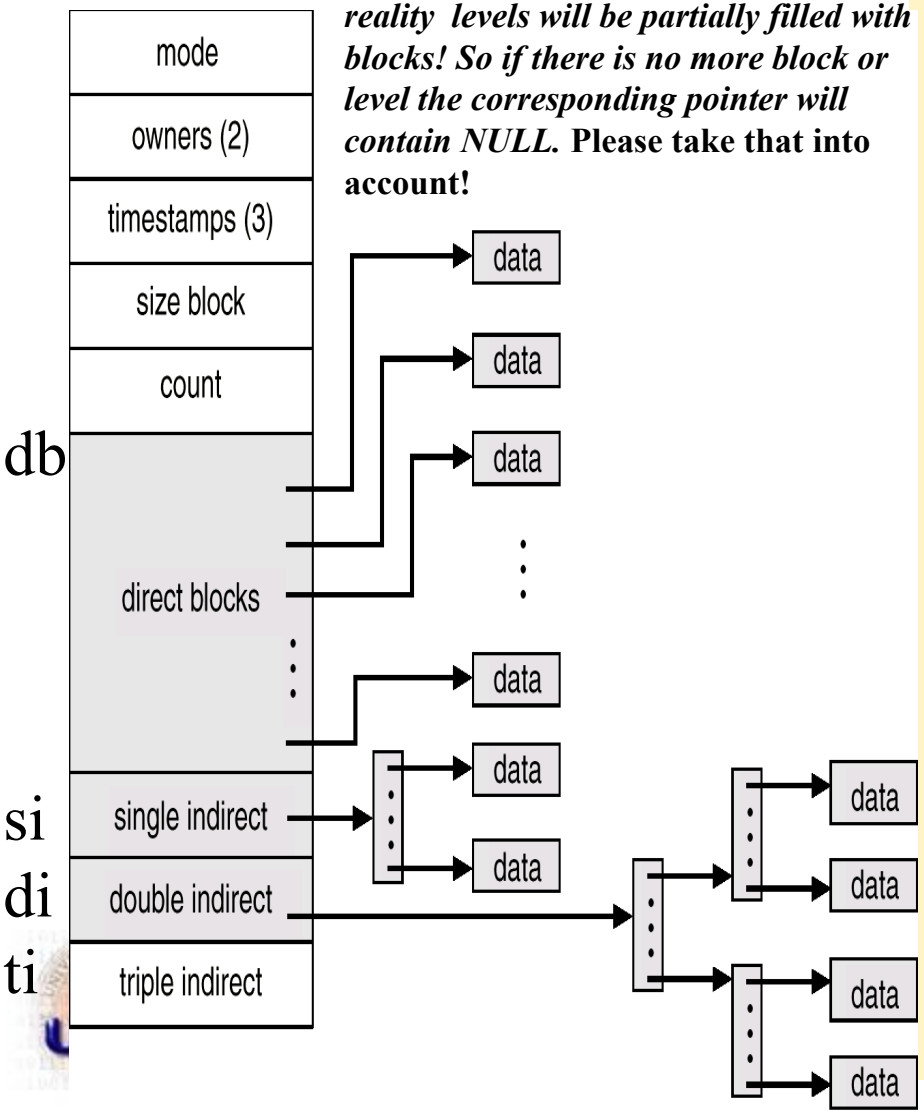
Draw the picture.

Exercise: access to content through inode



- Suppose our inode structure has 10 direct pointers. Also assume that Block size is 8K and pointers are 4 bytes
- Given a pointer to inode structure, write a function to print all the data on the screen?
- Assume that every data block is occupied and contains a string text!

Exercise: access to content through inode



***We assumed all blocks are full. But in reality levels will be partially filled with blocks! So if there is no more block or level the corresponding pointer will contain NULL.* Please take that into account!**

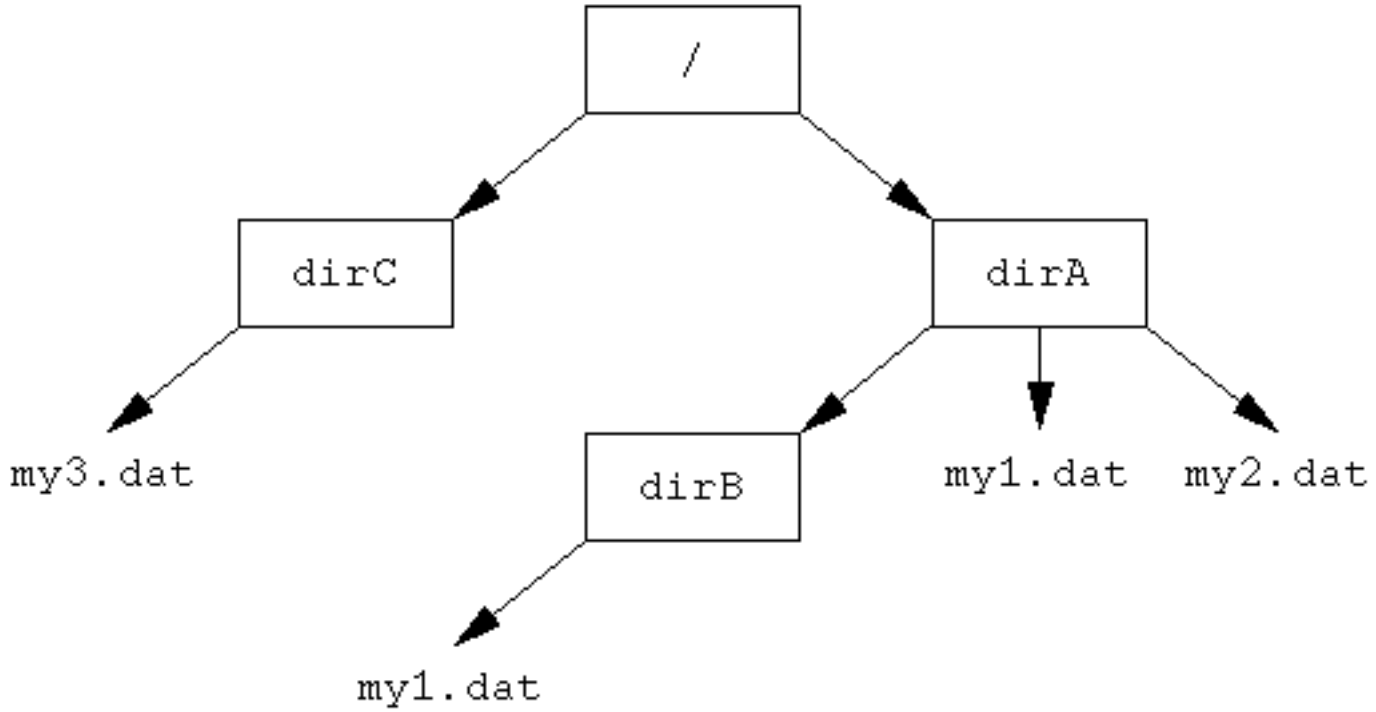
```
#define BLK_SIZE (8*1024)
#define BLK_PTR_SIZE 4
#define NUM_DIRECT_BLK 10
int *print_data(struct inode_t *inode) {
 int i, j, k;
 int num_blk = BLK_SIZE / BLK_PTR_SIZE;

 for (i=0; i < NUM_DIRECT_BLK; i++)
       printf("%s\", inode->db[i]);

 for(i=0; i < num_blk; i++)
       printf("%s\", inode->si[i]);

 for(i=0; i < num_blk; i++)
   for(j=0; j < num_blk; j++)
       printf("%s\", inode->di[i][j]);

 for(i=0; i < num_blk; i++)
   for(j=0; j < num_blk; j++)
     for(k=0; k < num_blk; k++)
       printf("%s\", inode->ti[i][j][k]);
}
```

Outline

- Basics of File Systems
- Directory and Unix File System:
  - inodes
  - Directory operations (mostly self study)
  - Links of Files: Hard vs. Symbolic
- UNIX I/O System Calls: open, close, read, write, ioctl
- File Representations:
  - FDT, SFT, inode table
  - Fork and inheritance,
  - Filters and redirection
- FILE pointers (fopen) and buffering

Directory Operations

(mostly self study)



Current Working Directory

```
#include <unistd.h>
int chdir(const char *path);                  // cd abc
char *getcwd(char *buf, size_t size);         // pwd
```

Program 5.1, getcwdpathmax.c, page 149

```
#include <limits.h>
#include <stdio.h>
#include <unistd.h>
#ifndef PATH_MAX
#define PATH_MAX 255
#endif

int main(void) {
    char mycwd[PATH_MAX];

    if (getcwd(mycwd, PATH_MAX) == NULL) {
        perror("Failed to get current working directory");
        return 1;
    }
    printf("Current working directory: %s\n", mycwd);
    return 0;
}
```

PATH_MAX to determine the size of the buffer needed

`pathconf`: Maximum Pathname Length

```
#include <unistd.h>
long fpathconf(int filedes, int name);
long pathconf(const char *path, int name);
long sysconf(int name);
```

Program 5.2, getcwdpathconf.c, page 150

ref

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

int main(void) {
    long maxpath;
    char *mycwdp;

    if ((maxpath = pathconf(".", _PC_PATH_MAX)) == -1) {
        perror("Failed to determine the pathname length");
        return 1;
    }
    if ((mycwdp = (char *) malloc(maxpath)) == NULL) {
        perror("Failed to allocate space for pathname");
        return 1;
    }
    if (getcwd(mycwdp, maxpath) == NULL) {
        perror("Failed to get current working directory");
        return 1;
    }
    printf("Current working directory: %s\n", mycwdp);
    return 0;
}
```

Functions for Directory Access

```
#include <dirent.h>
DIR *opendir(const char *filename);
```

- provides a handle for the other functions

```
struct dirent *readdir(DIR *dirp);
```

- gets the **next entry** in the directory

```
void rewinddir(DIR *dirp);
```

- restarts from the beginning

```
int closedir(DIR *dirp);
```

- closes the handle.

inode	name

- Functions are **not re-entrant** (like strtok)

Program 5.3, shownames.c, page 153

Lists Files in a Directory

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

int main(int argc, char *argv[]) {
    struct dirent *direntp;
    DIR *dirp;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s directory_name\n", argv[0]);
        return 1;
    }

    if ((dirp = opendir(argv[1])) == NULL) {
        perror ("Failed to open directory");
        return 1;
    }

    while ((direntp = readdir(dirp)) != NULL)
        printf("%s\n", direntp->d_name);
    while ((closedir(dirp) == -1) && (errno == EINTR)) ;
    return 0;
}
```

Functions to Access File Status

```
#include <sys/stat.h>
int stat(const char *restrict path,
         struct stat *restrict buf);
```

- use the name of a file

```
int lstat(const char *restrict path,
          struct stat *restrict buf);
```

- Same as stat; for symbolic link → information about link, not the file it links to

```
int fstat(int fildes, struct stat *buf);
```

- used for open files

Contents of the `struct stat`

- System dependent
- At least the following fields

```
dev_t     st_dev;      /* device ID of device containing file */
ino_t     st_ino;      /* file serial number */
mode_t    st_mode;     /* file mode */
nlink_t   st_nlink;    /* number of hard links */
uid_t     st_uid;      /* user ID of file */
gid_t     st_gid;      /* group ID of file */
off_t     st_size;     /* file size in bytes (regular files) */
                       /* path size (symbolic links) */
time_t    st_atime;    /* time of last access */
time_t    st_mtime;    /* time of last data modification */
time_t    st_ctime;    /* time of last file status change */
```

Example: Last Access Time of A File

Example 5.8, printaccess.c, page 155

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

void printaccess(char *path) {
   struct stat statbuf;

   if (stat(path, &statbuf) == -1)
      perror("Failed to get file status");
   else
      printf("%s last accessed at %s", path, ctime(&statbuf.st_atime));
}
```

Access and Modified Times

Exercise 5.9, printaccessmod.c, page 156

```
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <sys/stat.h>
#define CTIME_SIZE 26

void printaccessmod(char *path) {
   char atime[CTIME_SIZE];   /* 26 is the size of the ctime string */
   struct stat statbuf;

   if (stat(path, &statbuf) == -1)
      perror("Failed to get file status");
   else {
      strncpy(atime, ctime(&statbuf.st_atime), CTIME_SIZE - 1);
      atime[CTIME_SIZE - 2] = 0;
      printf("%s accessed: %s modified: %s", path, atime,
               ctime(&statbuf.st_mtime));
   }
}
```

File Mode

Use the following **macros** to test the **`st_mode`** field for the file type.

