系統程式設計

鄭卜壬教授
臺灣大學資訊工程系

Contents

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

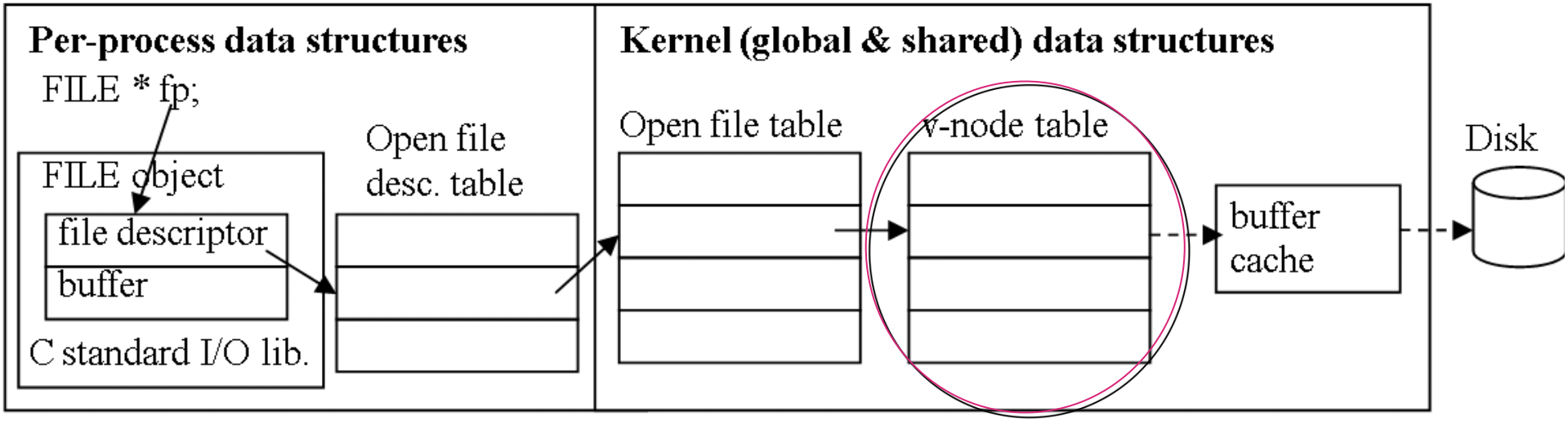
Outline

- **Objectives**
  - additional features of the file system
  - properties of a file.
- **Properties of Files and Directories**
  - ownership
  - access permission
  - time attributes, type, management, etc.
- **File systems**
  - structure of directory tree
  - how to manipulate the directory tree

Files and Directories

- **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);
```



Files and Directories

- **Differences on stat(), fstat(), lstat():**
  - lstat() returns info regarding the symbolic link, instead of the referenced file, if it happens.
  - Used by command “ls –l”

```
struct stat {
  mode_t    st_mode;      /* file type & mode (permissions) */
  ino_t     st_ino;       /* i-node number (serial number) */
  dev_t     st_dev;       /* device number (file system) */
  dev_t     st_rdev;      /* device number for special files */
  nlink_t   st_nlink;     /* number of links */
  uid_t     st_uid;       /* user ID of owner */
  gid_t     st_gid;       /* group ID of owner */
  off_t     st_size;      /* size in bytes, for regular files */
  time_t    st_atime;     /* time of last access */
  time_t    st_mtime;     /* time of last modification */
  time_t    st_ctime;     /* time of last file status change */
  blksize_t st_blksize;   /* best I/O block size */
  blkcnt_t  st_blocks;    /* number of disk blocks allocated */
};
```

512-byte

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

File Type Macros <sys/stat.h>

The argument to each of these macros is the st_mode member from the stat structure

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

```
#define S_IFMT     0xF000  /* type of file */
#define S_IFDIR    0x4000  /* directory */
#define S_ISDIR(mode)   (((mode)& S_IFMT) == S_IFDIR)
```

```
int
main(int argc, char *argv[])
{

    int          i;
    struct stat buf;
    char         *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;

         }
         if (S_ISREG(buf.st_mode))
            ptr = "regular";
         else if (S_ISDIR(buf.st_mode))
            ptr = "directory";
         else if (S_ISCHR(buf.st_mode))
            ptr = "character special";
         else if (S_ISBLK(buf.st_mode))
            ptr = "block special";
         else if (S_ISFIFO(buf.st_mode))
            ptr = "fifo";
         else if (S_ISLNK(buf.st_mode))
            ptr = "symbolic link";
         else if (S_ISSOCK(buf.st_mode))
            ptr = "socket";
         else
            ptr = "** unknown mode **";
         printf("%s\n", ptr);
   }
    exit(0);
}
```

```
$ ./a.out /etc/passwd /etc /dev/initctl /dev/log /dev/tty \
> /dev/scsi/host0/bus0/target0/lun0/cd /dev/cdrom
/etc/passwd: regular
/etc: directory
/dev/initctl: fifo
/dev/log: socket
/dev/tty: character special
/dev/scsi/host0/bus0/target0/lun0/cd: block special
/dev/cdrom: symbolic link
```

An example of counts and percentages of different file types in a Medium-Sized System

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

Access Permissions & UID/GID

- **All files have access permissions**
  - st_mode mask – owner, group, other

