系統程式設計

施吉昇
臺灣大學資訊工程系

Agenda

- Properties of Files and Directories
- File systems and Directory Structures
- Special access permission
- Ownership of new files

Properties of Files and Directories

Files in POSIX and Unix Systems

- Resources in operating systems can be classified into
  - computable resources: process, memory, etc.
  - non-computable resources: storage systems, hardware devices, communication links, etc.
- File is a general concept to mange non-computable resources.
  - Storage systems:
    - regular files, directories, etc.
  - Devices: keyboard, pointer devices, USB, etc.
  - /proc/*: interfaces to communicate with kernel
  - /etc/sockets: interface to communicate with processes (on different machines.)

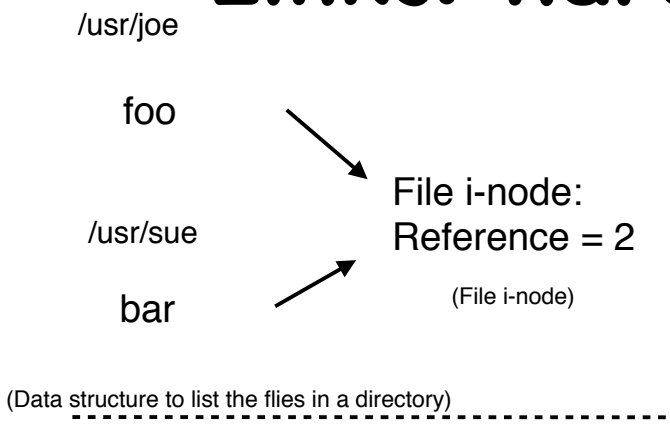
File Types

- Regular Files: text, binary, etc.
- Directory Files: Only Kernel can update these files – { (filename, pointer) }.
- Character Special Files, e.g., tty, audio, etc.
- Block Special Files, e.g., disks, etc.
- FIFO – named pipes
- Sockets – not POSIX.1 or SVR4
  - SVR4 uses library of socket functions, instead. 4.3+BSD has a file type of socket.
- Symbolic Links – not POSIX.1 or SVR4

Link

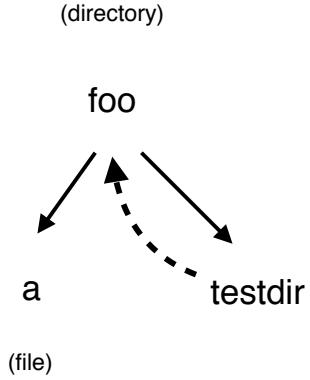
- Files are identified by i-nodes in the file systems.
  - It is a long string, usually represented by a long address, to be read by the operating system.
  - I-node is not designed for human users.
- Directories and filename are widely used by librarian/human users to organize large volumes of documents, long before there are computers.
  - Link is the pointer to bridge legible file name/ directory name to i-node.
  - One i-node can have more than one name, each of which is represented by one link.

Links: hard and soft



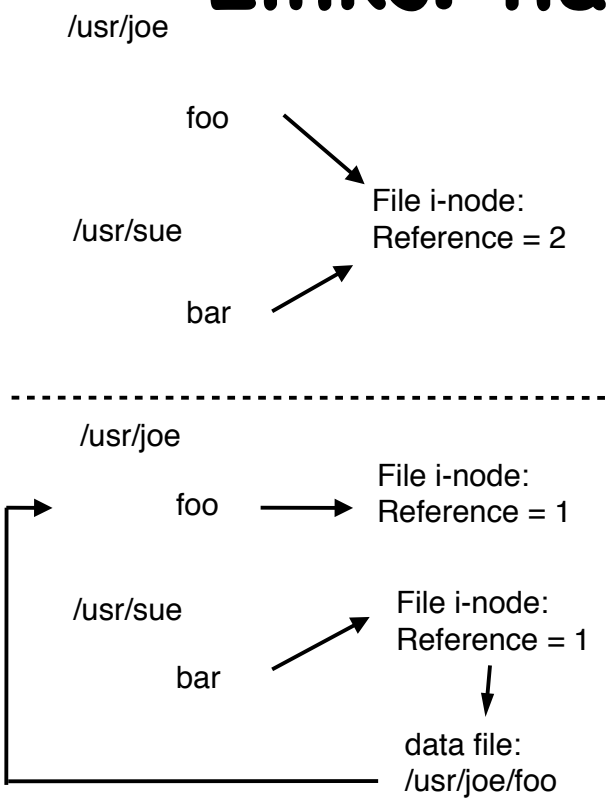
- (Hard) Link
  - Each directory entry creates a hard link of a filename to the i-node that describes the file's contents.

Symbolic Links (soft link)



- Initially introduced by 4.2BSD
- Provide a pointer to a file/directory, similar to the concept of alias.
  - It has a dedicated file type: symbolic link
  - It is represented as a regular file in the system.
  - The content of the file is the path of the targeted file/directory.
  - The existence of the targeted file/directory is NOT validated by the file system and the link may
    - form a loop or
    - point to an invalid path for file/directory.

Links: hard and soft



- Hard Link
  - Each directory entry creates a hard link of a filename to the i-node that describes the file's contents.
- Symbolic Link (Soft Link)
  - It is implemented as a file that contains a pathname.
  - Filesize = pathname length
  - Example: Shortcut on Windows

* Problem – infinite loop in tracing a path name with symbolic links – 4.3BSD, no 8 passings of soft links
* Dangling pointers

Hard Link vs. Soft Link

- Hard Link
  - Cannot link to different mounted file system.
  - is a different name for the same set of data blocks.
  - Only superuser can create a link to a directory.
- Soft Link (or Symbolic link)
  - It is implemented as a file that contains a pathname.
  - File size = pathname length
  - Can be a directory or file
  - Is a pointer to a set of data blocks.
  - File type is S_IFLINK
- What's the difference on their i-nodes?

File Information

- Three major functions:

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

int stat(const char *pathname, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *pathname, struct stat *buf);
int fstatat(int fd, const char *restrict pathname, struct
stat *restrict buf, int flag);
```

Differences on stat(), fstat(), lstat():

lstat() returns info regarding the symbolic link itself, instead of the referenced file, if it happens.

fstatat(): returns info for a pathname relative to an open directory represented by *fd* argument.

File Properties

- Differences on stat(), fstat(), lstat():
  - lstat() returns info regarding the symbolic link, instead of the referenced file, if it happens.

```
struct stat {
   mode_t        st_mode; /* type & mode */
   ino_t         st_ino; /* i-node number */
   dev_t         st_dev; /* device no (file system) */
   dev_t         st_rdev; /* device no for special file */
   nlink_t       st_nlink; /* # of links */
   uid_t         st_uid; gid_t  st_gid;
   off_t         st_size; /* sizes in bytes */
   struct timespec        st_atime; /* last access time */
   struct timespec        st_mtime; /* last modification time */
   struct timespec        st_ctime; /* time for last status change
   blksize_t          st_blk_size; /* best I/O block size */
   blkcnt_t           st_blocks; /* number of disk blocks allocate
};
```

File Types

- Program 4.1 - Page 96
  - lstat() and Figure 4.1