```
m=statbuf.st_mode
```

`S_ISBLK(m)`	block special file
`S_ISCHR(m)`	character special file
`S_ISDIR(m)`	directory
`S_ISFIFO(m)`	pipe or FIFO special file
`S_ISLNK(m)`	symbolic link
`S_ISREG(m)`	regular file
`S_ISSOCK(m)`	socket

Check Whether a File is A Directory

Example 5.10, isdirectory.c, page 157

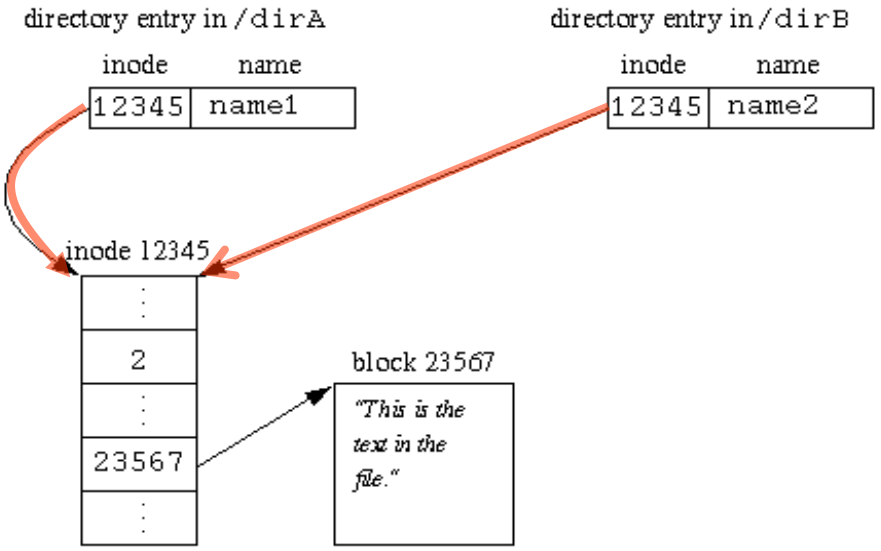
```
#include <stdio.h>
#include <time.h>
#include <sys/stat.h>

int isdirectory(char *path) {
    struct stat statbuf;

    if (stat(path, &statbuf) == -1)
        return 0;
    else
        return S_ISDIR(statbuf.st_mode);
}
```

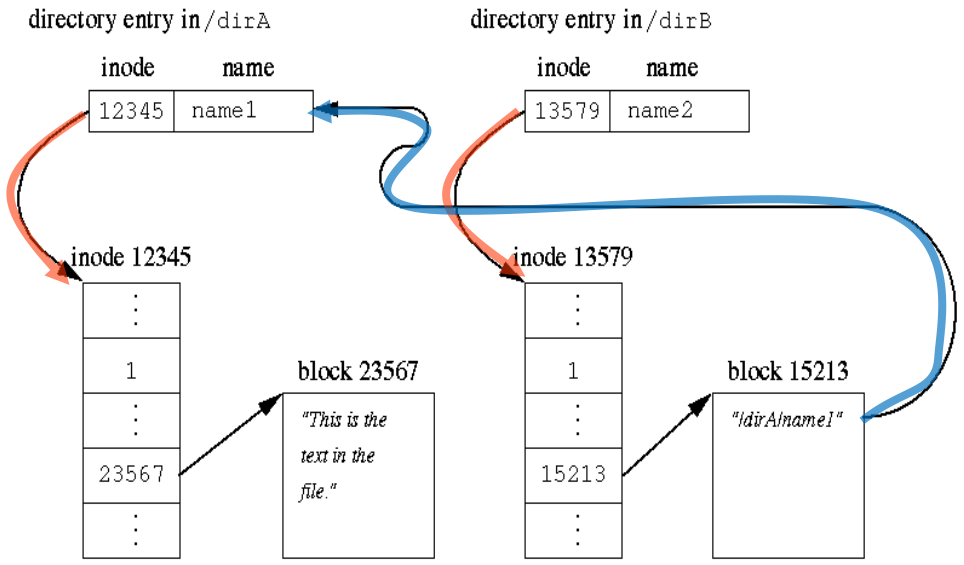
Outline

- Basics of File Systems
- Directory and Unix File System:
  - inodes
  - Directory operations (mostly self study)
  - Links of Files: Hard vs. Symbolic
- UNIX I/O System Calls: open, close, read, write, ioctl
- File Representations:
  - FDT, SFT, inode table
  - Fork and inheritance,
  - Filters and redirection
- FILE pointers (fopen) and buffering

Links in Unix

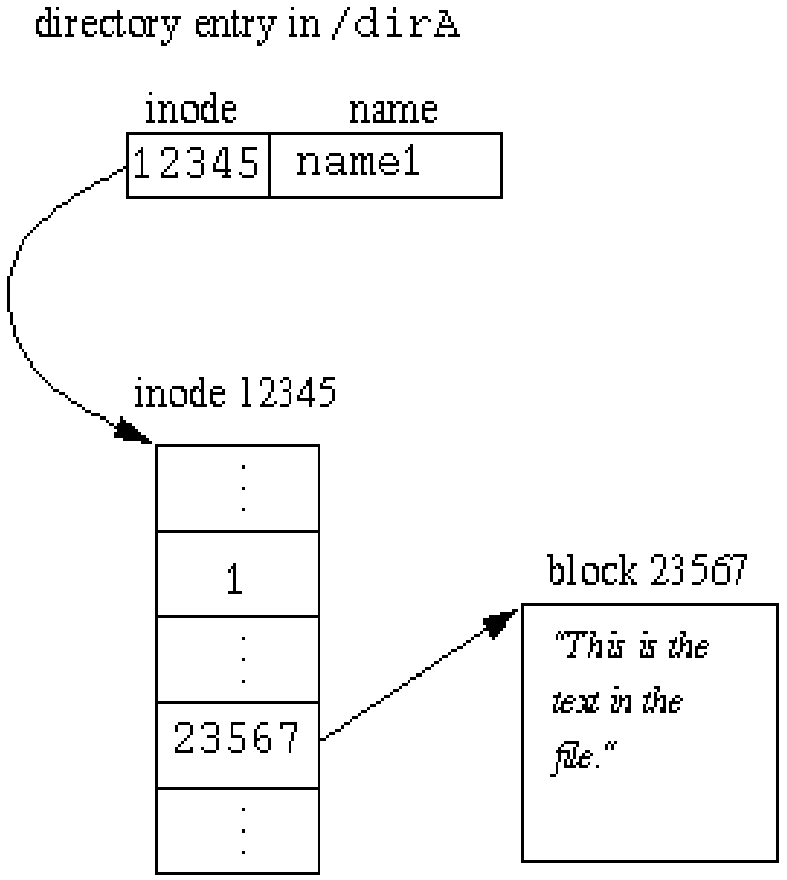
- **Link**: association between a filename and an inode
- Two types of links in Unix:
  - **Hard link:** A hard link just creates another file (a new entry in directory) with a link to the same underlying inode.
  - **Symbolic/Soft link:** link to another filename in the file system





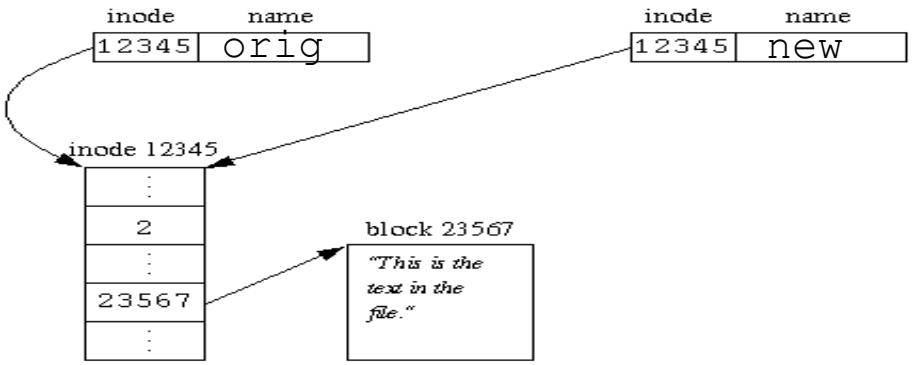
First hard link…

- When a file is created
  - A new inode is assigned
  - A new directory entry is created: directly link the filename to its inode
    - → first **hard** link
- Additional hard links can be created by
  - `ln oldname newname`
  - inode tracks the number of hard links to the inode



Creating links

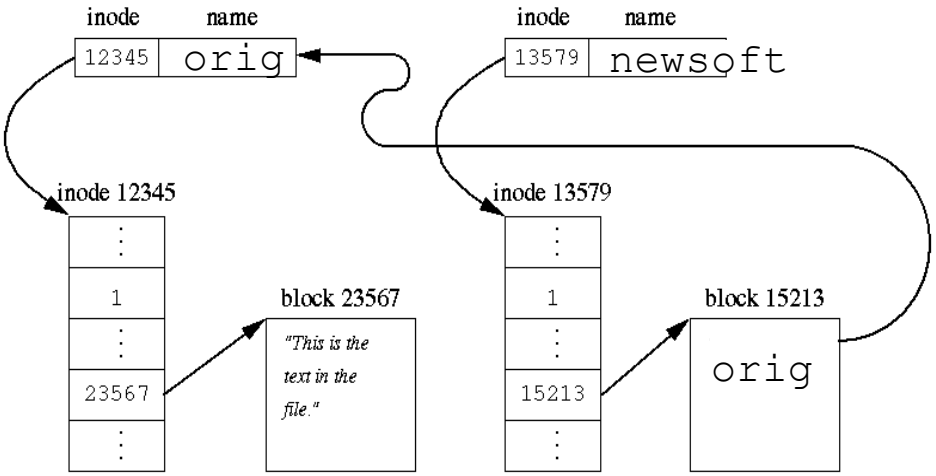
- Create a hardlink: `ln oldname newname`




```
3733/examples/lec05/link$ ln orig new
3733/examples/lec05/link$ ls -la

iu 4096 Sep 21 16:58 .
iu 4096 Sep 21 16:58 ..
iu 8872 Sep 21 16:36 new
iu 8872 Sep 21 16:36 orig
```

$ ls -i

- Create a softlink: `ln -s oldname newname`