```
#define S_IRWXU 00700      /* read, write, execute: owner */
#define S_IRUSR 00400      /* read permission: owner */
#define S_IWUSR 00200      /* write permission: owner */
#define S_IXUSR 00100      /* execute permission: owner */
#define S_IRWXG 00070      /* read, write, execute: group */
#define S_IRGRP 00040      /* read permission: group */
#define S_IWGRP 00020      /* write permission: group */
#define S_IXGRP 00010      /* execute permission: group */
#define S_IRWXO 00007      /* read, write, execute: other */
#define S_IROTH 00004      /* read permission: other */
#define S_IWOTH 00002      /* write permission: other */
#define S_IXOTH 00001      /* execute permission: other */
```

Access Permissions & UID/GID

- **Operations vs Permissions**
  - Directory
    - X – pass through the dir (search bit)
    - R – list of files under the dir.
    - W – update the dir, e.g., delete or create a file.
      - To create a new file, must have write+execute permission for the directory
      - To delete a file, need write+execute on directory, file doesn't matter
      - 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), exec()
    - R – O_RDONLY or O_RDWR
    - W – O_WRONLY, O_RDWR, or O_TRUNC

Set-User-ID/Set-Group-ID

- st_uid/st_gid: user/group ID of the owner.
- 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 have more than one ID.

real user ID real group ID	who we really are
effective user ID effective group ID supplementary group IDs	used for file access permission checks
saved set-user-ID saved set-group-ID	saved by `exec` functions

Access Permissions & UID/GID

- **Real User/Group ID**
  - from /etc/passwd
- **Effective User/Group ID, Supplementary GID's**
  - check for file access permissions
- **Saved Set-User/Group-ID**
  - saved by exec()
  - saved IDs are optional in older versions of POSIX.
  - _POSIX_SAVED_IDS at compile time or call sysconf(_SC_SAVED_IDS) at runtime

- **Set-User-ID**
  - when this file is executed, set the effective user ID of the process to be the owner of the file
  - S_ISUID on st_mode
- **Set-Group-ID**
  - when this file is executed, set the effective group ID of the process to be the group owner of the file
  - S_ISGID on st_mode

Extra File Permissions

- ***Octal Value*** ***Meaning***

`04000` **Set user-id on execution.**
Symbolic: `--s --- ---`

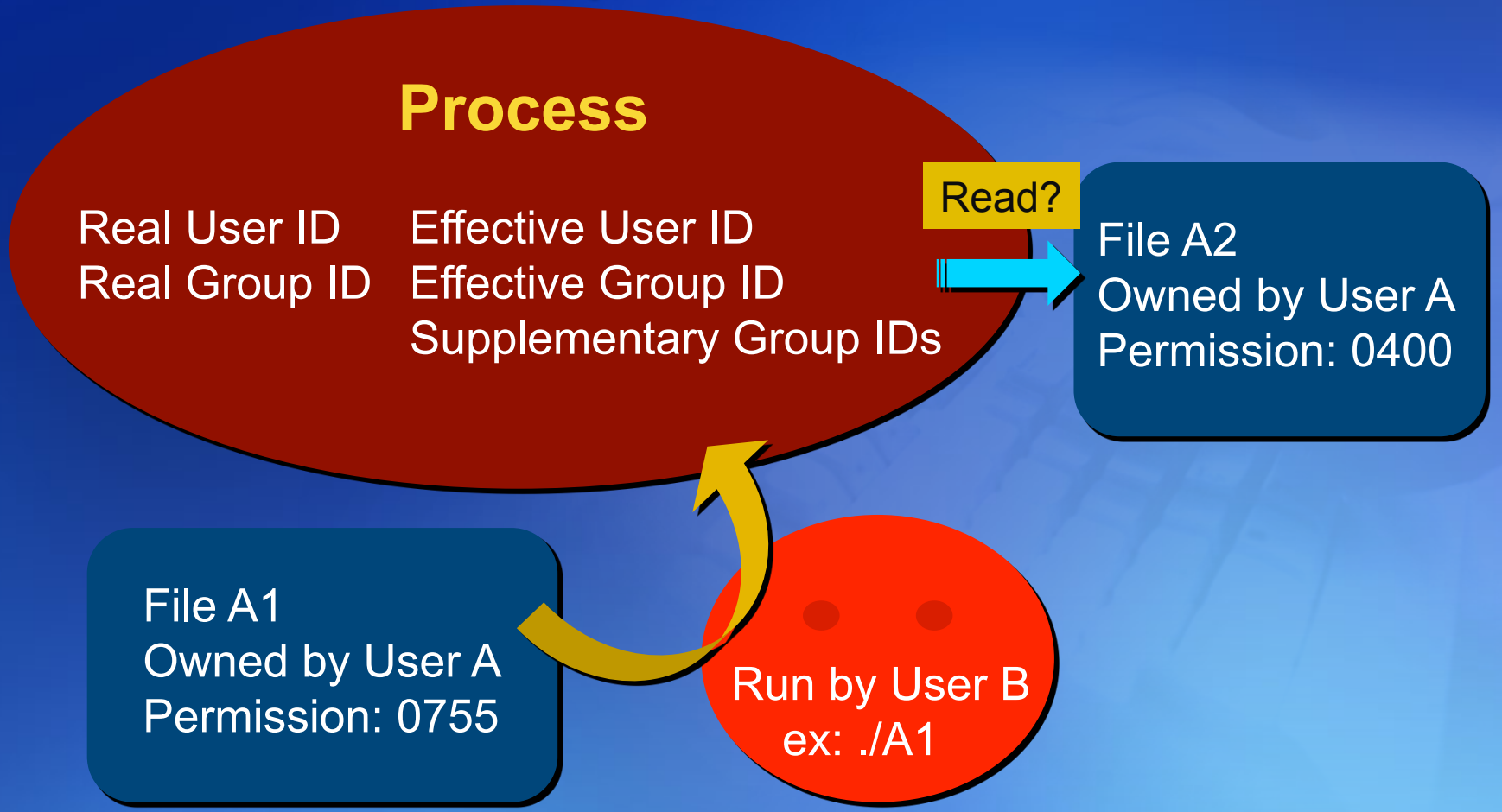
`02000` **Set group-id on execution.**
Symbolic: `--- --s ---`

Permission	Class	Executable[1]	Non-executable[2]
Set User ID (`setuid`)	User	`s`	`S`
Set Group ID (`setgid`)	Group	`s`	`S`
Sticky bit	Others	`t`	`T`

1. The character that will be used to indicate that the execute bit is also set.
2. The character that will be used when the execute bit is not set.

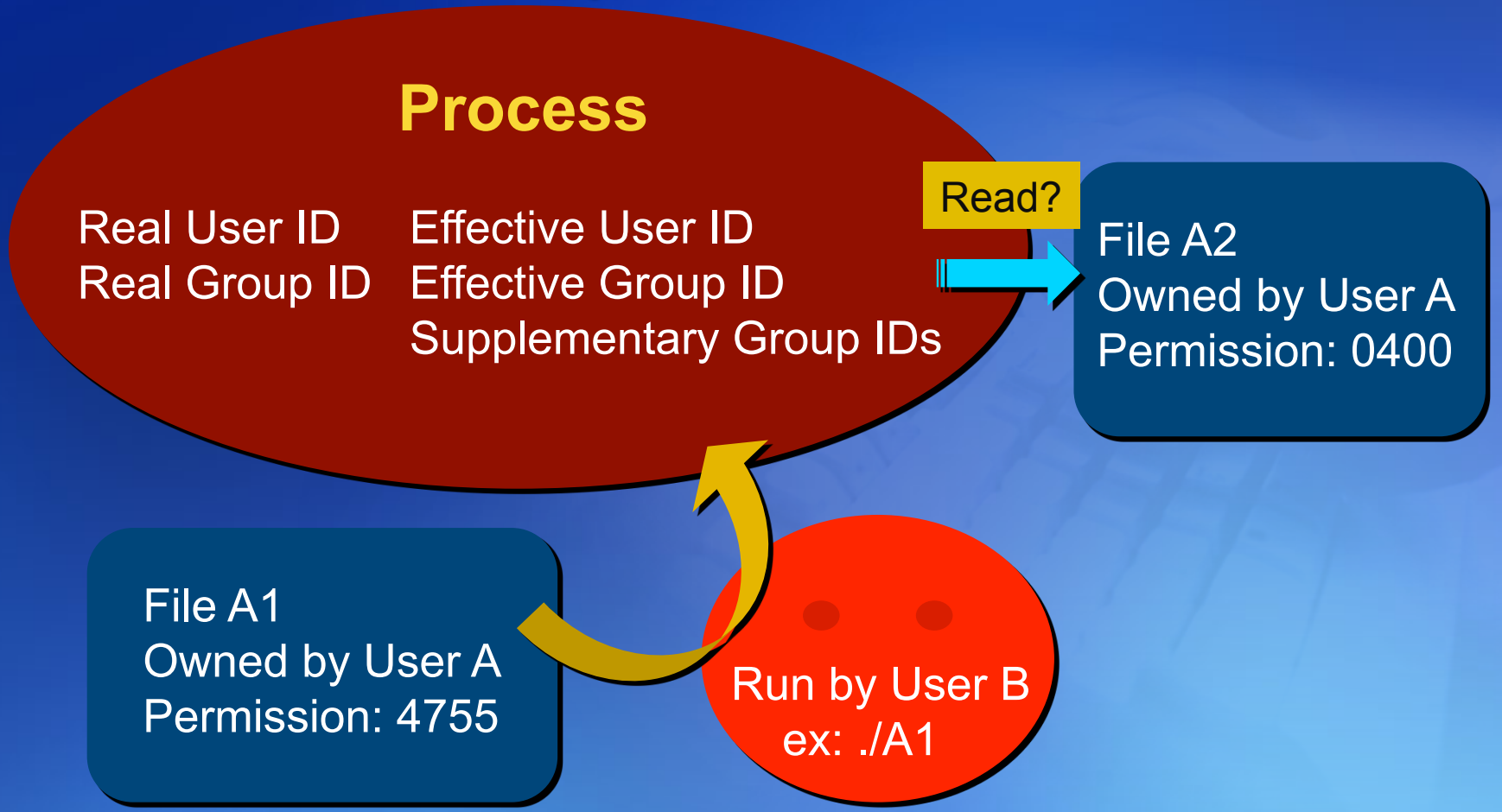
- **For example:**
  - `$ ls -alt /usr/bin/passwd`
    `-rwsr-xr-x 1 root root 25692 May 24...`

What are Real/Effective User ID?



All of the IDs are B; the process cannot read File A2.
If File A2's permission is 0644, the process can read it.

What are Real/Effective User ID?



Real User/Group ID is B. Effective User ID is A.
The process can read File A2.

Think About ...

- **How can a user just allow his friend to copy his files?**
  - Give appropriate permissions to the files (How?)
  - Write a set-user-id program and protect the files himself (How?)
  - Any other methods?

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 a New File

- **UID and GID of a new file by calling open() or creat()**
- **Rules**
  - UID of a file (st_uid) = the effective UID of the creating process
  - GID of a file (st_gid) – options under POSIX
    1. GID of the file = the effective GID of the process
    2. GID of the file = the GID of directory in which it is being created
       - 4.3BSD and FIPS 151-1 always do it.
       - SVR4 needs to set the set-group-ID bit of the residing dir (mkdir)!
       - Ex: /var/spool/mail directory on Linux

Function – access

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

- Check the real UID/GID!
- A process with set-user-ID to root wants to verify that the real user can access a given file

mode	Description
R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute permission
F_OK	test for existence of file

Example