```
<sys/stat.h>
        #define S_IFMT     0xF000  /* type of file */
        #define S_IFDIR    0x4000  /* directory */
        #define S_ISDIR(mode)   (((mode)&0xF000) == 0x4000)
        #define S_ISDIR(mode)   (((mode) & S_IFMT) == S_IFDIR)
```

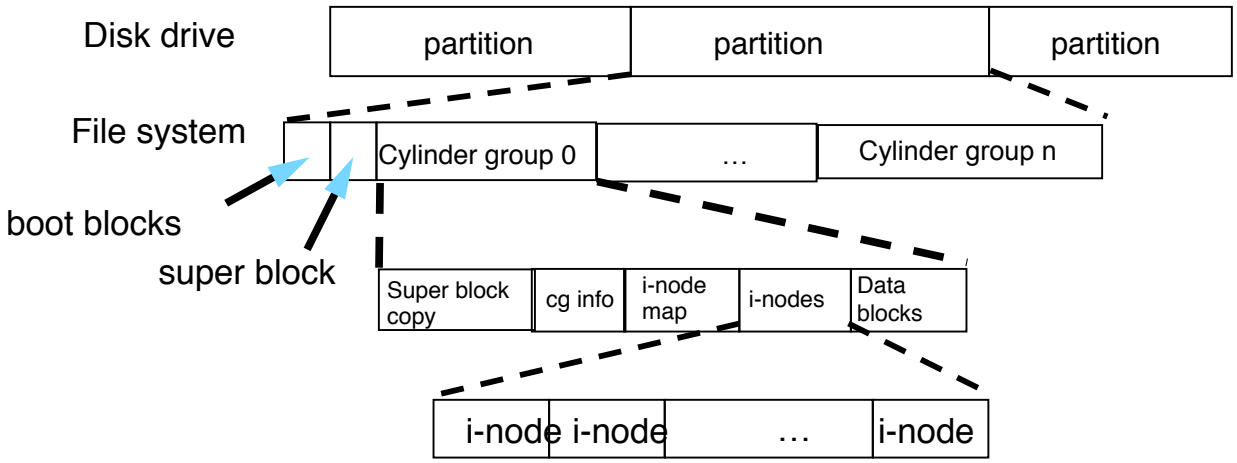
  - st_mode
- Percentage of Files in a Medium-Sized System - Figure 4.4

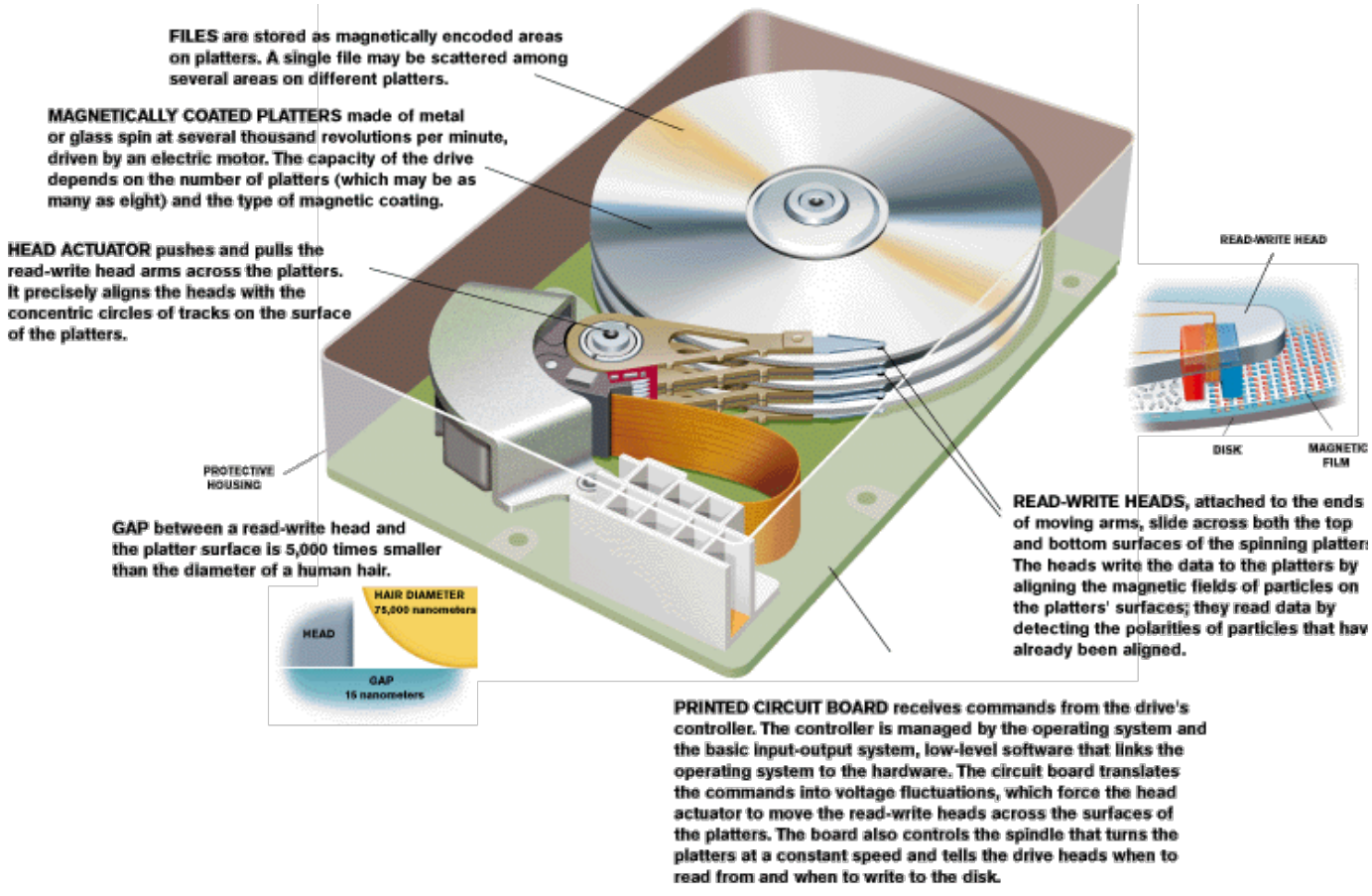
File Type	Count	Percentage
regular file	226,856	88.22%
directory	23,017	8.95%
symbolic link	6,442	2.51%
character special	447	0.17%
block special	312	0.12%
socket	69	0.03%
FIFO	1	0.00%

File systems and Directory Structure

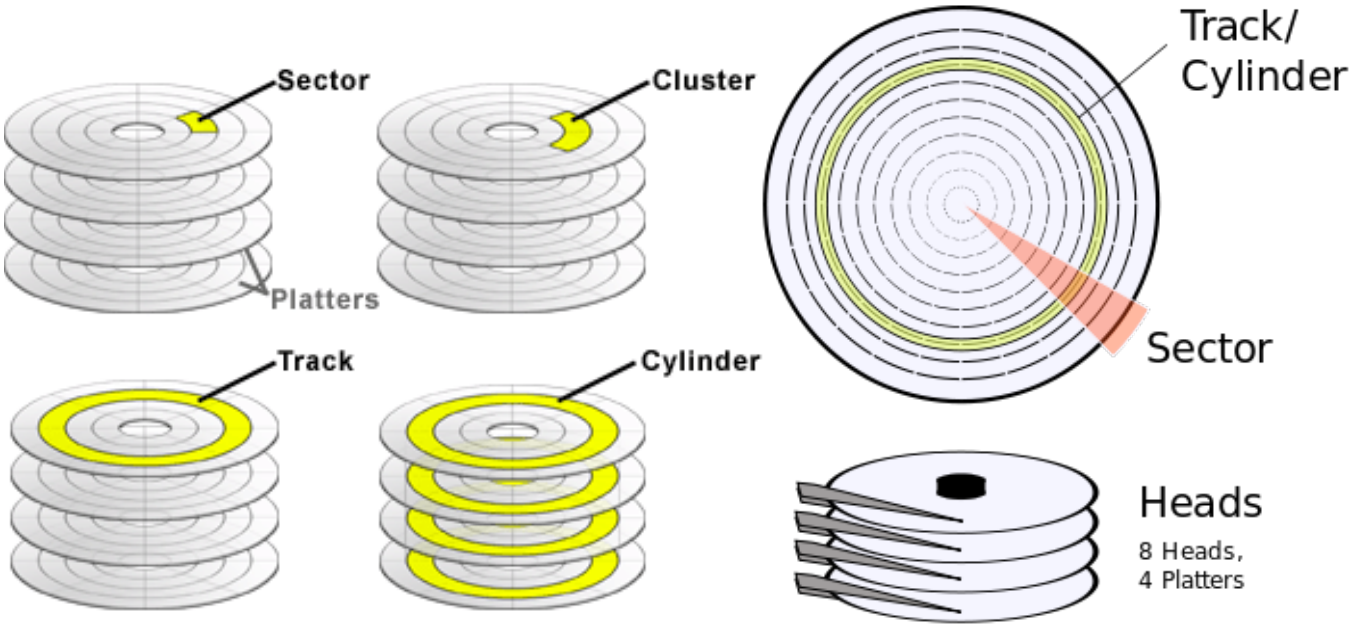
File Systems Arrangement

- A hierarchical arrangement of directories and files – starting in root /
- Several Types, e.g., SVR4: Unix System V File Systems (S5), Unified File System (UFS) – Figure 2.6 (Page 39)
- System V:





Cylinder, track, cluster, and sector



Partition vs. Physical Devices