```
33/examples/lec05/link$ ls

33/examples/lec05/link$ ln -s orig newsoft
33/examples/lec05/link$ ls -la

 4096 Sep 21 17:01 .
 4096 Sep 21 16:58 ..
 8872 Sep 21 16:36 new
    4 Sep 21 17:01 newsoft -> orig
 8872 Sep 21 16:36 orig
```

$ ls -i

Hard Links in Unix

- New hard link to an existing file
  - creates a new directory entry
  - no other additional disk space
  - Increment the link count in the inode
- Remove a hard link
  - the rm command or the unlink system call
  - decrement the link count in the inode
- When inode's link count → 0
  - The inode and associated disk space are freed

Example: Hard Link

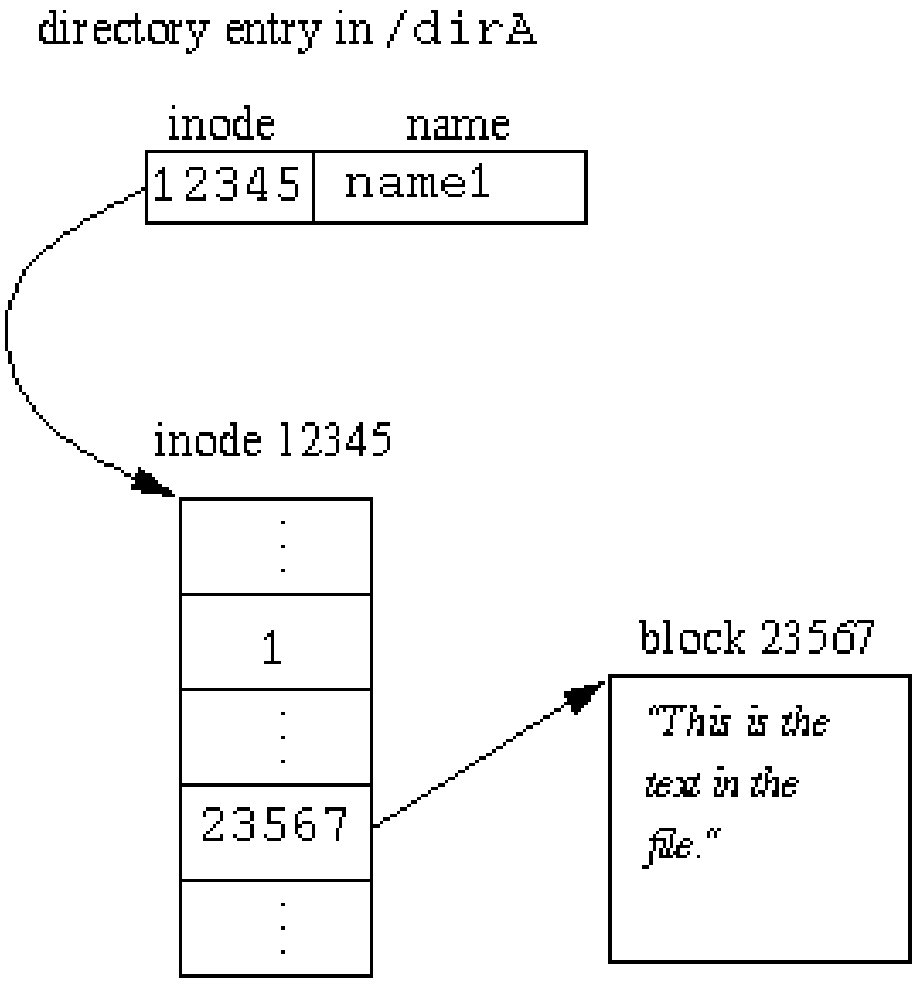


Figure 5.4 (page 163): A directory entry, inode, and data block for a simple file;

Example: Hard Link (cont.)

```
if (link("/dirA/name1", "/dirB/name2") == -1)…
```

```
> ln /dirA/name1 /dirB/name2
```

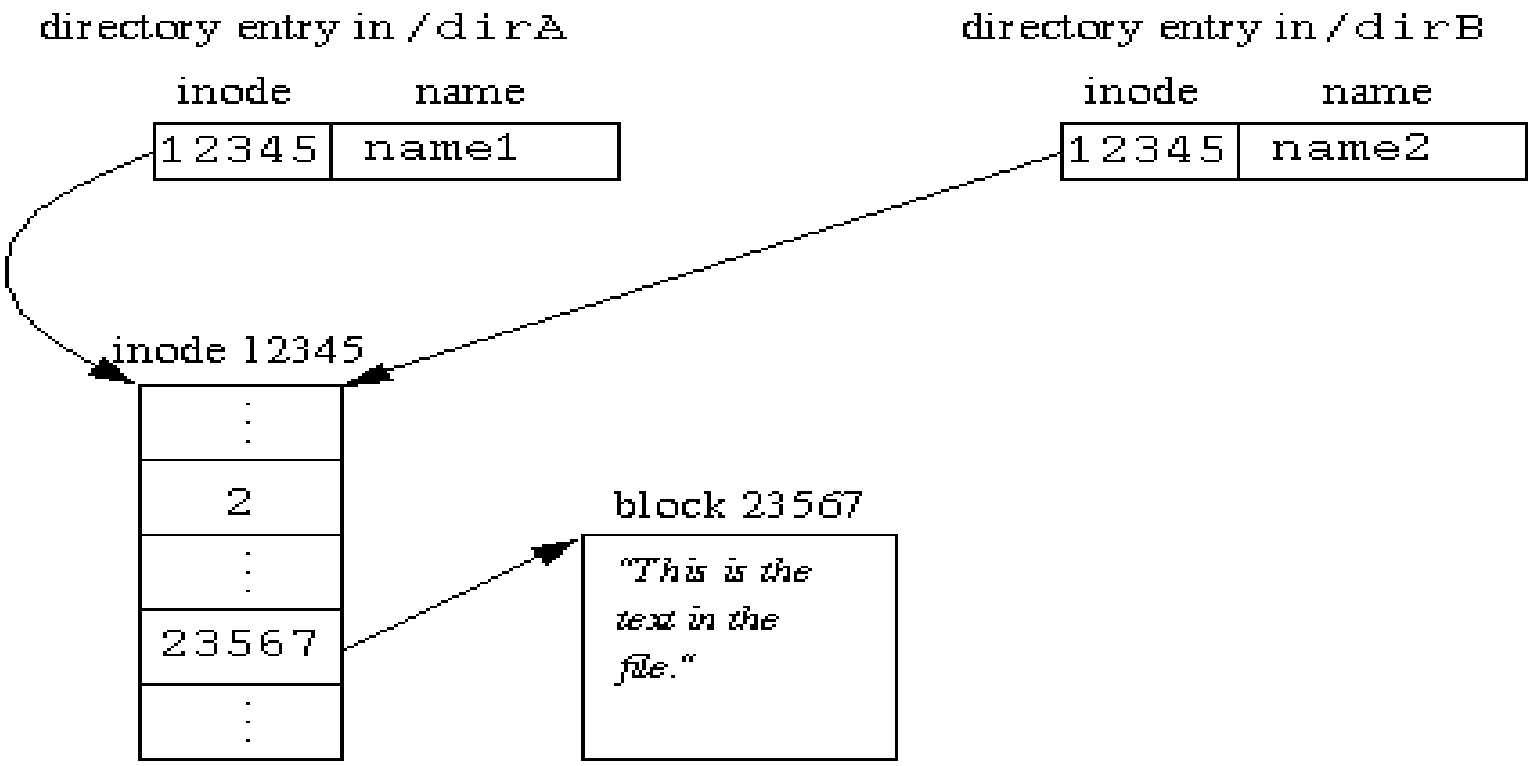


Figure 5.5 (page 165): Two hard links to the same file;

File Modification with Multiple Hard Links

- **Exercise 5.17, page 166**: What would happen to Figure 5.5 after the following operations:

```
open("/dirA/name1");
read
close
modify memory image of the file
unlink("/dirA/name1");
open("/dirA/name1");
write
close
```

File Modification (cont.)

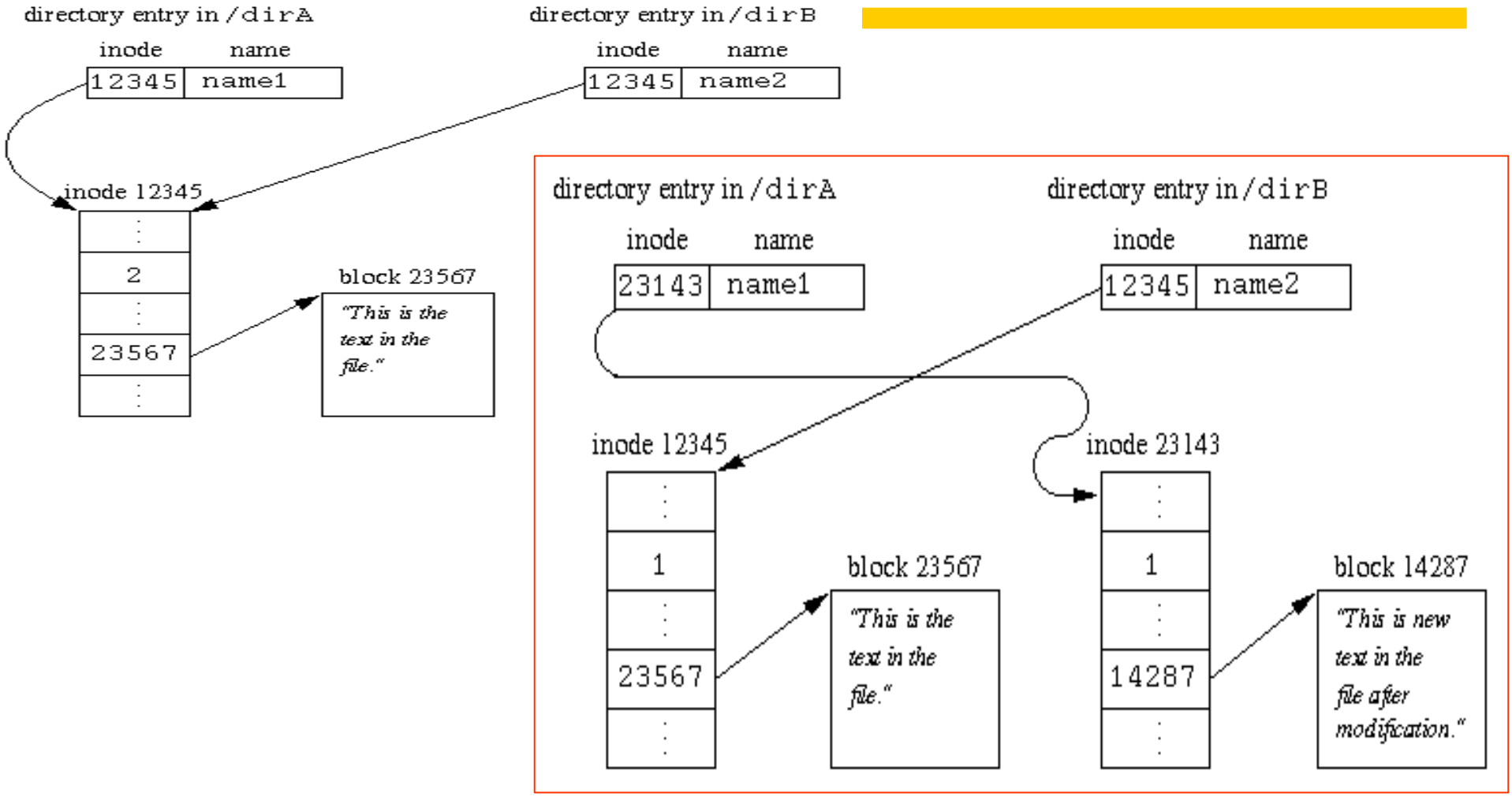


Figure 5.6 (page 167): The situation after editing the file. A new file with new inode is created;

Symbolic Links

- Symbolic link
  - **special type of file** that contains the name of another file
- A reference to the name of a symbolic link
  - OS use the name stored in the file; not the name itself.
- Create a symbolic link
  - `ln -s /dirA/name1 /dirB/name2`
- Symbolic links do not affect link count in the inode
- symbolic links can span filesystems (while hard links cannot)

Example: A Symbolic Link