```
int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if (access(argv[1], R_OK) < 0)
        err_ret("access error for %s", argv[1]);
    else
        printf("read access OK\n");
    if (open(argv[1], O_RDONLY) < 0)
        err_ret("open error for %s", argv[1]);
    else
        printf("open for reading OK\n");
    exit(0);
}
```

```
$ ls -l a.out
-rwxrwxr-x 1 sar         15945 Nov 30 12:10 a.out
$ ./a.out a.out
read access OK
open for reading OK
$ ls -l /etc/shadow
-r-------- 1 root         1315 Jul 17 2002 /etc/shadow
$ ./a.out /etc/shadow
access error for /etc/shadow: Permission denied
open error for /etc/shadow: Permission denied
$ su                              become superuser
Password:                         enter superuser password
# chown root a.out                change file's user ID to root
# chmod u+s a.out                 and turn on set-user-ID bit
# ls -l a.out                     check owner and SUID bit
-rwsrwxr-x 1 root      15945 Nov 30 12:10 a.out
# exit                            go back to normal user
$ ./a.out /etc/shadow
access error for /etc/shadow: Permission denied
open for reading OK
```

Function – umask

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

- Turn OFF the file mode
  - cmask = bitwise-OR S_I[RWX]USR, etc.
- The mask goes with the process only.
  - Inheritance from the parent
  - No affection for the mask of its parent
- Return: the previous value of the mask

- **Why is umask a built-in command in a shell?**

Example of umask()

```
#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

int
main(void)
{
    umask(0);
    if (creat("foo", RWRWRW) < 0)
        err_sys("creat error for foo");
    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", RWRWRW) < 0)
        err_sys("creat error for bar");
    exit(0);
}
```

```
$ umask          first print the current file mode creation mask
002
$ ./a.out
$ ls -l foo bar
-rw------- 1 sar              0 Dec 7 21:20 bar
-rw-rw-rw- 1 sar              0 Dec 7 21:20 foo
$ umask                  see if the file mode creation mask changed
002
```

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);
```

- 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 S_I[RWX]USR, S_ISVTX (sticky bit), S_IS[UG]ID, etc.

mode	Description
S_ISUID	set-user-ID on execution
S_ISGID	set-group-ID on execution
S_ISVTX	saved-text (sticky bit)
S_IRWXU	read, write, and execute by user (owner)
S_IRUSR	read by user (owner)
S_IWUSR	write by user (owner)
S_IXUSR	execute by user (owner)
S_IRWXG	read, write, and execute by group
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXO	read, write, and execute by other (world)
S_IROTH	read by other (world)
S_IWOTH	write by other (world)
S_IXOTH	execute by other (world)