```
Disk /dev/hda: 123.5 GB, 123522416640 bytes
255 heads, 63 sectors/track, 15017 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot    Start       End    Blocks   Id  System
/dev/hda1   *         1        13    104391   83  Linux
/dev/hda2            14     14895 119539665   83  Linux
/dev/hda3         14896     15017    979965   82  Linux swap
```

```
[root@oris /]# more /etc/mtab
/dev/hda1       /boot    ext3     rw 0 0
/dev/hda2       /        ext3     rw 0 0
none            /proc    proc     rw 0 0
none            /dev/pts devpts   rw,gid=5,mode=620 0 0
usbdevfs        /proc/bus/usb usbdevfs rw 0 0
none            /dev/shm  tmpfs   rw 0 0
/dev/sda1       /mnt/usbhd ext3   rw 0 0
```

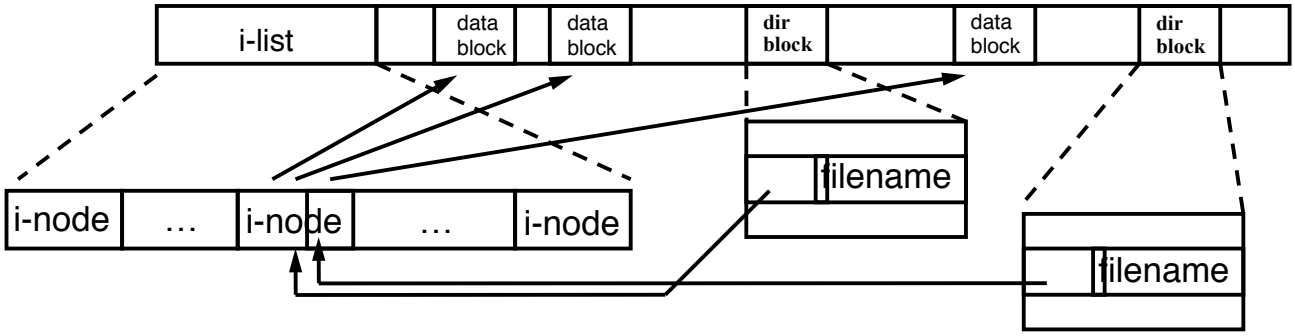
Superblock in File System

- Each file system has different format and information:
  - Type: ext2, ext3, FAT, FAT32 etc.
  - Size: 5 GB, 10 GB, etc and
  - Status such as mount status.
- A superblock contains information about file system such as: File system type, Size, Status, and Information about other metadata structures.
- When superblock of a filesystem is lost, we are in big trouble. So, you will find redundant superblocks in the system.
- Example: Linux with root permission