```
if (symlink("/dirA/name1", "/dirB/name2") == -1)…
```

```
> ln -s /dirA/name1 /dirB/name2
```

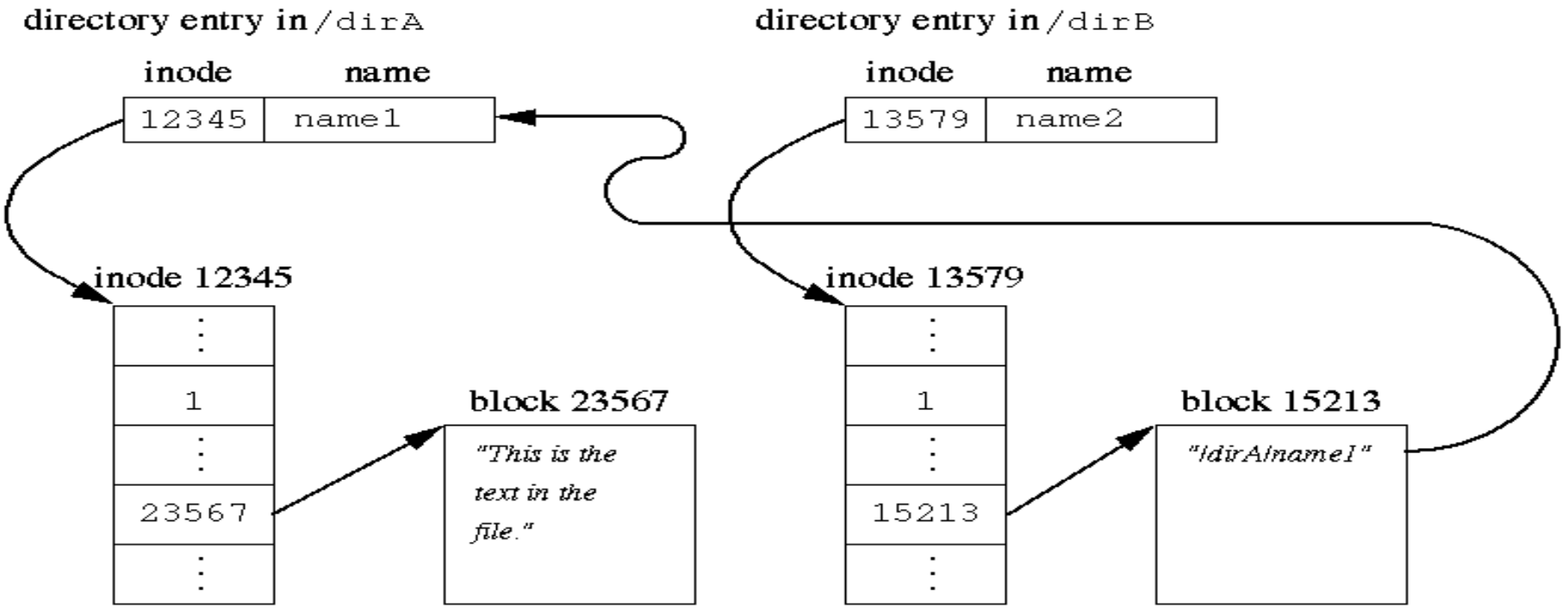
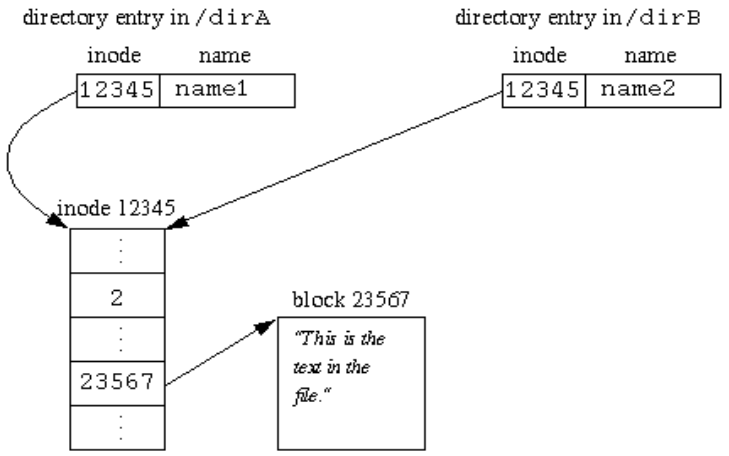


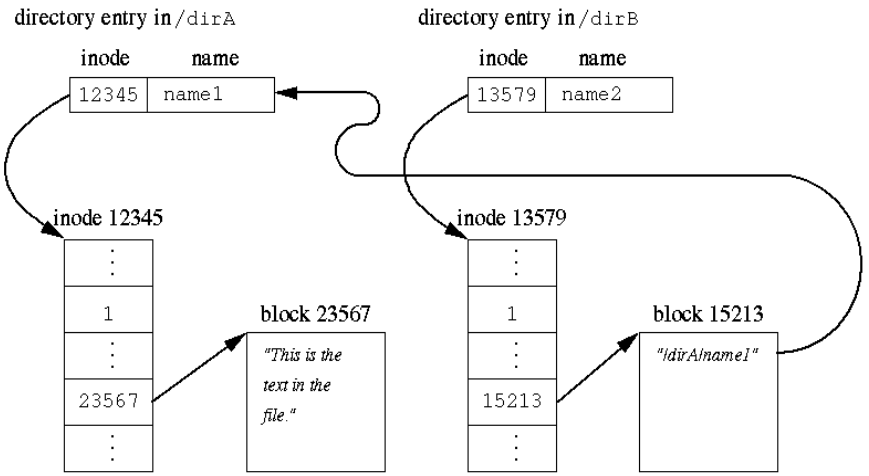
Figure 5.8 (page 170): An ordinary file with a symbolic link to it.

Hard vs. Symbolic Links

- Hard links
  - Hard links are relative to a given filesystem
  - References a physical file (unless the filesystem is corrupt)
  - A count of the hard links is kept in the inode
  - Removing the last hard link frees the physical file



- Symbolic links
  - Can make a symbolic link to a file that do not exist
  - Even if it did exist once, a symbolic link might not reference anything
  - Removing a symbolic link cannot free a physical file



Why do we need hard links and soft links?

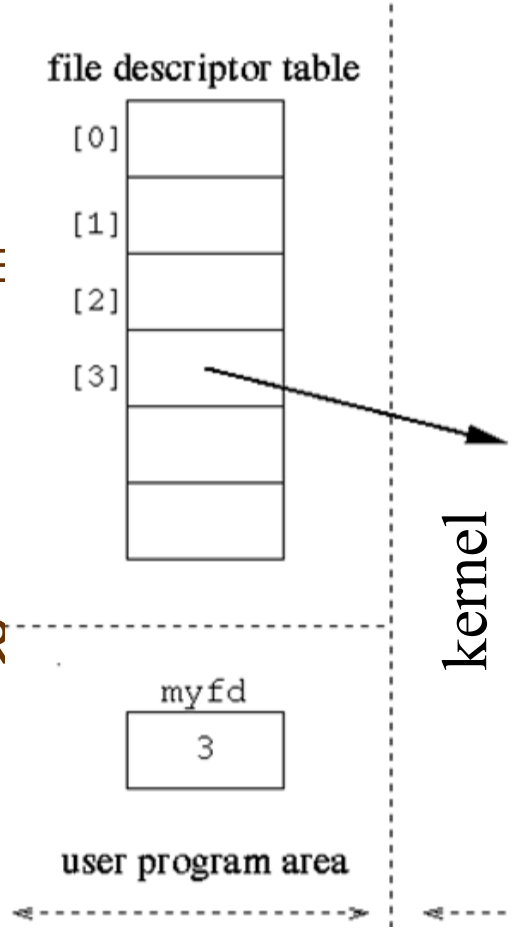
- Soft link
  - Short and easy way to access a file located somewhere else
- Hard link
  - Have one copy of a file with different names under different directories
  - Grouping your files under different categories
  - cp, mv and rm may all be the same executable
  - bunzip2, bzcat and bzip all use the same inode
  - create file-based locks, the link(2) system call is atomic.
  - How about implementing a recycle bin?

Outline

- Basics of File Systems
- Directory and Unix File System:
  - inodes
  - Directory operations
  - Links of Files: Hard vs. Symbolic
- UNIX I/O System Calls: open, close, read, write, ioctl
- File Representations:
  - FDT, SFT, inode table
  - Fork and inheritance,
  - Filters and redirection
- FILE pointers (fopen) and buffering

Unix I/O Related System Calls

- Use **file descriptors,** e.g., `int myfd;`
- 3 file descriptors open when a program starts: STDIN_FILENO, STDOUT_FILENO, STDERR_FILE
- 5 main system calls for I/O
  - **`open, close, read, write, ioctl`**
  - Return -1 on error and set **`errno`**
  - a return value of -1 with **`errno`** set to EINTR not usually an error.
- Device independence: uniform device interface
- I/O through device drivers with standard interface



open

```
#include <fcntl.h>
#include <sys/stat.h>
int open(const char *path, int oflag);
```

```
        O_RDONLY:       read only
        O_WRONLY:       write only
        O_RDWR:         read and write
        O_APPEND:       writes always write to end
        O_CREAT:        create the file if it does not exist
        O_EXCL:         used with O_CREAT, return an error if file exists
        O_NOCTTY:       do not become a controlling terminal
        O_NONBLOCK:     do not block if not ready to open, affects reads and w
        O_TRUNC:        discard previous contents
```