Function – chmod & fchmod

- **chmod – updates on i-nodes**
- **chmod() automatically clears two permission bits under the conditions**
  - If we try to set the sticky bit (S_ISVTX) on a regular file and do not have superuser privileges, the sticky bit in the *mode* is automatically turned off.
  - 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!
  - Clear up set-user/group-ID bits if a non-superuser process writes to a set-uid/gid file.

```
int
main(void)
{
      struct stat       statbuf;

      /* turn on set-group-ID and turn off group-execute */

      if (stat("foo", &statbuf) < 0)
          err_sys("stat error for foo");
      if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
          err_sys("chmod error for foo");

      /* set absolute mode to "rw-r--r--" */

      if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
          err_sys("chmod error for bar");

      exit(0);
}
```

```
$ ls -l foo bar
-rw-r--r-- 1 sar              0 Dec 7 21:20 bar
-rw-rwSrw- 1 sar              0 Dec 7 21:20 foo
```

Function – chmod & fchmod

- **Sticky Bit (S_ISVTX) – saved-text bit**
  - Not POSIX.1 – by SVR4 & 4.3+BSD
  - 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.
  - If the bit is set for a directory, a file in the directory can be removed or renamed only if the user has write permission for the directory and one of the following:
    - Owns the file
    - Owns the directory
    - Is the superuser
  - 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

Sticky Bit

- ***Octal*** ***Meaning***

  `01000` **Save text image on execution.**

  Symbolic: `--- --- --t`

- **For example:**

  ls –ld /tmp

  drwxrwxrwt 4 root sys 485 Nov 10 06:01 /tmp

Function – chown, fchown, 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 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, 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
    - This means that when _POSIX_CHOWN_RESTRICTED is in effect, you can't change the user ID of other users' files. You can change the group ID of files that you own, but only to groups that you belong to.
- 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 16/512
  - Symbolic links – pathname length
- **File Holes**
  - st_blocks vs st_size (st_blksize)

```
$ ls -l core
-rw-r--r-- 1 sar       8483248 Nov 18 12:18 core
$ du -s core
272         core        (272 512-byte blocks (139,264 bytes))
$ wc -c core
 8483248 core
$ cat core > core.copy
$ ls -l core*
-rw-r--r--  1 sar       8483248 Nov 18 12:18 core
-rw-rw-r--  1 sar       8483248 Nov 18 12:27 core.copy
$ du -s core*
272     core
16592   core.copy
```

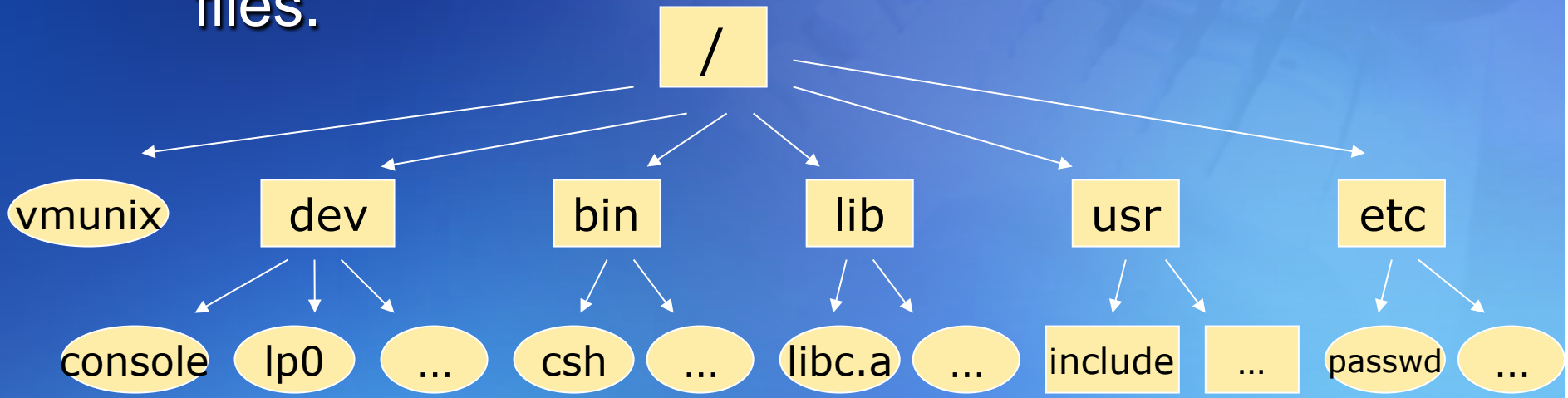

Functions – truncate & ftruncate

```
#include <sys/types.h>
#include <unistd.h>
int truncate(const char *pathname, off_t
    length);
int ftruncate(int filedes, off_t length);
```

- Not POSIX.1
- SVR4: creates holes if length > fsize.
- 4.3+BSD: only truncates files to make them smaller.
- Solaris: fcntl with F_FREESP to free any part of a file.