```
dumpe2fs /dev/sda1 | grep -i superblock
```

```
[root@oris sbin]# ./dumpe2fs /dev/sda1 | grep -i superblock
dumpe2fs 1.40.4 (31-Dec-2007)
  Primary superblock at 0, Group descriptors at 1-7
  Backup superblock at 32768, Group descriptors at 32769-32775
  Backup superblock at 98304, Group descriptors at 98305-98311
  Backup superblock at 163840, Group descriptors at 163841-163847
  Backup superblock at 229376, Group descriptors at 229377-229383
  Backup superblock at 294912, Group descriptors at 294913-294919
  Backup superblock at 819200, Group descriptors at 819201-819207
  Backup superblock at 884736, Group descriptors at 884737-884743
  Backup superblock at 1605632, Group descriptors at 1605633-1605639
  Backup superblock at 2654208, Group descriptors at 2654209-2654215
  Backup superblock at 4096000, Group descriptors at 4096001-4096007
  Backup superblock at 7962624, Group descriptors at 7962625-7962631
  Backup superblock at 11239424, Group descriptors at 11239425-11239431
  Backup superblock at 20480000, Group descriptors at 20480001-20480007
  Backup superblock at 23887872, Group descriptors at 23887873-23887879
[root@oris sbin]#
```

i-node and data blocks

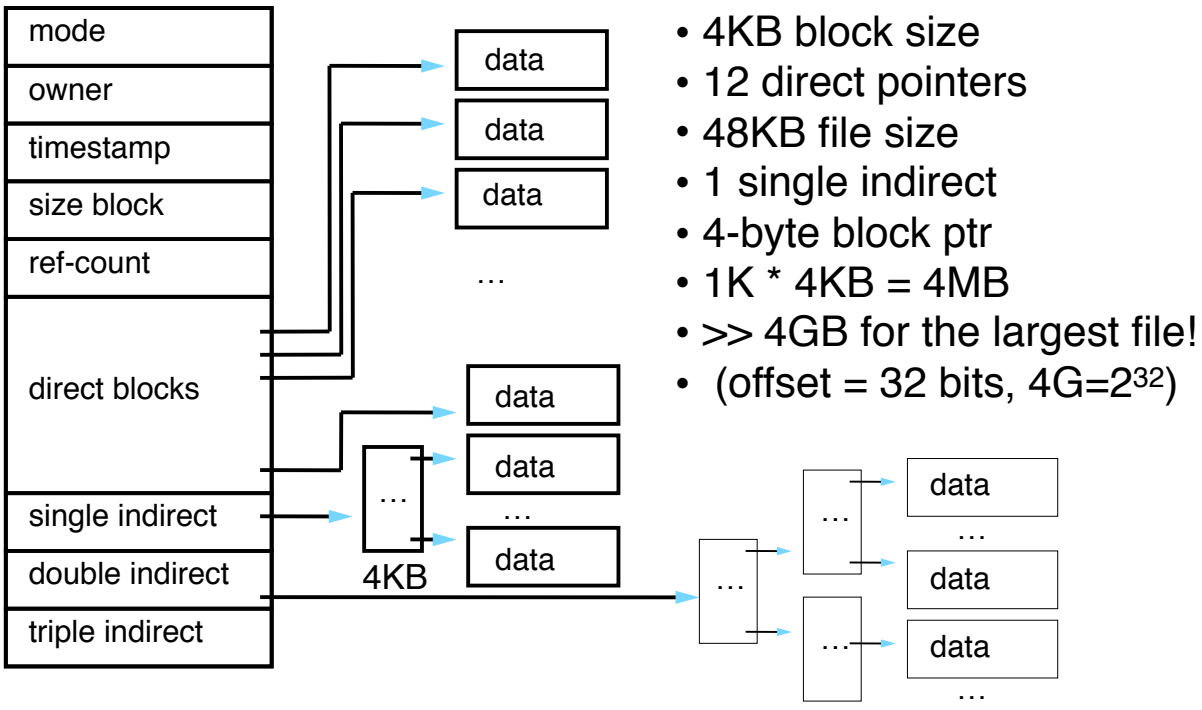


- i-node:
  - Version 7: 64B, 4.3+BSD:128B, S5:64B, UFS:128B
  - File type, access permission, file size, data blocks, etc.
- Link count - hard links
  - st_nlink in stat, LINK_MAX in POSIX.1
  - Unlink/link a file

Moving files among directories

- When files are moved within the same mounted file system, no need to physically move the file.
  - Only directory block is updated.
- Otherwise, the files have to be physically moved.

File System - 4.4BSD i-node



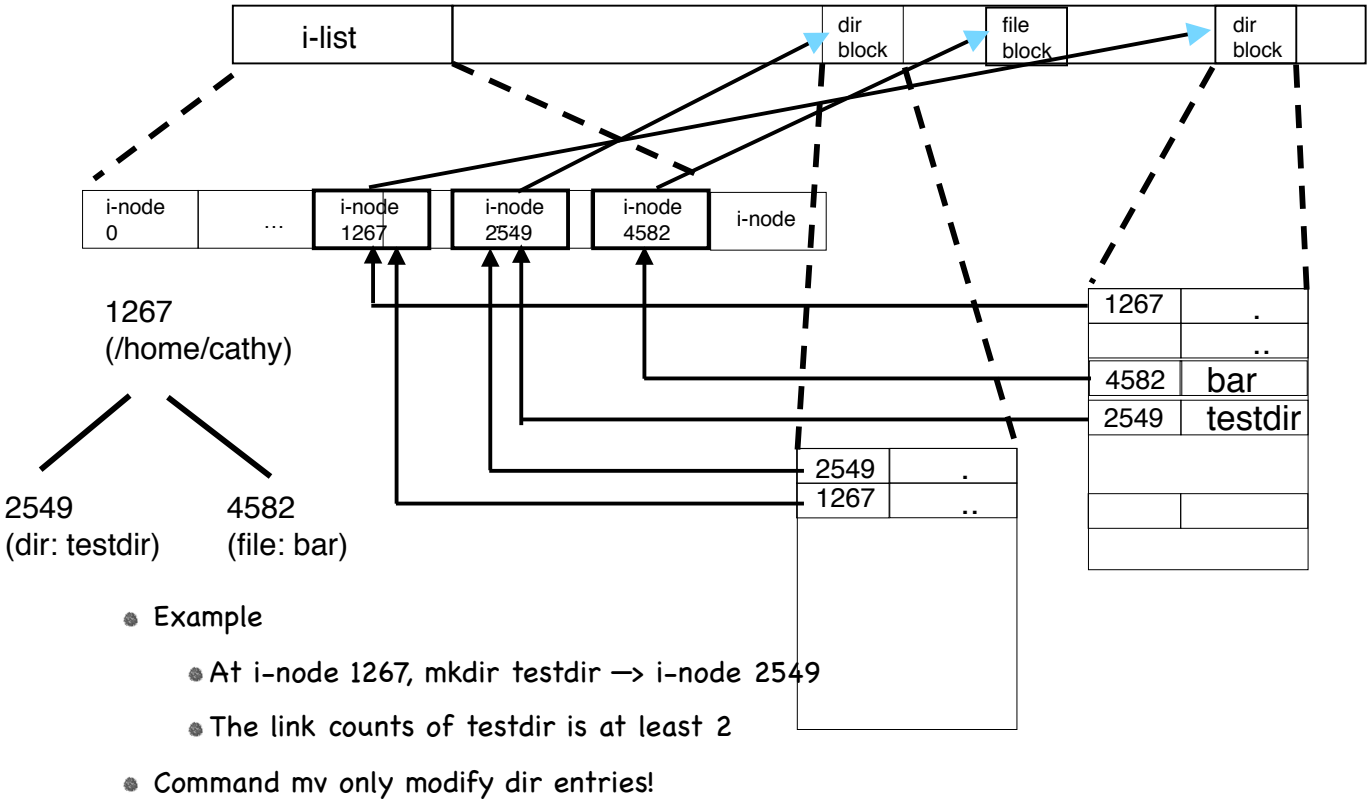
- 4KB block size
- 12 direct pointers
- 48KB file size
- 1 single indirect
- 4-byte block ptr
- 1K * 4KB = 4MB
- >> 4GB for the largest file!
- (offset = 32 bits, $4G=2^{32}$)

* “Operating system concept”, Silberschatz and Galvin, Addison Wesley, pp. 380.

Link

- Files are identified by i-nodes in the file systems.
  - It is a long string, usually represented by a long address, to be read by the operating system.
  - I-node is not designed for human users.
- Directories and filename are widely used by librarian/human users to organize large volumes of documents, long before there are computers.
  - Link is the pointer to bridge legible file name/ directory name to i-node.
  - One i-node can have more than one name, each of which is represented by one link.

Links to organize files/directories



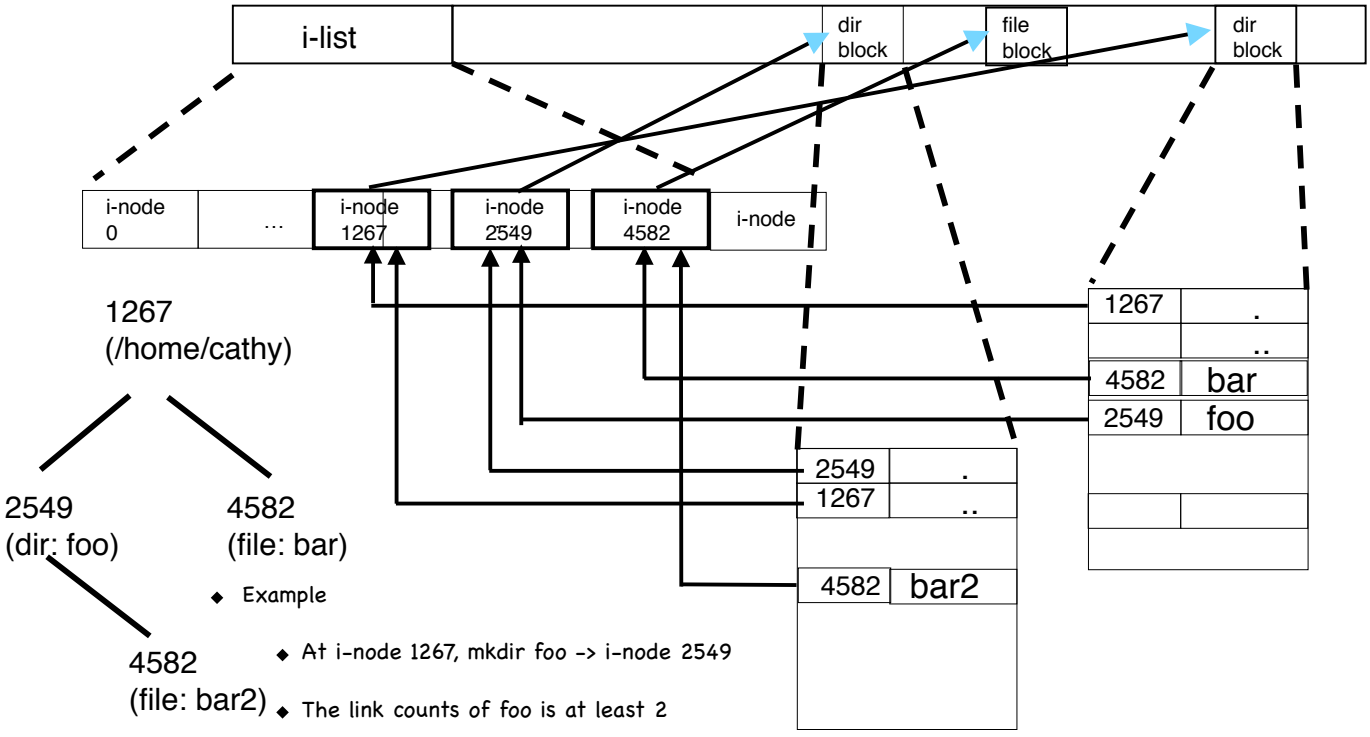
- Example
  - At i-node 1267, mkdir testdir —> i-node 2549
  - The link counts of testdir is at least 2
- Command mv only modify dir entries!
- No directory entry points at any i-node residing at other file systems.

Functions – link and unlink