```
// if O_CREAT flag is used
int open(const char *path, int oflag,
                           mode_t mode);
```

open (cont.)

- O_CREAT flag: must use the 3-parameter form of open and specify mode for permissions

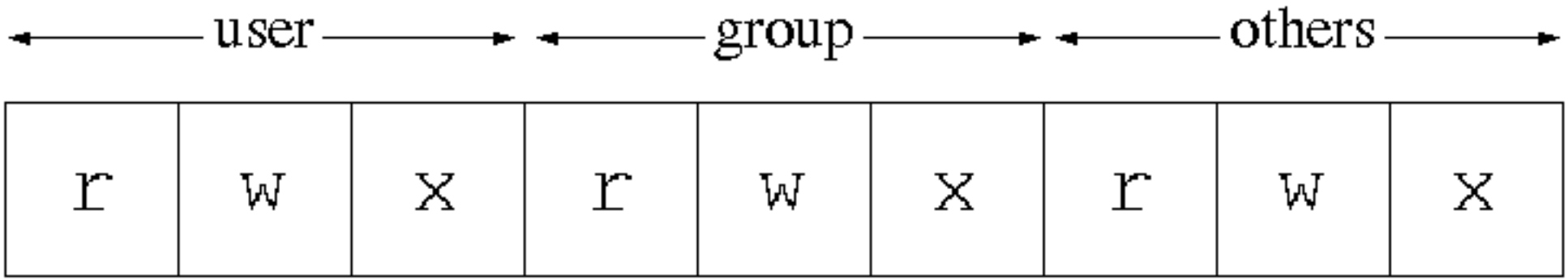


Figure 4.10 (page 105): Historical layout of the permissions mask.

Ref

POSIX Symbolic Names for Permissions (mode)

defined in sys/stat.h

S_IRUSR	read permission bit for owner
S_IWUSR	write permission bit for owner
S_IXUSR	execute permission bit for owner
S_IRWXU	read, write, execute for owner
S_IRGRP	read permission bit for group
S_IWGRP	write permission bit for group
S_IXGRP	execute permission bit for group
S_IRWXG	read, write, execute for group
S_IROTH	read permission bit for others
S_IWOTH	write permission bit for others
S_IXOTH	execute permission bit for others
S_IRWXO	read, write, execute for others
S_ISUID	set user ID on execution
S_ISGID	set group ID on execution

Program 4.9: copyfilemain.c, page 106

copy a file.

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include "restart.h"

#define READ_FLAGS O_RDONLY
#define WRITE_FLAGS (O_WRONLY | O_CREAT | O_EXCL)
#define WRITE_PERMS (S_IRUSR | S_IWUSR)

int main(int argc, char *argv[]) {
    int bytes;
    int fromfd, tofd;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s from_file to_file\n", argv[0]);
        return 1;
    }

    if ((fromfd = open(argv[1], READ_FLAGS)) == -1) {
        perror("Failed to open input file");
        return 1;
    }

    if ((tofd = open(argv[2], WRITE_FLAGS, WRITE_PERMS)) == -1) {
        perror("Failed to create output file");
        return 1;
    }

    bytes = copyfile(fromfd, tofd);
    printf("%d bytes copied from %s to %s\n", bytes, argv[1], argv[2]);
    return 0;                                   /* the return closes the files */
}
```

Close files!

close and its usage

```
#include <unistd.h>
int close(int fildes);
```

- Open files are closed when program exits normally.

Program 4.10: r_close.c, page 107

```
#include <errno.h>
#include <unistd.h>

int r_close(int fd) {
    int retval;

    while (retval = close(fd), retval == -1 && errno == EINTR) ;
    return retval;
}
```

System Call: read

```
#include <unistd.h>
ssize_t read(int fildes, void *buf,
                         size_t nbyte);
```

- Need to allocate space for buf to hold the bytes read;
- **size_t** is an unsigned long type;
- **ssize_t** is a signed long type;
- **Can return fewer bytes than requested**;
- Return value of -1 with **errno** set to EINTR is not usually an error.

What is wrong here…

```
char *buf;
ssize_t bytesread;
bytesread = read(STDIN_FILENO, buf, 100);
```

- Can the following fix it?

```
char buf[100];
ssize_t bytesread;
bytesread = read(STDIN_FILENO, buf, 100);
```

Error checking!

Program 4.1: readline.c, page 95

An example to read in a line

```
#include <errno.h>
#include <unistd.h>

int readline(int fd, char *buf, int nbytes) {
    int numread = 0;
    int returnval;

    while (numread < nbytes - 1) {
        returnval = read(fd, buf + numread, 1);
        if ((returnval == -1) && (errno == EINTR))
            continue;
        if ( (returnval == 0) && (numread == 0) )
            return 0;
        if (returnval == 0)
            break;
        if (returnval == -1)
            return -1;
        numread++;
        if (buf[numread-1] == '\n') {
            buf[numread] = '\0';
            return numread;
        }
    }
    errno = EINVAL;
    return -1;
}
```

Read one char at a time!

System Call: write

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf,
                          size_t nbyte);
```

- Not error if return value > 0 but **less than `nbyte`**
  - Must restart write if it returns fewer bytes than requested
- Return value of -1 with **`errno`** set to EINTR is not usually an error

Program 4.2: copyfile1.c, page 98

```
#include <errno.h>
#include <unistd.h>
#define BLKSIZE 1024

int copyfile(int fromfd, int tofd) {
   char *bp;
   char buf[BLKSIZE];
   int bytesread;
   int byteswritten = 0;
   int totalbytes = 0;

   for (  ;  ;  ) {
      while (((bytesread = read(fromfd, buf, BLKSIZE)) == -1) &&
             (errno == EINTR)) ;          /* handle interruption by signal */
      if (bytesread <= 0)             /* real error or end-of-file on fromfd */
         break;
      bp = buf;
      while (bytesread > 0) {
         while(( byteswritten = write(tofd, bp, bytesread)) == -1 ) &&
               (errno == EINTR)) ;        /* handle interruption by signal */
         if (byteswritten < 0)                          /* real error on tofd */
            break;
         totalbytes += byteswritten;
         bytesread -= byteswritten;
         bp += byteswritten;
      }
      if (byteswritten == -1)                           /* real error on tofd */
          break;
   }
   return totalbytes;
}
```

Function *copyfile* reads from one file and writes out to another

```c
#include <errno.h>
#include <unistd.h>

ssize_t r_read(int fd, void *buf, size_t size) {
  ssize_t retval;
  while (retval = read(fd, buf, size),
         retval == -1 && errno == EINTR) ;
  return retval;
}
```

```c
ssize_t r_write(int fd, void *buf, size_t size) {
  char *bufp;
  size_t bytestowrite;
  ssize_t byteswritten;
  size_t totalbytes;

  for(bufp = buf, bytestowrite = size, totalbytes = 0;
      bytestowrite > 0;
      bufp += byteswritten, bytestowrite -= byteswritten) {
    byteswritten = write(fd, bufp, bytestowrite);
    if ((byteswritten) == -1 && (errno != EINTR)) return -1;
    if (byteswritten == -1) byteswritten = 0;
    totalbytes += byteswritten;
  }
  return totalbytes;
}
```

Ref

File control: fcntl

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
```

```
int fcntl(int fildes, int cmd, /*arg*/...);
```

- The `fcntl` function may take additional parameters depending on the value of `cmd`

cmd	meaning
F_DUPFD	duplicate a file descriptor
F_GETFD	get file descriptor flags
F_SETFD	set file descriptor flags
F_GETFL	get file status flags and access modes
F_SETFL	set file status flags and access modes
F_GETOWN	if fildes is a socket, get process or group ID for out-of-band signals
F_SETOWN	if fildes is a socket, set process or group ID for out-of-band signals
F_GETLK	get first lock that blocks description specified by arg
F_SETLK	set or clear segment lock specified by arg
F_SETLKW	same as FSETLK except it blocks until request satisfied

Ref

File control: fcntl

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
```

- Example: How to set/unset an already opened file descriptor fd for nonblocking I/O.

```
int setnonblock(int fd) {
  int fdflags;
  if ((fdflags=fcntl(fd, F_GETFL, 0))==-1)
    return -1;

  fdflags |= O_NONBLOCK;
```

```
fdflags &= ~O_NONBLOCK;
```

```
  if (fcntl(fd, F_SETFL, fdflags) == -1)
     return -1;
return 0;
}
```

Exercise : Read Example

- Suppose the file **infile** contains "abcdefghijklmnop" Assuming no errors occur, **what are the possible outputs of the following program**?

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fd;
    char buf[5] = "WXYZ";
    fd = open("infile",O_RDONLY);
    read(fd, buf,2);
    read(fd, buf+2,2);
    close(fd);
    printf("%c%c%c%c\n",buf[0],buf[1],buf[2],buf[3]);
    return 0;
}
```

Example 4.10: Read a pair of integers…

```
struct {
    int x;
    int y;
} point;
if (read(fd, &point, sizeof
    fprintf(stderr, "Cannot
if (readblock(fd, &point, s
    fprintf(stderr, "Cannot
```

```
#include <errno.h>
#include <unistd.h>

ssize_t readblock(int fd, void *buf, size_t size) {
   char *bufp;
   size_t bytestoread;
   ssize_t bytesread;
   size_t totalbytes;

   for (bufp = buf, bytestoread = size, totalbytes = 0;
        bytestoread > 0;
        bufp += bytesread, bytestoread -= bytesread) {
      bytesread = read(fd, bufp, bytestoread);
      if ((bytesread == 0) && (totalbytes == 0))
         return 0;
      if (bytesread == 0) {
         errno = EINVAL;
         return -1;
      }
      if ((bytesread) == -1 && (errno != EINTR))
         return -1;
      if (bytesread == -1)
         bytesread = 0;
      totalbytes += bytesread;
   }
   return totalbytes;
}
```

Exercise: Binary vs. Text file

```
int num=54321;     char str[8] = "12345";
```

- What will we see in the file after the following writes?

```
write(fdw1, &num, sizeof(int)); // A
  vs.
write(fdw2, str, sizeof(int));  // B
```

- How about the following reads for each write?

```
read(fdr1, &num, sizeof(int));  // X
  vs.
read(fdr2, str, sizeof(int));  // Y
```

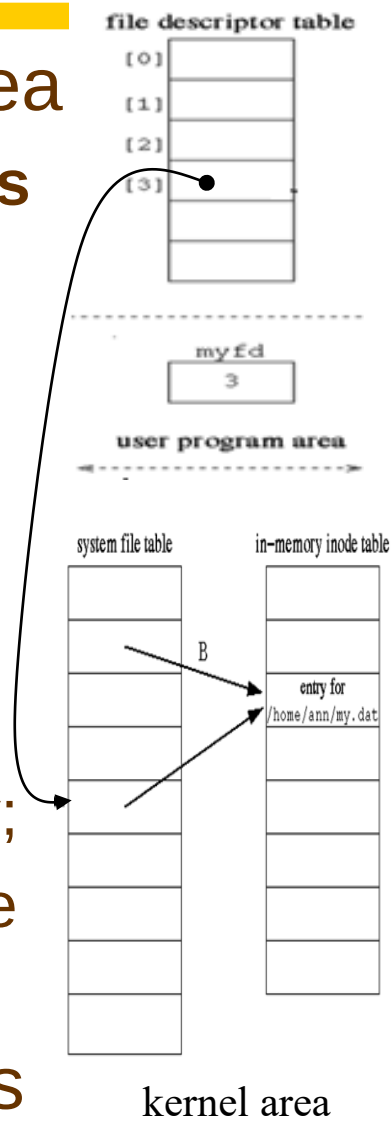
write/ read	X	Y
A		
B		

Outline

- Basics of File Systems
- Directory and Unix File System:
  - inodes
  - Directory operations
  - Links of Files: Hard vs. Symbolic
- UNIX I/O System Calls: open, close, read, write, ioctl
- File Representations:
  - FDT, SFT, inode table
  - Fork and inheritance,
  - Filters and redirection
- FILE pointers (fopen) and buffering

File Representation

- **File Descriptor Table (FDT):** user program area
  - An array of pointers **indexed by the file descriptors**
  - The pointers in FDT point to entries in SFT
- **System File Table (SFT):** kernel area
  - Contains entries for **each open file**
  - Entries contain pointers to a table of inodes kept in memory
  - Other information in an entry: **current file offset**; **count of file descriptors that are using this entry**;
  - When a file is closed, the count is decremented. The entry is freed when the count becomes 0.
- **In-Memory Inode Table**: copies of used inodes