UNIX File and Directory

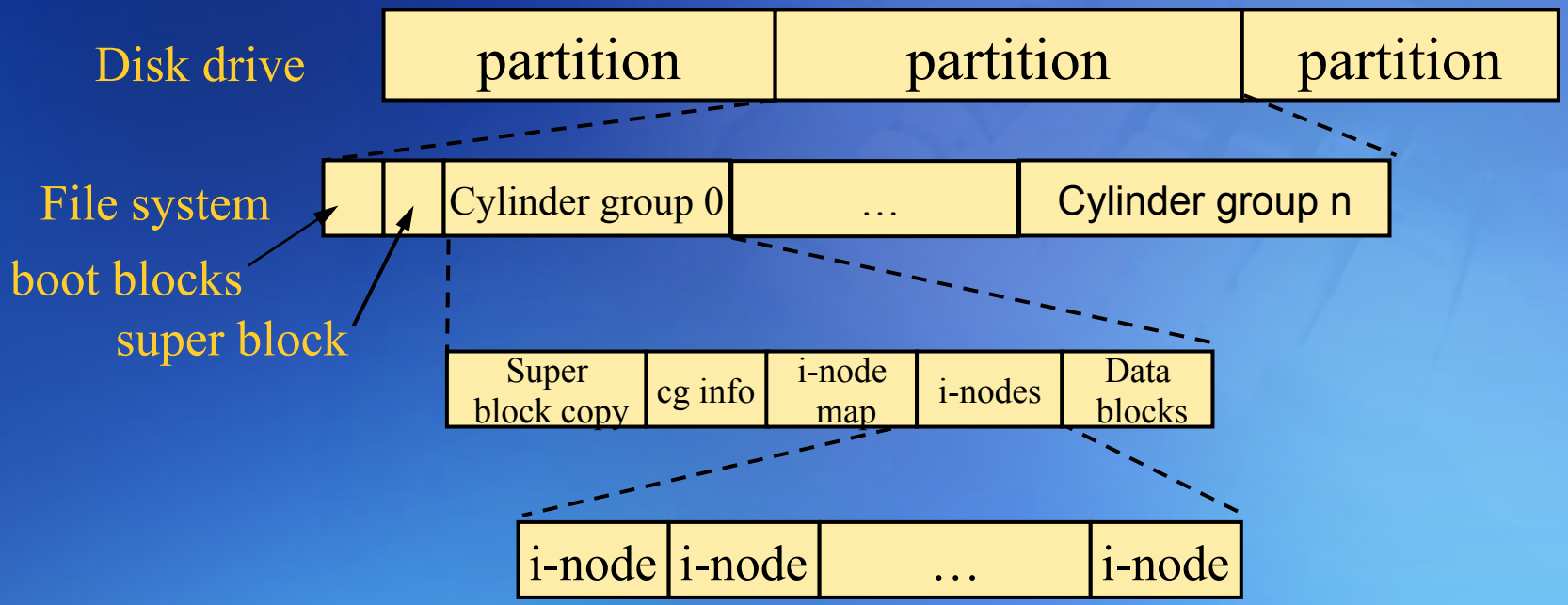
- **File**
  - A sequence of bytes
- **Directory**
  - A file that includes info on how to find other files.



* Use command "mount" to show all mounted file systems!

File Systems

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

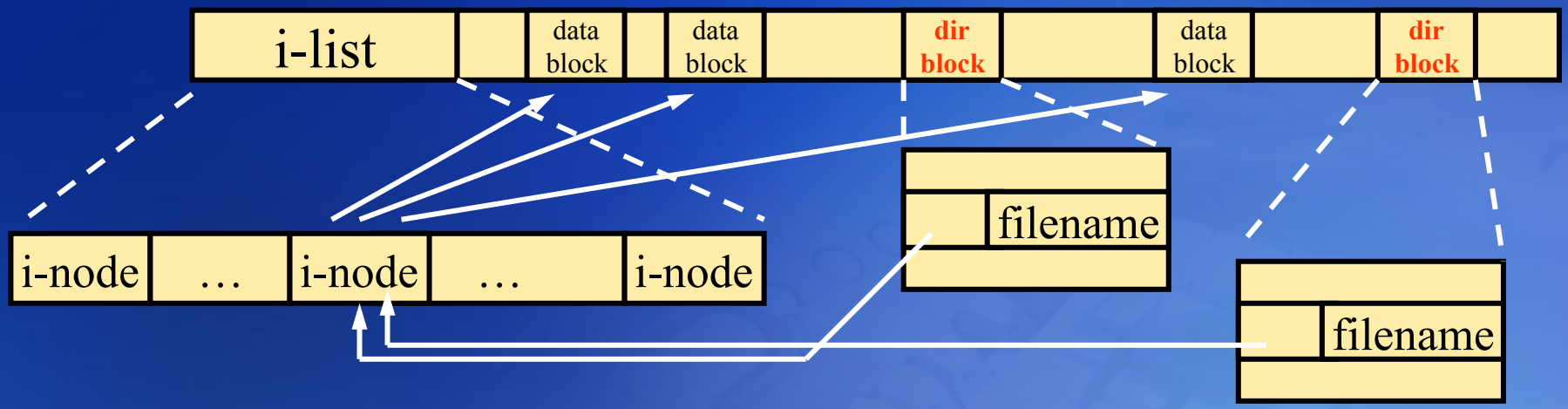


Partition v.s. Physical Devices

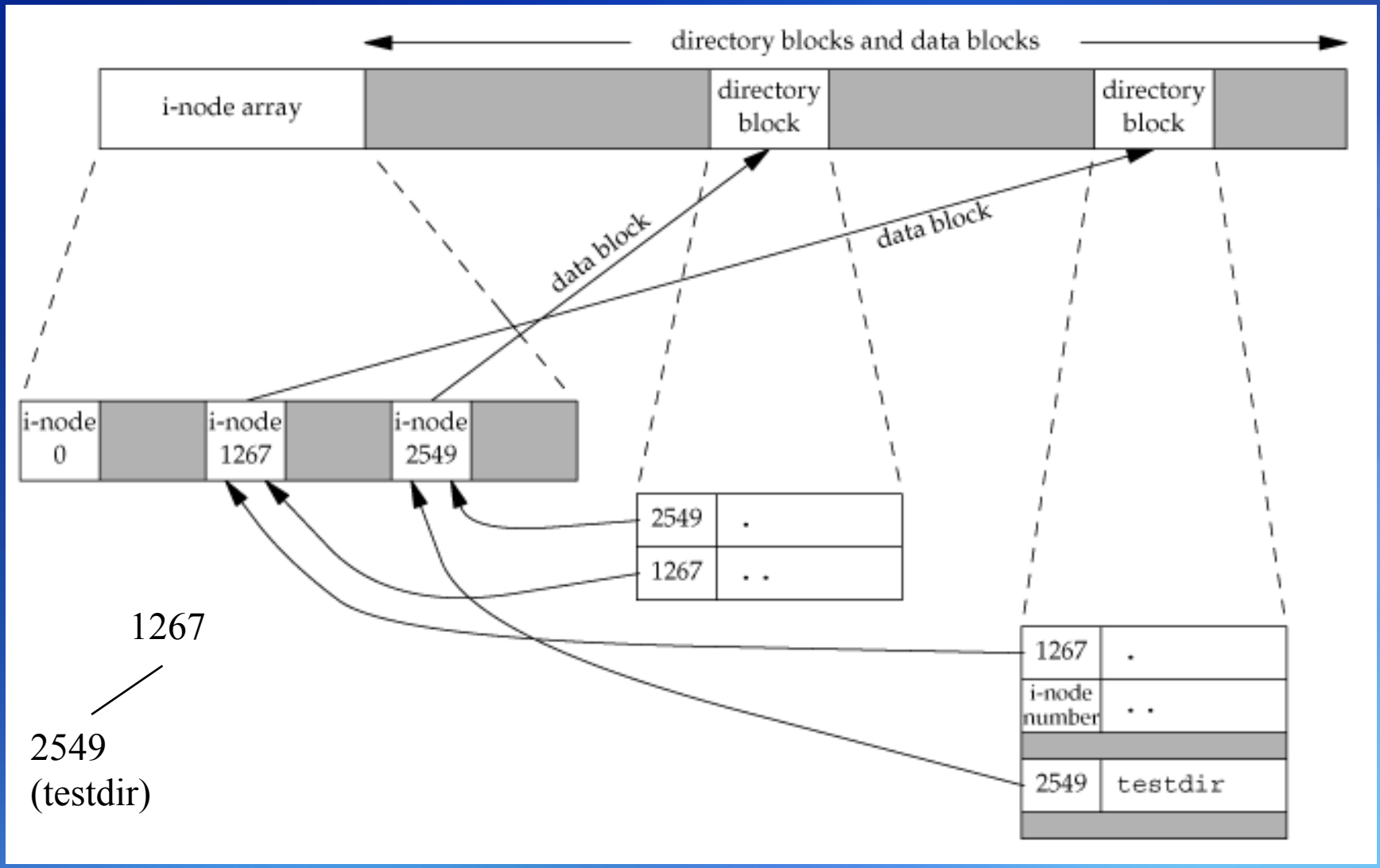
```
Disk /dev/hda: 123.5 GB, 123522416640 bytes
255 heads, 63 sectors/track, 15017 cylinders

   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
```