```
#include <unistd.h>
int link(const char *existingpath, const char *newpath);
int linkat(int efd, const char *existingpath, int nfd, const
    char *newpath, int flag);
int unlink(const char *pathname);
```

- Atomic action for link – hard link
  - POSIX.1 allows linking across file systems (most systems do not support this implementation.)
  - Only superusers could create a link to a dir
- Error if newpath exists
- Unlink – WX permission at the residing dir
  - Remove the dir entry & delete the file if link count reaches zero and no one still opens the file (Remark: sticky bit & dir rights).

Links to organize files/directories



- Example
  - At i-node 1267, mkdir foo -> i-node 2549
  - The link counts of foo is at least 2
- Command mv only modify dir entries!
- No directory entry points at any i-node residing at other file systems.

Example for link and unlink

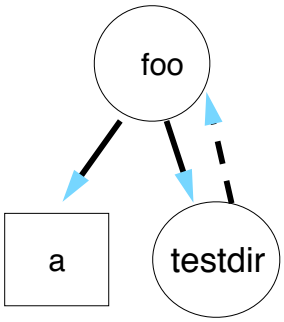
- Program 4.16 – Page 118
  - Open & unlink a file
- Unlink a file
  - When sticky bits set for a residing dir, we must have the write permission for the directory and either owner of the file, the dir, or super users.
  - If pathname is a symbolic link, it removes the symbolic link and does not change the count value of the referred i-node.

Functions – rename and remove

```
#include <stdio.h>
int remove(const char *pathname);
int rename(const char *oldname, const char *newname);
```

- remove =
  - unlink if pathname is a file.
  - rmdir if pathname is a dir. (ANSI C)
- Rename – ANSI C
  - File: both files, newname is removed first, WX permission for both residing directories
  - Directory: both dir, newname must be empty, newname could not contain oldname. E.g., rename("bar", "bar/foo").

Symbolic Links



- Goal
  - Get around the limitations of hard links: (a) file system boundary (b) link to a dir.
  - Initially introduced by 4.2BSD
- Example – ftw: file tree walk
  - ln -s ../foo testdir
- Figure 4.17 – functions follow slinks
  - No functions which take filedes
  - Example: unlink(testdir)

Files follow symbolic link

	Does not	Follows
access		Y
chdir		Y
chmod		Y
chown	Y	Y
creat		Y
exec		Y
lchown	Y	
link		Y
truncate		Y

	Does not	Follows
lstat	Y	
open		Y
opendir		Y
pathconf		Y
readlink	Y	
remove	Y	
rename	Y	
stat		Y
unlink	Y	

Create/Read Symbolic Links

```
#include <unistd.h>
int symlink(const char *actualpath, const char *sympath);
int readlink(const char *pathname, char *buf, int
    bufsize);
```

- actualpath does not need to exist!
  - They do not need to be in the same file system.
- readlink is an action consisting of open, read, and close – not null terminated.

Hard Link/Soft Link in Windows-based OS

- NTFS Hard link, similar to hard link in POSIX
  - Available since Windows 2000
  - Link to files on the same volumes
  - fsutil hardlink create <NewFileName> <ExistingFileName>
- NTFS Junction,
  - Can link files and directories to different volumes on the same host.
  - linkd mydesktop user profile/desktop
  - linkd mydesktop /d
- Symbolic link, similar to soft link in POSIX
  - Available since NTFS 5.0, i.e., Windows Vista
  - Can also point to a file or remote SMB network path
  - mklink [[/D] | [/H] | [/J]] link target
    - /D – Creates a directory symbolic link. Default is a file symbolic link.
    - /H – Creates a hard link instead of a symbolic link.
    - /J – Creates a Directory Junction.

Discussion

- Alice creates a symbolic link and removes the target file
  - ln -s bar foo
  - rm bar
- and, in her program, she writes
  - open("foo", O_RDONLY);
- What will open() return?

Functions - mkdir and rmdir

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *pathname, mode_t mode);
```

- umask, UID/GID setup (Sec 4.6)
- From 4.2BSD & SVR3 (SVR4 - inheritance of the S_ISGID bit)
- At least one of the execute bits are enabled.

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

- An empty dir is deleted.
- Link count reaches zero, and no one still opens the dir.

Functions - opendir, readdir, rewinddir, closedir

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *pathname);
struct dirent *readdir(DIR *dp);
void rewinddir(DIR *dp);
int closedir(DIR *dp);
```

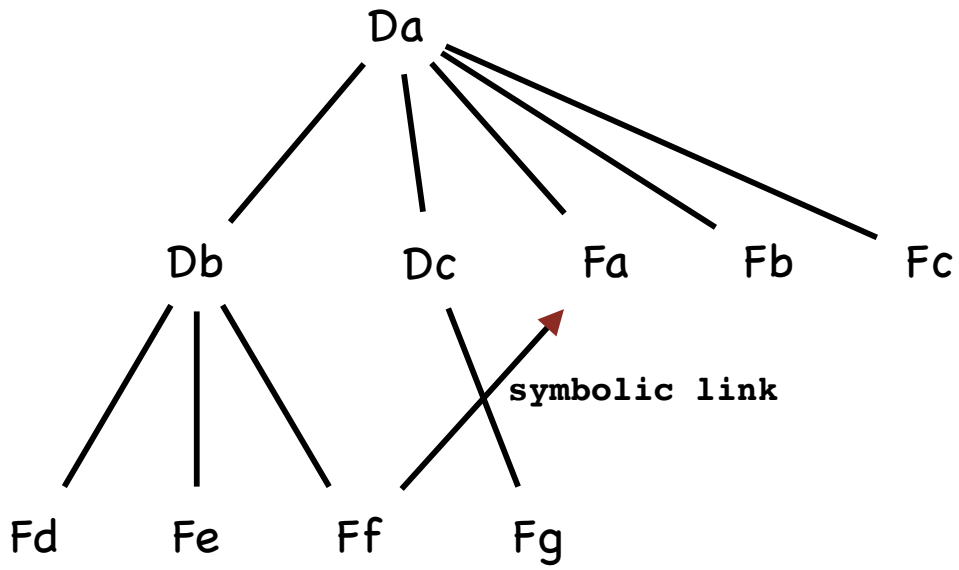
- Only the kernel can write to a dir!!!
- WX for creating/deleting a file!
- Not POSIX standard but XSI extension for UNIX

Functions - opendir, readdir, rewinddir, closedir

- dirent struct is implementation-dependent, e.g.,

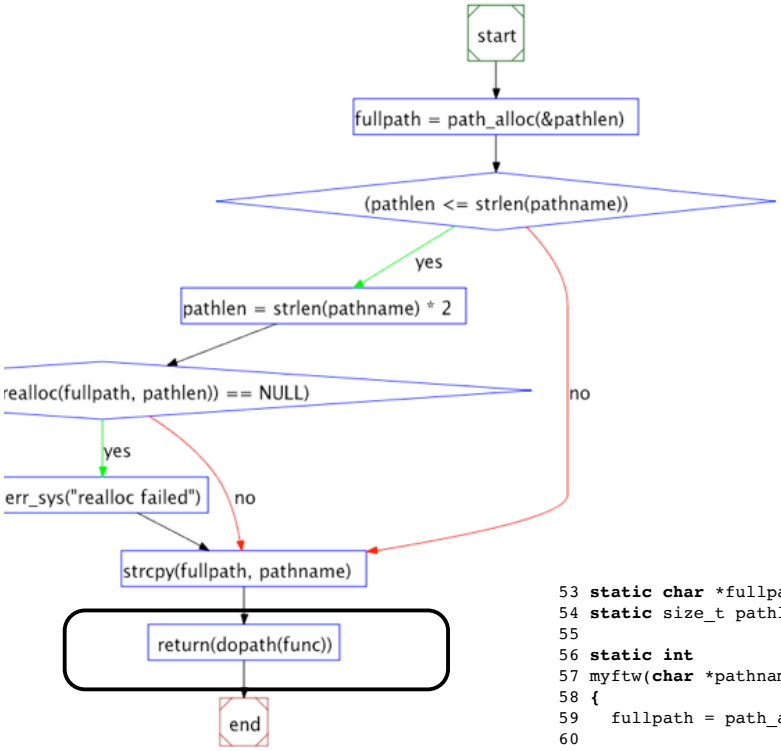
```
struct dirent {
  ino_t d_ino;  /* not in POSIX.1 */
  char d_name[NAME_MAX+1];
 } /* fpathconf() */
```

- Program 4.22 - Pages 132-133
  - ftw/nftw - recursively traversing the file system

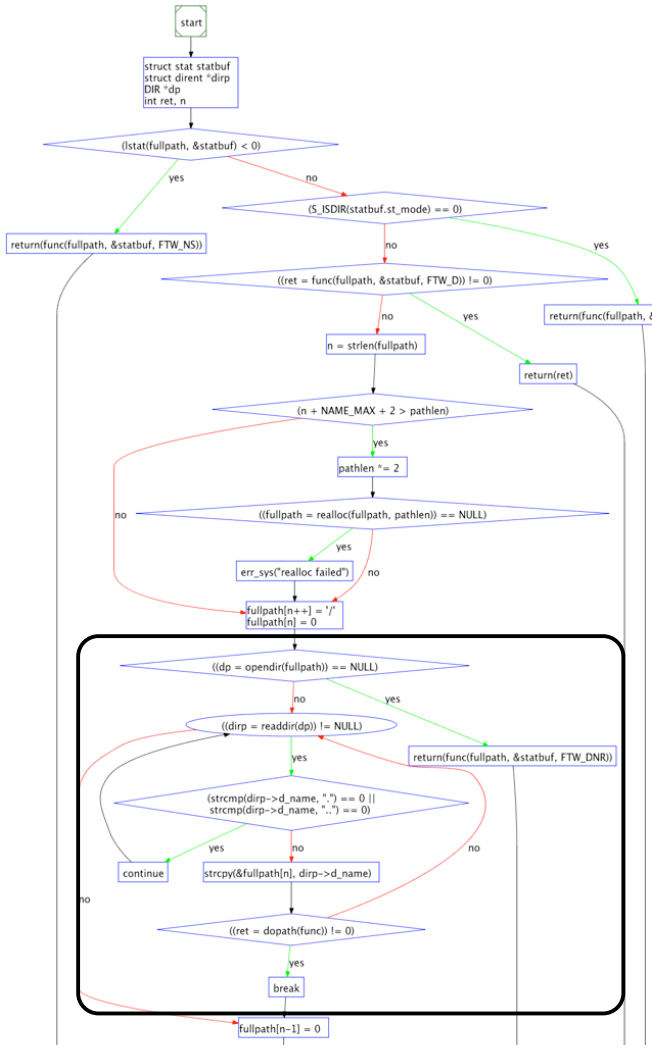


	Da	Db	Fd	Fe	Ff	Dc	Fg
nreg	0	0	1	2	2	2	3
ndir	1	2	2	2	2	3	3

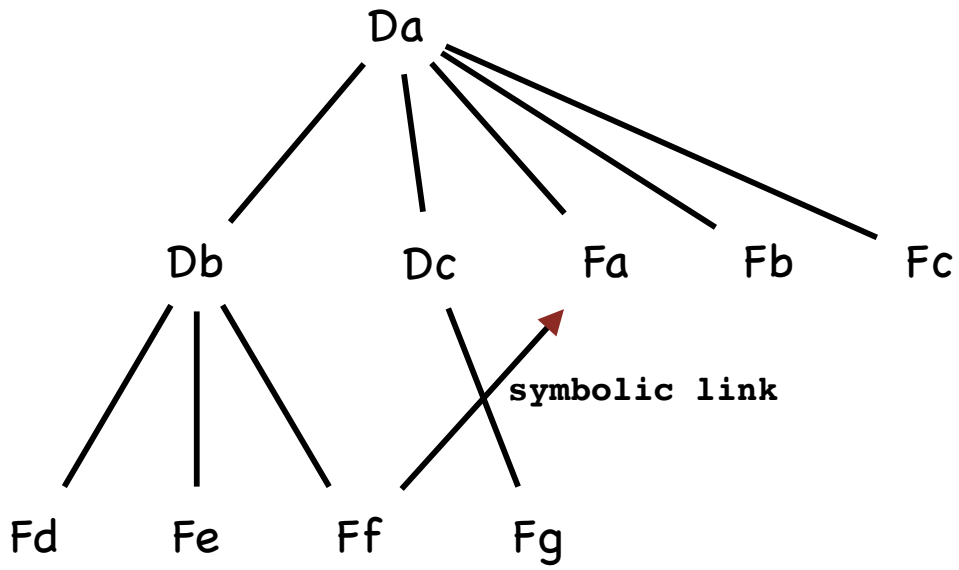