File Representation (cont.)

```
myfd = open("/home/ann/my.dat", O_RDONLY);
```

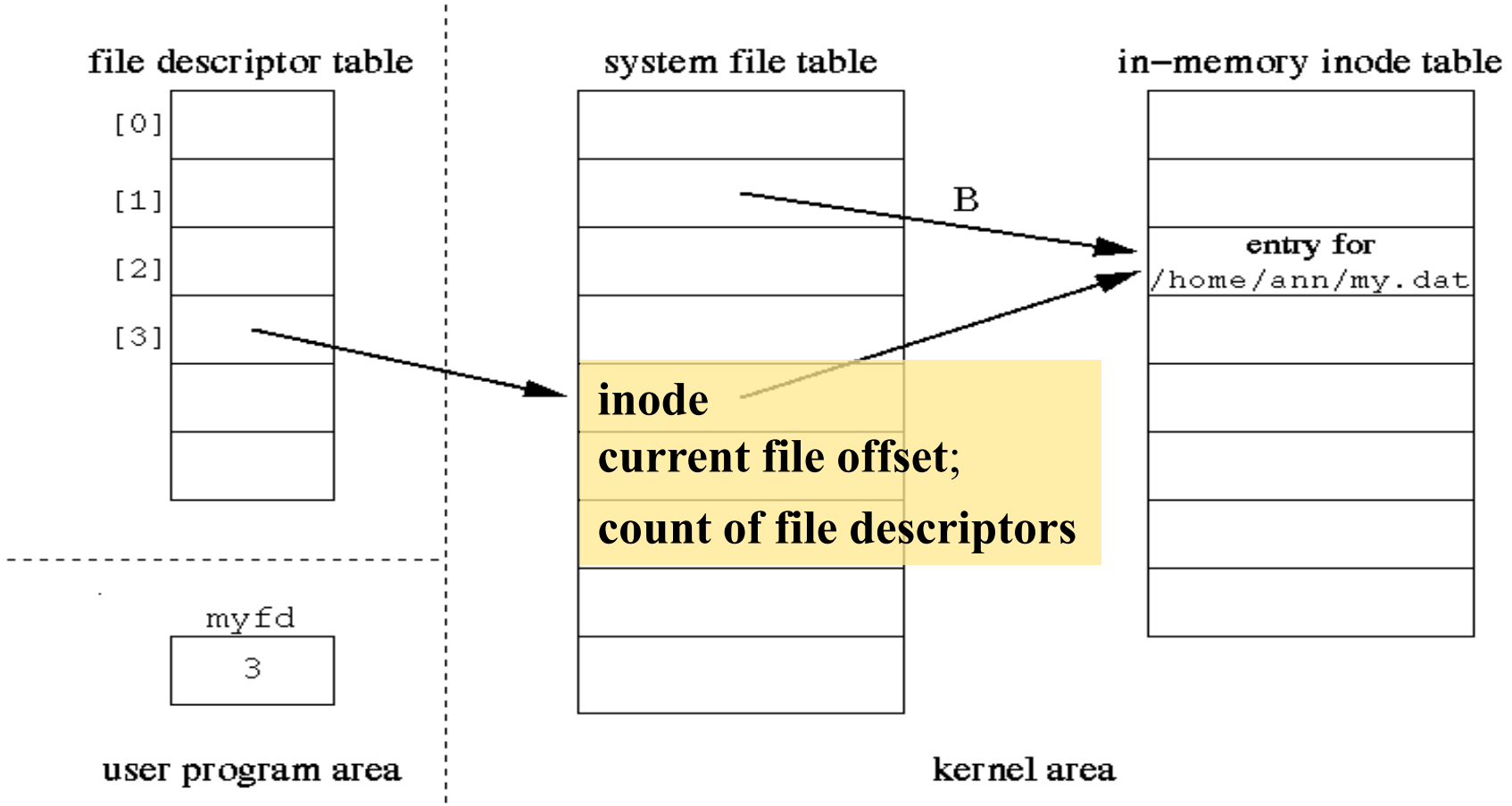
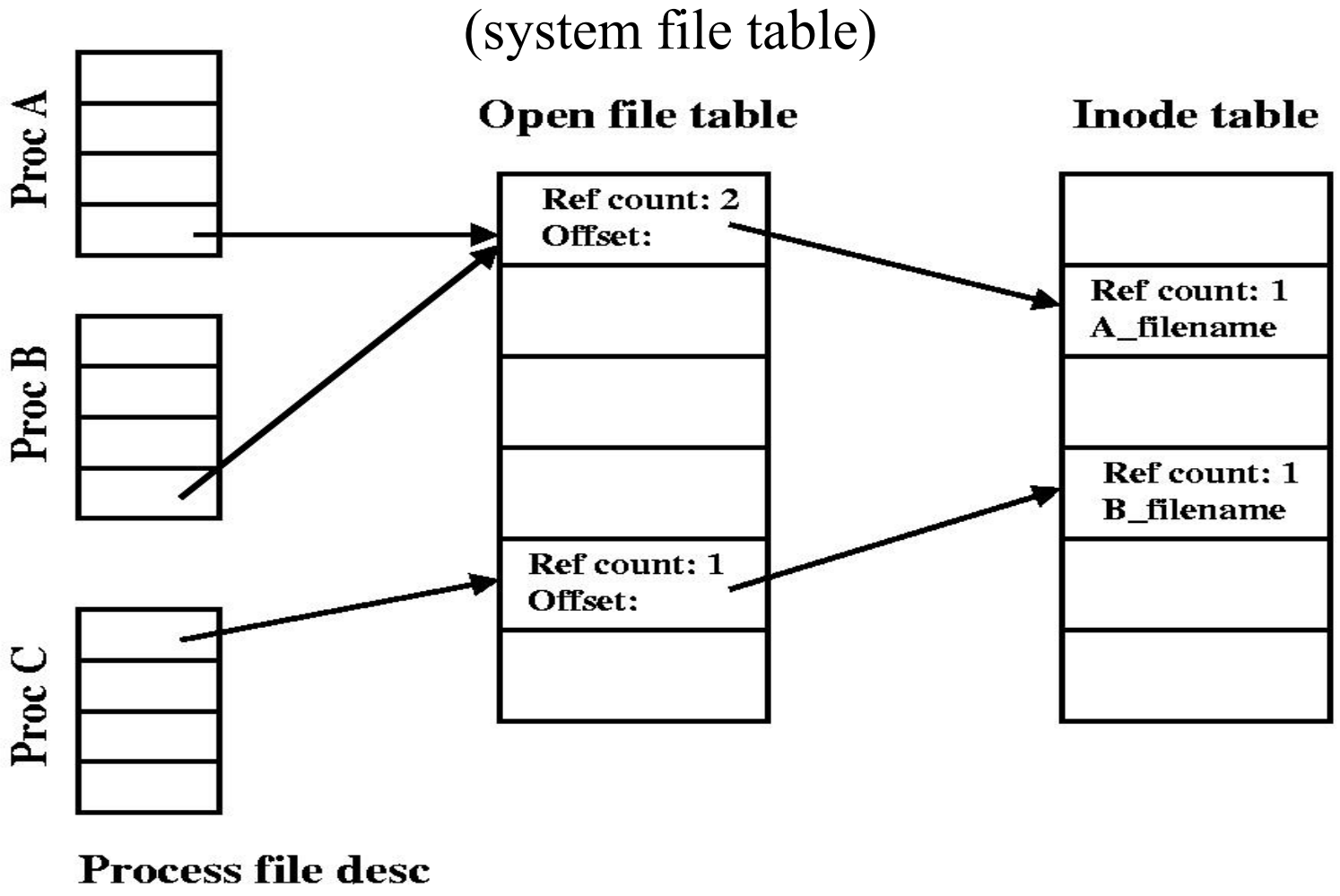


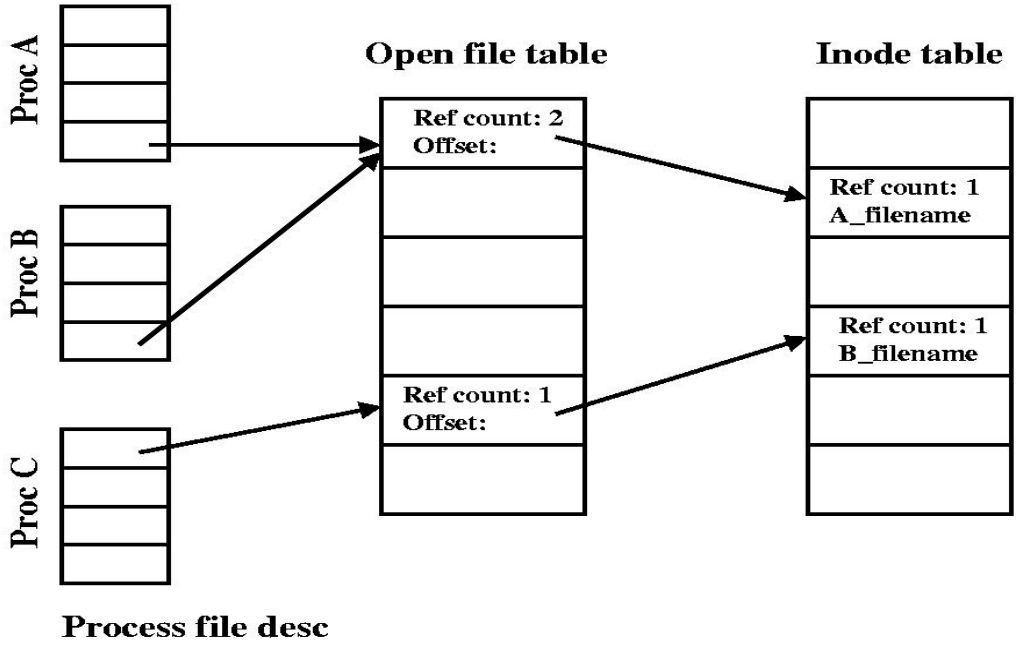
Figure 4.2 (page 120): Relationship between the file descriptor table, the system file table and the in-memory inode table.

File Representation (cont.)



Quiz about file-related tables

- If a file is opened twice (say in Proc C),
  - How many entries will be created in file descriptor table?
  - How many will be created in open file table (system file table)?
  - How many in inode table?



Inheritance of File Descriptors

When fork creates a child, what happens to FDT, SFT, INODE?