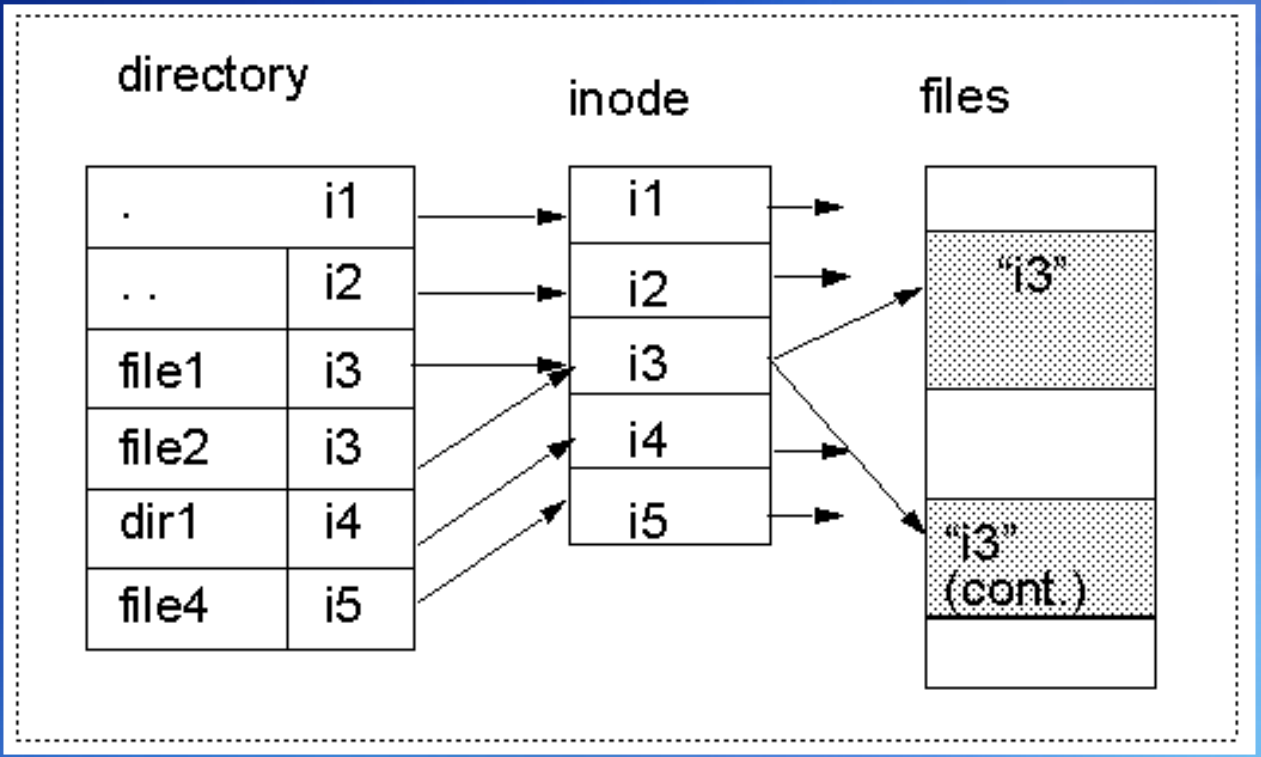
i-node and data blocks



- **i-node (fixed size) :**
  - Version 7: 64B, 4.3+BSD:128B, S5:64B, UFS:128B
  - File type, access permission, file size, data blocks, etc.
  - Predefined number of files that can be created.
  - It can happen that there is enough size, but i-node table is full.
  - ls –i filename (show i-node number)
- **Link count – hard links**
  - st_nlink in stat, LINK_MAX in POSIX.1
  - Unlink/link a file


directory blocks and data blocks
i-node array
directory block
directory block
data block
data block
i-node 0
i-node 1267
i-node 2549
2549 .
1267 . .
1267
2549
(testdir)
1267 .
i-node number . .
2549 testdir