```
14 int
15 main(int argc, char *argv[])
16 {
17    int   ret;
18
19    if (argc != 2)
20       err_quit("usage:  ftw  <starting-pathname>");
21
22    ret = myftw(argv[1], myfunc);   /* does it all */
23
24    ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock;
25    if (ntot == 0)
26       ntot = 1;   /* avoid divide by 0; print 0 for all counts */
27    printf("regular files  = %7ld, %5.2f %%\n", nreg,
28      nreg*100.0/ntot);
29    printf("directories    = %7ld, %5.2f %%\n", ndir,
30      ndir*100.0/ntot);
31    printf("block special  = %7ld, %5.2f %%\n", nblk,
32      nblk*100.0/ntot);
33    printf("char special   = %7ld, %5.2f %%\n", nchr,
34      nchr*100.0/ntot);
35    printf("FIFOs          = %7ld, %5.2f %%\n", nfifo,
36      nfifo*100.0/ntot);
37    printf("symbolic links = %7ld, %5.2f %%\n", nslink,
38      nslink*100.0/ntot);
39    printf("sockets        = %7ld, %5.2f %%\n", nsock,
40      nsock*100.0/ntot);
41    exit(ret);
42 }
43
```




```
53 static char *fullpath;    /* contains full pathname for every file */
54 static size_t pathlen;
55
56 static int          /* we return whatever func() returns */
57 myftw(char *pathname, Myfunc *func)
58 {
59   fullpath = path_alloc(&pathlen);  /* malloc PATH_MAX+1 bytes */
60                     /* ({Prog pathalloc}) */
61   if (pathlen <= strlen(pathname)) {
62     pathlen = strlen(pathname) * 2;
63     if ((fullpath = realloc(fullpath, pathlen)) == NULL)
64       err_sys("realloc failed");
65   }
66   strcpy(fullpath, pathname);
67   return(dopath(func));
```




```
76 static int          /* we return whatever func() returns */
77 dopath(Myfunc* func)
78 {
79   struct stat   statbuf;
80   struct dirent *dirp;
81   DIR       *dp;
82   int       ret, n;
83
84   if (lstat(fullpath, &statbuf) < 0)  /* stat error */
85     return(func(fullpath, &statbuf, FTW_NS));
86   if (S_ISDIR(statbuf.st_mode) == 0)  /* not a directory */
87     return(func(fullpath, &statbuf, FTW_F));
88
89   /*
90    * It's a directory.  First call func() for the directory,
91    * then process each filename in the directory.
92    */
93   if ((ret = func(fullpath, &statbuf, FTW_D)) != 0)
94     return(ret);
95
96   n = strlen(fullpath);
97   if (n + NAME_MAX + 2 > pathlen) { /* expand path buffer */
98     pathlen *= 2;
99     if ((fullpath = realloc(fullpath, pathlen)) == NULL)
100      err_sys("realloc failed");
101  }
102  fullpath[n++] = '/';
103  fullpath[n] = 0;
104
105  if ((dp = opendir(fullpath)) == NULL) /* can't read directory */
106    return(func(fullpath, &statbuf, FTW_DNR));
107
108  while ((dirp = readdir(dp)) != NULL) {
109    if (strcmp(dirp->d_name, ".") == 0  ||
110        strcmp(dirp->d_name, "..") == 0)
111      continue;    /* ignore dot and dot-dot */
112    strcpy(&fullpath[n], dirp->d_name); /* append name after "/" *
113    if ((ret = dopath(func)) != 0)    /* recursive */
114      break;  /* time to leave */
115  }
116  fullpath[n-1] = 0;  /* erase everything from slash onward */
117
118  if (closedir(dp) < 0)
119    err_ret("can't close directory %s", fullpath);
120  return(ret);
121 }
```



	Da	Db	Fd	Fe	Ff	Dc	Fg
nreg	0	0	1	2	2	2	3
ndir	1	2	2	2	2	3	3

Functions - chdir, fchdir, getcwd

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

- fchdir - not POSIX.1
- chdir must be built into shells!
- The kernel only maintains the i-node number and dev ID for the current working directory!
- Per-process attribute - working dir!

- Program 4.23 - Page 135
  - chdir, which does perform like chdir in command line.
  - How can you write a program to change the directory of your shell environment?

Functions - getcwd

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

- The system keeps the current working directory in the form of i-node number. getcwd() return the full path.
- The buffer must be large enough, or an error returns!
- chdir follows symbolic links, and getcwd has no idea of symbolic links!

- Program 4.24 - Page 137
  - getcwd

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

int main(int argc, char *argv[])
{
        char save_cwd[1024];
        char buf[1024];

        /* Save the current working directory */
        if (getcwd(save_cwd, sizeof(save_cwd)) == NULL)
                perror("getcwd error");
        printf("before chdir, cwd = %s\n", save_cwd);



        /* change to a different working directory */
        if (chdir("/") < 0)
                perror("chdir error");
        if (getcwd(buf, sizeof(buf)) == NULL)
                perror("getcwd error");
        printf("after chdir, cwd = %s\n", buf);

        /* Return to earlier working directory */
        if (chdir(save_cwd) < 0)
                perror("chdir error");
        if (getcwd(buf, sizeof(buf)) == NULL)
                perror("getcwd error");
        printf("previous chdir, cwd = %s\n", buf);
        exit(0);
}
```