- child inherits a copy of the parent's address space
- including the file descriptor table

Example 4.27: openfork.c, page 124

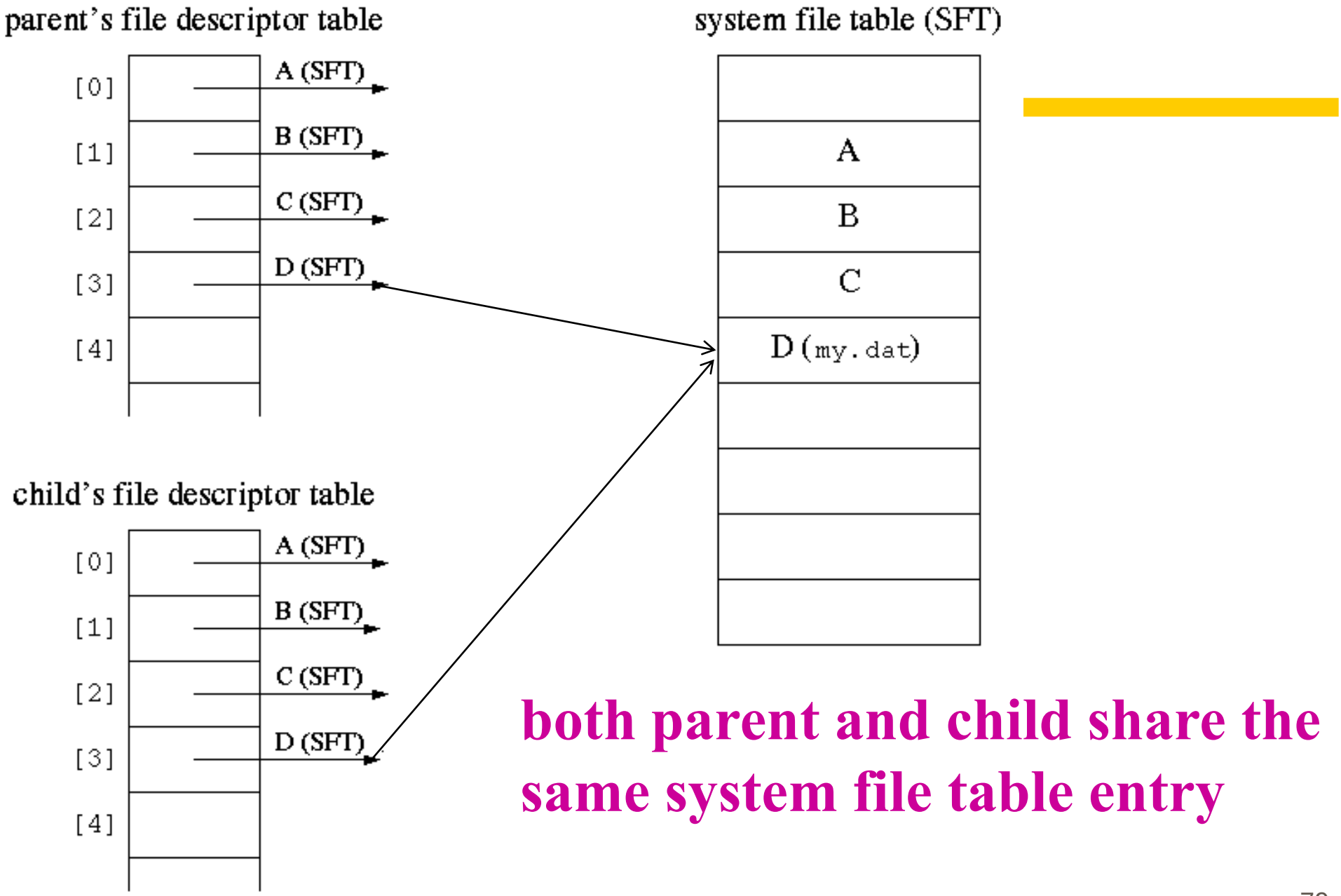
```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>

int main(void) {
    char c = '!';
    int myfd;

    if ((myfd = open("my.dat", O_RDONLY)) == -1) {
        perror("Failed to open file");
        return 1;
    }
    if (fork() == -1) {
        perror("Failed to fork");
        return 1;
    }
    read(myfd, &c, 1);
    printf("Process %ld got %c\n", (long)getpid(), c);
    return 0;
}
```

File is opened BEFORE fork

So, parent and child use the same system file table entries

Suppose my.dat contains: `xyz`

What will each process read?



both parent and child share the same system file table entry

Example 4.31: forkopen.c, pages 125-126

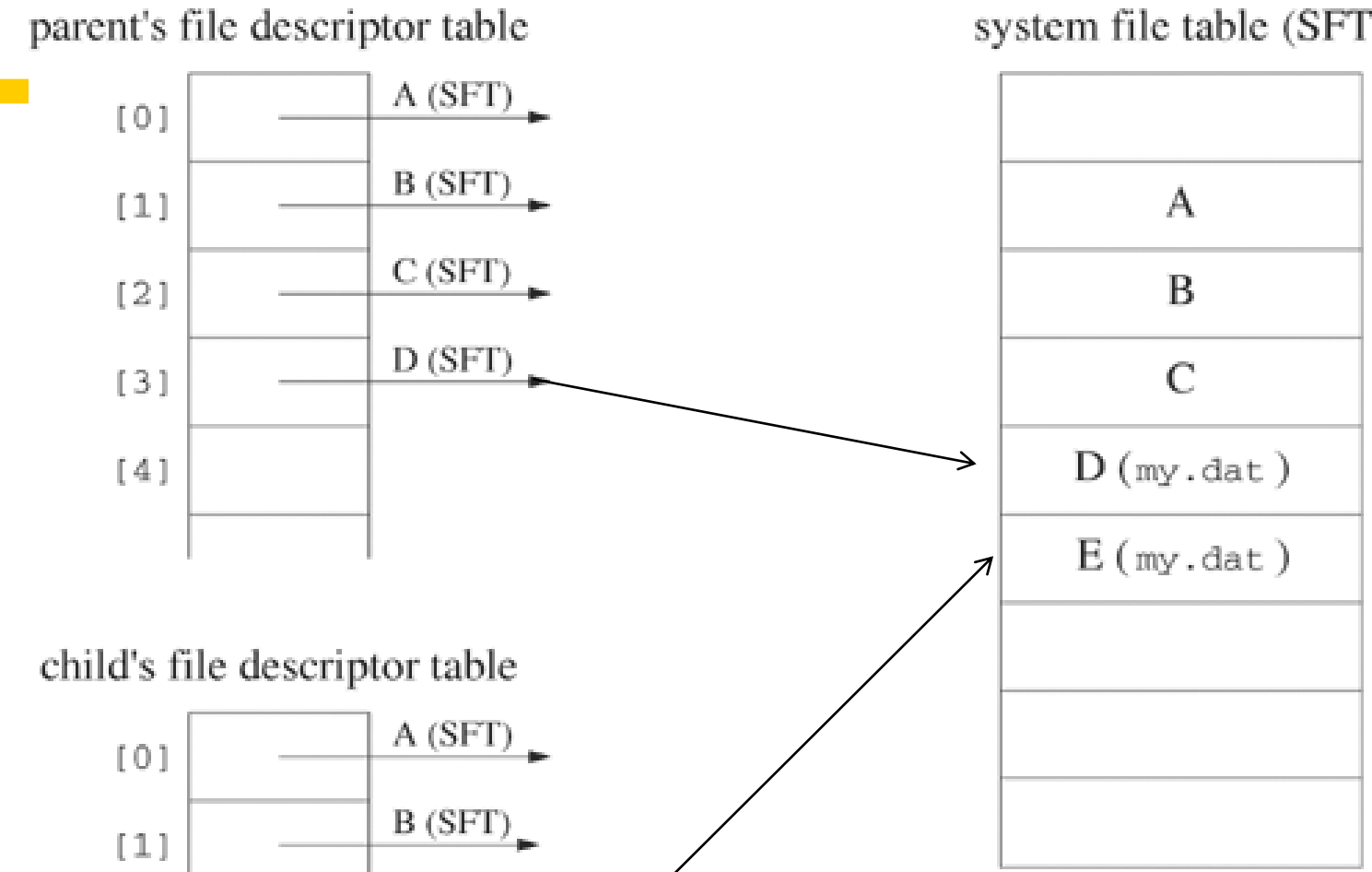
```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>

int main(void) {
    char c = '!';
    int myfd;

    if (fork() == -1) {
        perror("Failed to fork");
        return 1;
    }
    if ((myfd = open("my.dat", O_RDONLY)) == -1) {
        perror("Failed to open file");
        return 1;
    }
    read(myfd, &c, 1);
    printf("Process %ld got %c\n", (long)getpid(), c);
    return 0;
}
```

File is opened AFTER fork

So, parent and child use different system file table entries

Again suppose my.dat contains: `xyz`

What will each process read?


parent's file descriptor table
[0]
[1]
[2]
[3]
[4]
A (SFT)
B (SFT)
C (SFT)
D (SFT)
system file table (SFT)
A
B
C
D (my.dat)
E (my.dat)
child's file descriptor table
[0]
[1]
[2]
[3]
[4]
A (SFT)
B (SFT)
C (SFT)
E (SFT)

Quiz: Read Example

```
int main(void) {
  char c = '!';
  char d = '!';
  int myfd;
  if (fork() == -1) {
     return 1;
  }
  if((myfd = open("my.dat", O_RDONLY))==-1){
     return 1;
  }

  read(myfd, &c, 1);
  read(myfd, &d, 1);
  printf("Process %ld got %c\n", (long)getpid(), c);
  printf("Process %ld got %c\n", (long)getpid(), d);
  return 0;
}
```

- Suppose the file **my.dat** contains "abcdefghijklmnop" and no errors occur; what will be the possible outputs?

Process 17514 got a
Process 17514 got b
Process 17515 got a
Process 17515 got b

Process 17515 got a
Process 17515 got b
Process 17514 got a
Process 17514 got b

Process 17515 got a
Process 17514 got a
Process 17515 got b
Process 17514 got b

Quiz: Read Example (open/fork are switched)

```
int main(void) {
  char c = '!';
  char d = '!';
  int myfd;
  if ((myfd = open("my.dat", O_RDONLY)) == -1) {
    return 1;
  }

  if (fork() == -1) {
    return 1;
  }

  read(myfd, &c, 1);
  read(myfd, &d, 1);
  printf("Process %ld got %c\n", (long)getpid(), c);
  printf("Process %ld got %c\n", (long)getpid(), d);
  return 0;
}
```

- Suppose the file **my.dat** contains "abcdefghijklmnop" and no errors occur; what will be the possible outputs?

Process 17514 got a
Process 17514 got b
Process 17515 got c
Process 17515 got d

Many possibilities on the order

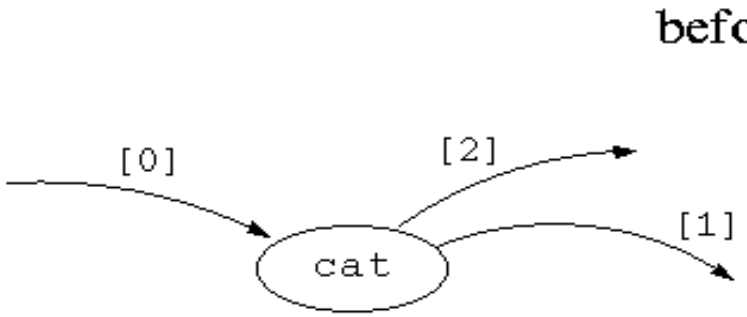
Process 17514 got a
Process 17515 got b
Process 17514 got c
Process 17515 got d

Filters and Redirection

A program can modify the file descriptor table entry so that it points to a different entry in the system file table. This action is known as redirection.