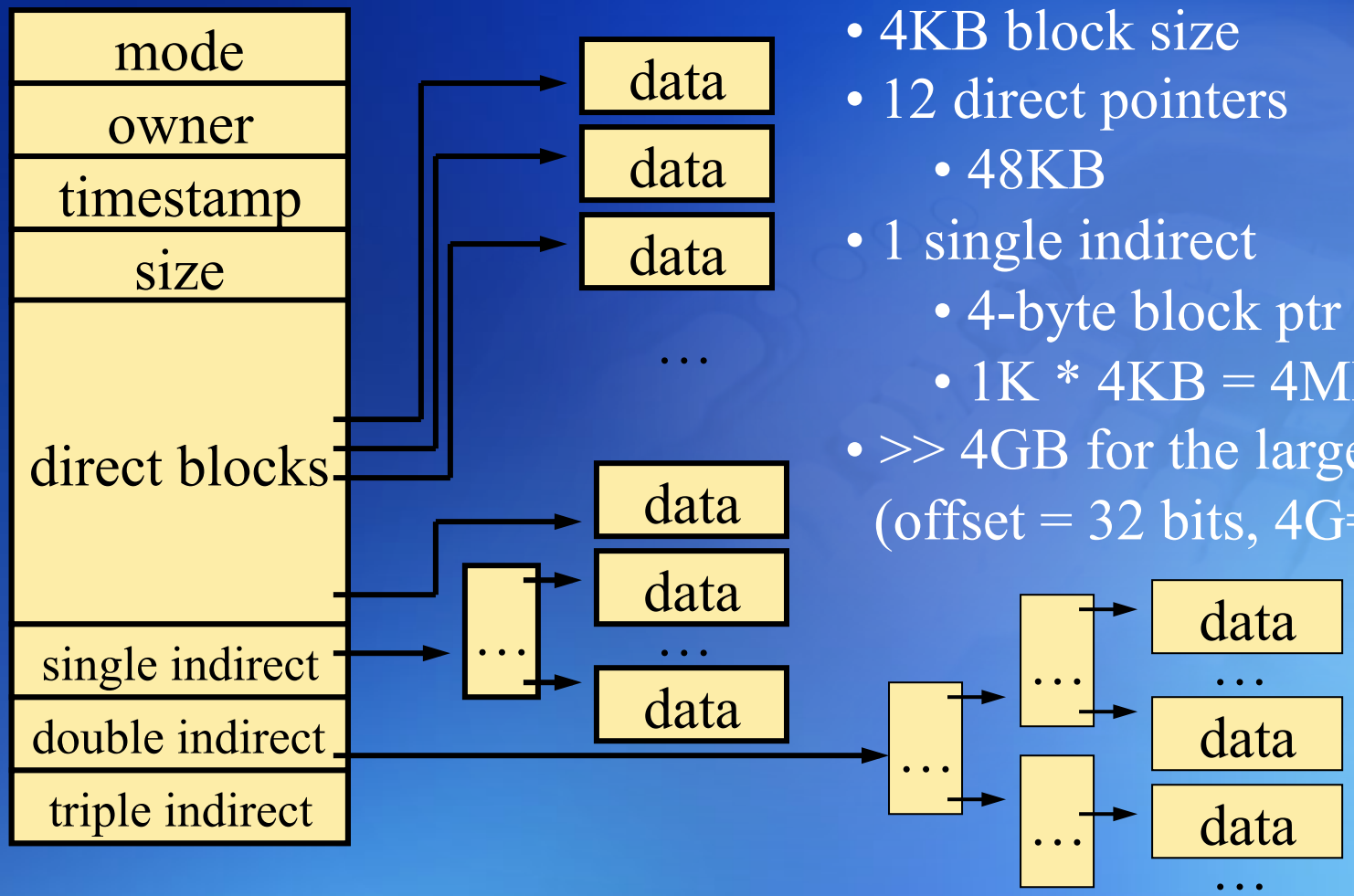
Relations between Directories and i-nodes



Moving files among directory

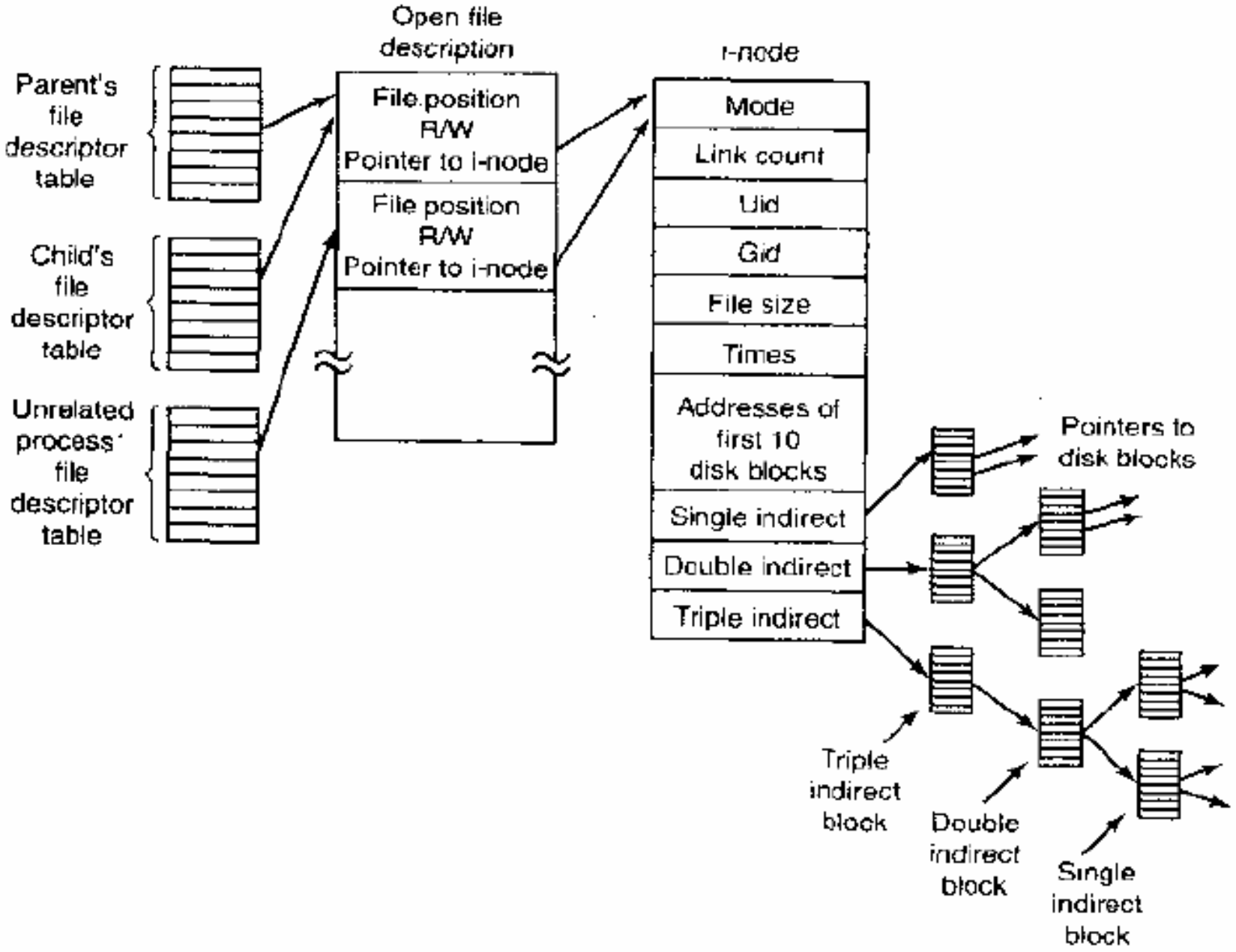
- **When files are moved in the same mounted file system, no need to physically move the file.**
  - Only directory block is updated.
- **Other, the file has to be physically moved.**