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

int main(int argc, char *argv[])
{
        char buf[1024];
        int fd;
        /* Save the current working directory */
        if ((fd = open(".", O_RDONLY)) < 0)
                perror("open error");
        if (getcwd(buf, sizeof(buf)) == NULL)
                perror("getcwd error");
        printf("before chdir, cwd = %s\n", buf);


        /* change to a different working directory */
        if (chdir("/") < 0)
                perror("chdir error");
        if (getcwd(buf, sizeof(buf)) == NULL)
                perror("getcwd error");
        printf("after chdir, cwd = %s\n", buf);


        /* Return to earlier working directory */
        if (fchdir(fd) < 0)
                perror("fchdir error");
        if (getcwd(buf, sizeof(buf)) == NULL)
                perror("getcwd error");
        printf("previous chdir, cwd = %s\n", buf);

        exit(0);
}
```

File Times

```
#include <sys/types.h>
#include <utime.h>
int utime(const char *pathname, const struct utimbuf *times);
int utimes(const char *pathname, const struct timeval
times[2]);
```

- time values are in seconds since the Epoch
- times = null -> set as the current time
  - Effective UID = file UID or W right to the file
- times != null -> set as requested
  - (Effective UID = file UID or superuser) and W right to the file.
- Program 4.6 (zap.c) – Page 118, utime
- What about ctime?

```
struct utimbuf {
 time_t actime;
 time_t modtime;
 }
```

```
struct timeval {
  time_t          tv_sec;   /* seconds */
   suseconds_t   tv_usec;   /* microsecon
};
```

File Times

- Three Time Fields:

Field	Description	Example	ls option
st_atim	last-access-time	Read	-u
st_mti	last-modification-	Write	default
st_ctim	last-i-node-change-	chmod, chown	-c

- Figure 4.20 – Effect of functions on times
- Changing the access permissions, user ID, link count, etc, only affects the i-node!
  - ctime is modified automatically! (stat, access)
  - Changed by the following functions: chmod(), chown(), creat(), link(2), mknod(), pipe(), unlink(2), utime(), and write().
- Example Question: reading/writing a file only affects the file, instead of the residing dir (Fig4.20).

On Windows platforms, st_ctime refers to the creation time of the file

Discussion

- When you change the name of a file, which time attributes are changed?

- When you are asked to remove the 'core' or unused files which have not been read for one month, which time attributes should you use?

Access Permission

Access Permissions

- File Access Permission:
  - Permissions include read, write, execute, and others.
  - Permissions are granted according to user identity and user groups identity.

```
  8 7 6   5 4 3   2 1 0
|-------|-------|-------|
| user    group   other |
| r w x | r w x | r w x |
|-----------------------|
```

Octal	Binary	Permission
0	000	no permission
1	001	execute
2	010	write
3	011	write and execute
4	100	read
5	101	read and execute
6	110	read and write
7	111	read, write, and execute

Access Permissions & UID/GID

- Operations vs. Permissions
  - Directory
    - X – pass through the dir (search bit), e.g., /usr/dict/words and PATH env var.
    - R – list of files under the dir.
    - W – update the dir, e.g., delete or create a file.
      - Everyone has the write permission for /tmp. Does it mean that you can delete any file in /tmp?
  - File
    - X – execute a file (which must be a regular file)
    - R – O_RDONLY or O_RDWR
    - W – O_WRONLY, O_RDWR, or O_TRUNC

Function - chmod & fchmod

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *pathname, mode_t  mode);
int fchmod(int filedes, mode_t  mode);
int fchmodat(int fd, const char *pathname, mode_t mode,
    int flag);
```

- fchmod() is not in POSIX.1, but in SVR4/4.3+BSD
- Callers must be a superuser or effective UID = file UID.
- Mode = bitwise-OR (Fig 4.11).
  - S_I[RWX]USR,
  - S_I[RWS]GRP,
  - S_I[RWX]OTH,
  - S_ISVTX: sticky bit,
  - S_IS[UG]ID:set-usr-id/set-group-id.

Special Bits

- Sticky bit: 'T' or 't' on directory represents a sticky bit to protect files in the directory from misuse.
- Set-User-ID: 's' on executable files lead to user ID changes to provide additional privilege.

Sticky Bit

Permission on shared folders

- On systems, there are directories which are shared by many users.
- The users can use such directories to exchange files/information among processes owned different users.
- However,
  - with write permission to the directory for all the users in the group, one may delete files owned by other users.

Sticky Bit

- Sticky Bit (S_ISVTX) – saved-text bit
  - Not POSIX.1 – by SVR4 & 4.3+BSD
  - Only superusers can set it!
  - S_ISVTX directory file, e.g., /tmp
    - Remove/rename its file only if 'w' permission of the dir is set, and the process is belonging to superusers/owner of the file/dir.
  - S_ISVTX executable file
    - Used to save a copy of a S_ISVTX executable in the swap area to speed up the execution next time.
    - Not needed for a system with a virtual memory system and fast file system.

Set-Usr-ID/set-Group-ID

The privilege of a process

```
int main()
{
    FILE *fp = fopen("./write.txt", "w+");
    testing_setbuf( fp );
    fclose( fp );
    return 0;
}
```

- The privilege of a process can be defined by the ID of the user starting the program.
- Single identity shows your royalty but limits the flexibility.

Real User ID vs. Effective User ID

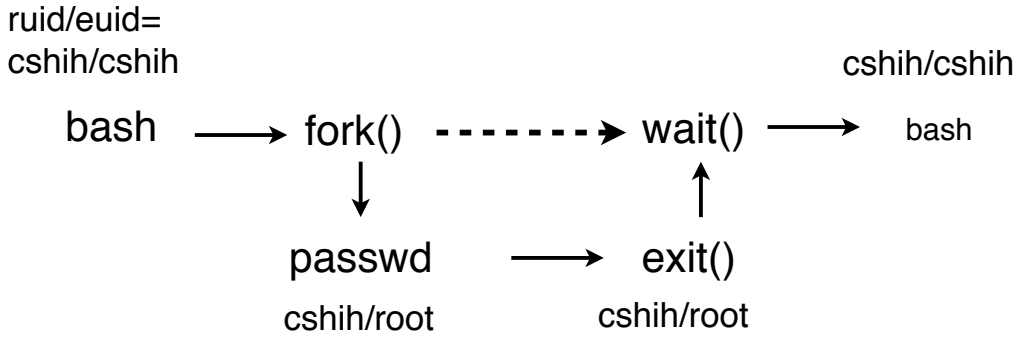
- Effective User ID is mostly used for determining the permission for accessing a file.

- **Real User ID:**
  - When a user executes a program file, the real user ID is the user ID of the user.
- **Effective User ID:**
  - When a user executes a program file, the effective user ID is USUALLY the real user ID.
  - However, when SUID bit is set, the effective user ID is the owner of the program file.