Redirection

- Example 4.35 (p129)
  - cat



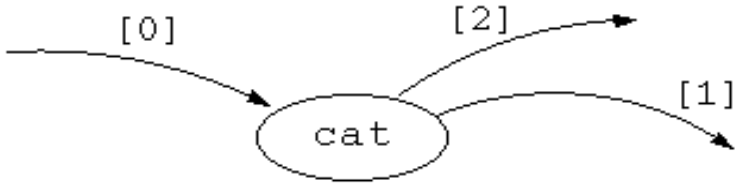
before redirection

file descriptor table

[0]	*standard input*
[1]	*standard output*
[2]	*standard error*

How about

- cat > my.file



to `my.file`

after redirection

file descriptor table

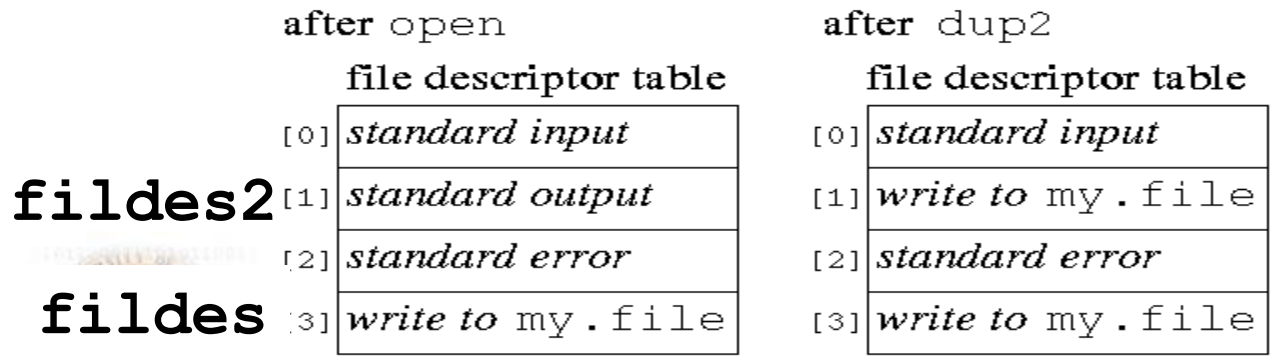
[0]	*standard input*
[1]	*write to* `my.file`
[2]	*standard error*

Redirection

- Copy one **file descriptor table** entry into another
- Can be done with **`dup2`** system call

```
#include <unistd.h>
int dup2(int fildes, int fildes2);
```

  - First close fildes2 silently if fildes2 exists
  - Then, copy the pointer of entry fildes into entry fildes2



Program 4.18: redirect.c, page 131

Redirection example

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include "restart.h"
#define CREATE_FLAGS (O_WRONLY | O_CREAT | O_APPEND)
#define CREATE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

int main(void) {
    int fd;

    fd = open("my.file", CREATE_FLAGS, CREATE_MODE);
    if (fd == -1) {
        perror("Failed to open my.file");
        return 1;
    }
    if (dup2(fd, STDOUT_FILENO) == -1) {
        perror("Failed to redirect standard output");
        return 1;
    }
    if (r_close(fd) == -1) {
        perror("Failed to close the file");
        return 1;
    }
    if (write(STDOUT_FILENO, "OK", 2) == -1) {
        perror("Failed in writing to file");
        return 1;
    }
    return 0;
}
```

FDT for Redirection Example

after `open`

file descriptor table

[0]	*standard input*
[1]	*standard output*
[2]	*standard error*
[3]	*write to* `my.file`

after `dup2`

file descriptor table

[0]	*standard input*
[1]	*write to* `my.file`
[2]	*standard error*
[3]	*write to* `my.file`

after `close`

file descriptor table

[0]	*standard input*
[1]	*write to* `my.file`
[2]	*standard error*

Figure 4.7 (page 131): The status of the file descriptor table during the execution of Program 4.18.

Outline

- Basics of File Systems
- Directory and Unix File System:
  - inodes
  - Directory operations
  - Links of Files: Hard vs. Symbolic
- UNIX I/O System Calls: open, close, read, write, ioctl
- File Representations:
  - FDT, SFT, inode table
  - Fork and inheritance,
  - Filters and redirection
- FILE pointers (fopen) and buffering

File pointers and buffering

FILE pointers vs. File Descriptors?

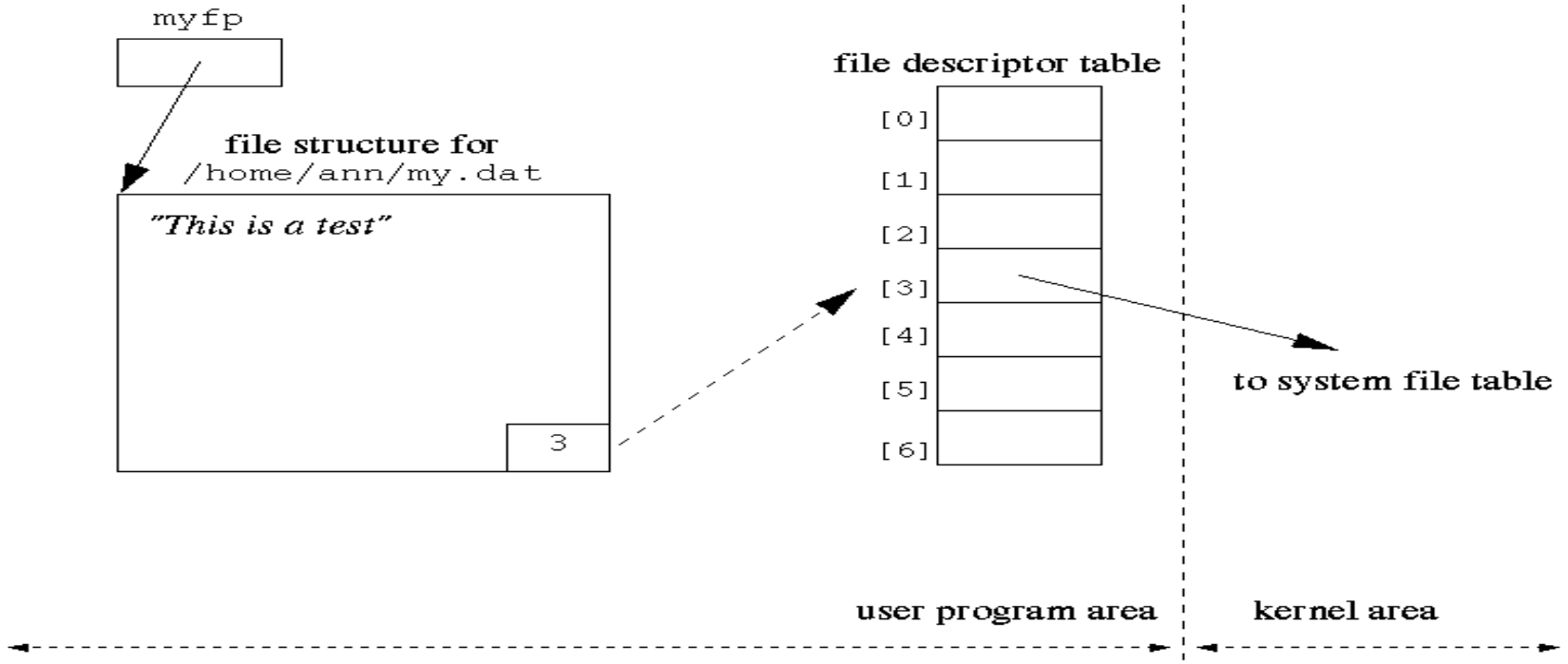
fopen, fclose, fread, fwrite, fprintf, fscanf,

vs.

open, close, read, write

Standard IO functions vs. IO sys calls

```
FILE *myfp;
if ((myfp = fopen("/home/ann/my.dat", "w")) == NULL)
    perror("Failed to open /home/ann/my.dat");
else
    fprintf(myfp, "This is a test");
```



File Pointers and Buffering

- I/O using file pointers → fread/fwrite from/to **buffer**;
- Buffer is filled or emptied when necessary
- Buffer size may vary
- **fwrite may fill part of the buffer without causing any physical I/O to the file**;
- If a fwrite is done to **standard output**, and program crashes, data written may not show up on screen
- **Standard error is NOT buffered**
- Interleaving output to standard output and error → output to appear in an unpredictable order
- Force the physical output to occur with **fflush(…)**

Exercise: bufferout.c

Exercise 4.25: bufferout.c, page 123

```
#include <stdio.h>

int main(void) {
    fprintf(stdout, "a");
    fprintf(stderr, "a has been written\n");
    fprintf(stdout, "b");
    fprintf(stderr, "b has been written\n");
    fprintf(stdout, "\n");
    return 0;
}
```

```
a has been written
b has been written
ab
```

How about switching std error add std out file descriptors? So printf("a"); will print right a way!

Exercise: bufferinout.c

Exercise 4.26: bufferinout.c, page 123

```
#include <stdio.h>

int main(void) {
    int i;
    fprintf(stdout, "a");
    scanf("%d", &i);
    fprintf(stderr, "a has been written\n");
    fprintf(stdout, "b");
    fprintf(stderr, "b has been written\n");
    fprintf(stdout, "\n");
    return 0;
}
```

```
a
a has been written
b has been written
b
```

Fileiofork.c

- What is the output?

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
  printf("This is my output.");
  fork();
  return 0;
}
```

```
This is my output.This is my output.
```

Fileioforkline.c

- What is the output?

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
  printf("This is my output.\n");
  fork();
  return 0;
}
```

```
This is my output.
```

The buffering of standard output is usually **line buffering**.
This means that the buffer is flushed when it contains a newline.

Other issues…

How to read from two or more files, IO devices…

Program 4.11: monitorfork.c, page 108

Read from two files with two processes

How about monitoring both files in the same process?

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "restart.h"

int main(int argc, char *argv[]) {
    int bytesread;
    int childpid;
    int fd, fd1, fd2;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s file1 file2\n", argv[0]);
        return 1;
    }
    if ((fd1 = open(argv[1], O_RDONLY)) == -1) {
        fprintf(stderr, "Failed to open file %s:%s\n", argv[1], strerror(errno));
        return 1;
    }
    if ((fd2 = open(argv[2], O_RDONLY)) == -1) {
        fprintf(stderr, "Failed to open file %s:%s\n", argv[2], strerror(errno));
        return 1;
    }
    if ((childpid = fork()) == -1) {
        perror("Failed to create child process");
        return 1;
    }
    if (childpid > 0)                                   /* parent code */
        fd = fd1;
    else
        fd = fd2;
    bytesread = copyfile(fd, STDOUT_FILENO);
    fprintf(stderr, "Bytes read: %d\n", bytesread);
    return 0;
}
```

select - monitor multiple file descriptors

```
#include <sys/select.h>
int select(int nfds,
           fd_set *restrict readfds,
           fd_set *restrict writefds,
           fd_set *restrict errorfds,
           struct   timeval *restrict timeout);

void FD_CLR(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

Program 4.12: whichisready.c, page 110

Return the ready file

```
#include <errno.h>
#include <string.h>
#include <sys/select.h>

int whichisready(int fd1, int fd2) {
    int maxfd;
    int nfds;
    fd_set readset;

    if ((fd1 < 0) || (fd1 >= FD_SETSIZE) ||
        (fd2 < 0) || (fd2 >= FD_SETSIZE)) {
        errno = EINVAL;
        return -1;
    }
    maxfd = (fd1 > fd2) ? fd1 : fd2;
    FD_ZERO(&readset);
    FD_SET(fd1, &readset);
    FD_SET(fd2, &readset);
    nfds = select(maxfd+1, &readset, NULL, NULL, NULL);
    if (nfds == -1)
        return -1;
    if (FD_ISSET(fd1, &readset))
        return fd1;
    if (FD_ISSET(fd2, &readset))
        return fd2;
    errno = EINVAL;
    return -1;
}
```

Summary

- Basics of File Systems
- Directory and Unix File System:
  - inodes
  - Directory operations
  - Links of Files: Hard vs. Symbolic
- UNIX I/O System Calls: open, close, read, write, ioctl
- File Representations:
  - FDT, SFT, inode table
  - Fork and inheritance,
  - Filters and redirection
- FILE pointers (fopen) and buffering