/usr/bin/passwd

- /usr/bin/passwd:
  - user perms (root) - read, setuid on execute
  - group perms (sys) - read and execute/read and setgid on execute
  - others/anyone perms - read and execute

```
iMac:~ cshih$ ls -l /usr/bin/passwd
-r-sr-xr-x  1 root  wheel  111968  8 18  2010 /usr/bin/passwd
iMac:~ cshih$
```



Set-User-ID/Set-Group-ID

- st_uid/st_gid: user/group ID of the owner of a file.
- How can a user change his/her password without having the write permission to /etc/passwd and /etc/shadow?

```
-rw-r--r--    1 root root 1746  2/21 13:00 /etc/passwd
-r--------    1 root root 1142  2/21 13:01 /etc/shadow
```

- A process could be associated with more than one ID.
  - Real user/group ID
  - Effective user/group ID
  - Supplementary group IDs
  - Saved set-user/group-ID

Example for chmod & fchmod

- Figure 4.12 - Page 107
  - chmod - updates on i-nodes
- For security reason, chmod() will clear set-group-ID
  - If the GID of a newly created file is not equal to the effective GID of the calling process (or one of the supplementary GID's), or the process is not a superuser, clear the set-group-ID bit!
  - if a non-superuser process writes to a set-uid/gid file.

Access Permissions & UID/GID

- File Access Test - each time a process creates/opens/deletes a file
  - If the effective UID == 0 —> superuser!
  - If the effective UID == UID of the file
    - Check appropriate access permissions!
  - If the effective GID or any of its supplementary group* ID == GID of the file
    - Check appropriate access permissions!
  - Check appropriate access permissions for others!
- Related Commands: chmod & umask

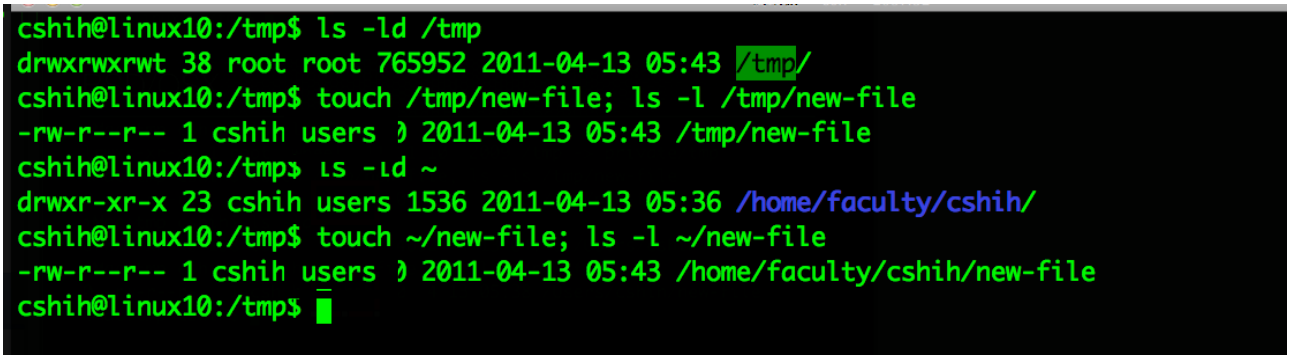
* According IEEE 1003.1 - 1990 standard.

Ownership of new files

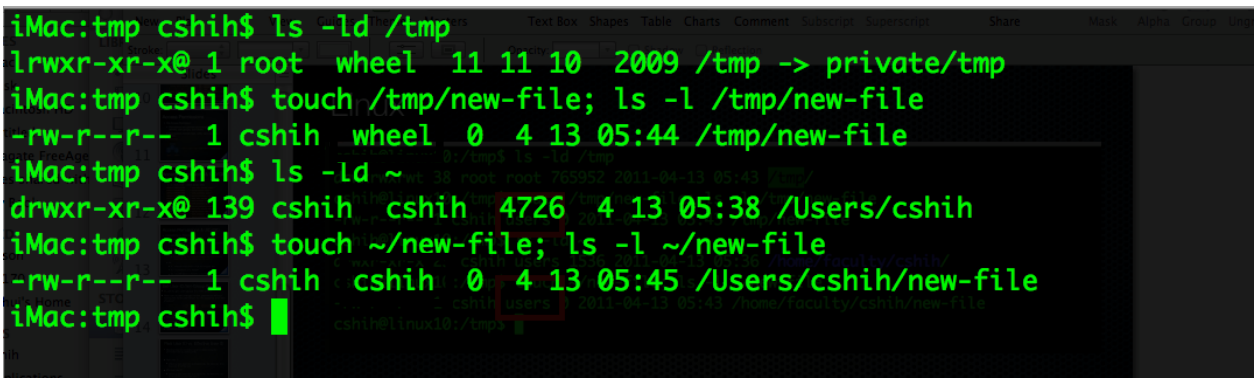
Ownership of a New File

- Rules:
  - Owner UID of a file = the effective UID of the creating process
  - Owner GID of a file - options under POSIX
    - GID of the file = the effective GID of the process
    - GID of the file = the GID of the residing dir
      - 4.3BSD and FIPS 151-1 always do it.
      - SVR4 needs to set the set-group-ID bit of the residing dir (mkdir)!

Linux User's group as the group owner



BSD/OSX Directory's group owner is that of new files



Function - umask

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t  cmask);
```

- The file mode to be turned OFF
  - cmask = bitwise-OR S_I[RWX]USR, etc (Figure 4.4).
  - Set permission to cmask & 0777 for directories
  - Set permission to cmask & 0666 for files
    - If cmask is 022, permission = 0666 & ~022 = 0644 = rw-r--r--
- The mask goes with the process only.
  - Inheritance from the parent!

- Program 4.9 - Page 104
  - umask

Permission for new file/dir

```
iMac:file cshih$ umask
0022
iMac:file cshih$ touch new-file; ls -l new-file
-rw-r--r--  1 cshih  cshih  0  4 13 06:07 new-file
iMac:file cshih$ rm -f new-file
iMac:file cshih$ umask 002
iMac:file cshih$ touch new-file; ls -l new-file
-rw-rw-r--  1 cshih  cshih  0  4 13 06:08 new-file
iMac:file cshih$ mkdir new-dir
iMac:file cshih$ umask
0002
iMac:file cshih$ ls -ld new-dir
drwxrwxr-x  2 cshih  cshih  68  4 13 06:10 new-dir
iMac:file cshih$
```

Function - chown, fchown, fchownat, lchown

```
#include <sys/types.h>
#include <unistd.h>
int chown(const char *pathname, uid_t  owner, gid_t, grp);
int fchown(int filedes, uid_t  owner, gid_t, grp);
int fchownat(int fd, const char *pathname, uid_t owner, gid_t
     group,
int flag);
int lchown(const char *pathname, uid_t  owner, gid_t, grp);
```

- lchown() is unique to SVR4. Under non-SVR4 systems, if the pathname to chown() is a symbolic link, only the ownership of the symbolic link is changed.
- -1 for owner or grp if no change is wanted.

Function - chown, fchown, fchownat, lchown

- _POSIX_CHOWN_RESTRICTED is in effect (check pathconf())
  - Superuser —> the UID of the file can be changed!
  - The GID of the file can be changed if
    - the process owns the file, and
    - Parameter owner = UID of the file & Parameter grp = the process GID or is in supplementary GID's
- set-user/group-ID bits would be cleared if chown is called by non-super users.

File Size

- File Sizes – st_size
  - Regular files – 0~max (off_t)
  - Directory files – multiples of 512 on Linux, or 34 on OSX.
  - Symbolic links – pathname length
  - /* a pipe's file size for SVR4 */
- File Holes
  - st_blocks vs st_size (st_blksize)
    - Many UNIX systems including OSX use 512bytes as the unit, not 1024 bytes.
  - Commands:
    - "ls -l file.hole" == "wc -c file.hole"
    - du -s file.hole —> actual used size on disks, in numbers of blocks.
    - cat -v file.hole > file.hole.copy