File System – 4.4BSD i-node



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

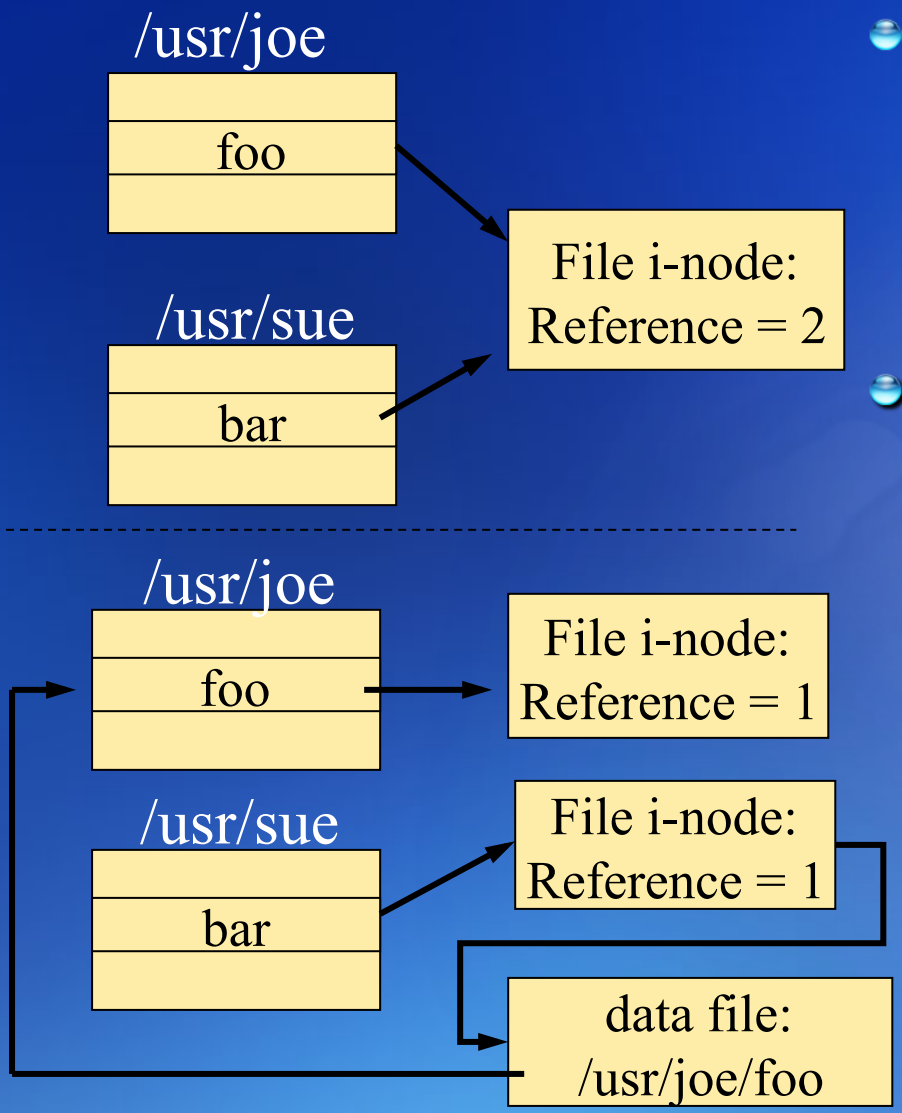
* "Operating system concept", Silberschatz and Galvin, Addison Wesley, pp. 380.


Open file description
i-node
Parent's file descriptor table
Child's file descriptor table
Unrelated process' file descriptor table
File position
R/W
Pointer to i-node
File position
R/W
Pointer to i-node
Mode
Link count
Uid
Gid
File size
Times
Addresses of first 10 disk blocks
Single indirect
Double indirect
Triple indirect
Pointers to disk blocks
Triple indirect block
Double indirect block
Single indirect block

Hard Link v.s. 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)**
  - 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?**

Sharing of Files



- **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**

Functions – link, unlink, rename, remove

- **int link(const char *existingpath, const char *newpath)**
  - Creation of the new dir entry and increment of the link count must be an atomic operation
- **int unlink(const char *pathname)**
  - 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, unlink references the symbolic link.
  - Checking if any process has the file open

Example of unlink()

```
int
main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("open error");
    if (unlink("tempfile") < 0)
        err_sys("unlink error");
    printf("file unlinked\n");
    sleep(15);
    printf("done\n");
    exit(0);
}
```

- A way to ensure that a temporary file won't be left in case the program crashes

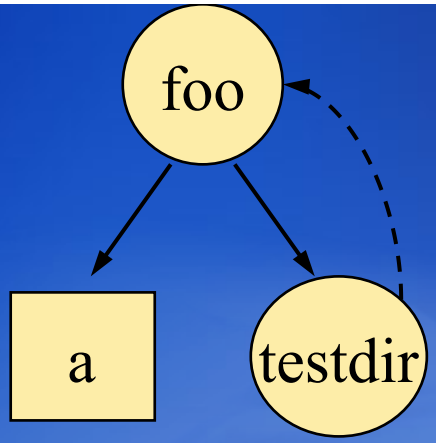
```
$ ls -l tempfile                    look at how big the file is
-rw-r----- 1 sar     413265408 Jan 21 07:14 tempfile
$ df /home                          check how much free space is available
Filesystem  1K-blocks     Used  Available  Use%  Mounted  on
/dev/hda4    11021440  1956332    9065108   18%  /home
$ ./a.out &                         run the program in Figure 4.16 in the background
1364                                the shell prints its process ID
$ file unlinked                     the file is unlinked
ls -l tempfile                      see if the filename is still there
ls: tempfile: No such file or directory          the directory entry is gone
$ df /home                          see if the space is available yet
Filesystem  1K-blocks     Used  Available  Use%  Mounted  on
/dev/hda4    11021440  1956332    9065108   18%  /home
$ done                              the program is done, all open files are closed
df /home                            now the disk space should be available
Filesystem  1K-blocks     Used  Available  Use%  Mounted on
/dev/hda4    11021440  1552352    9469088   15%  /home
                                    now the 394.1 MB of disk space are available
```

Functions – link, unlink, rename, 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
  - If *oldname* refers to a file:
    - if *newname* exists and it is not a directory, it's removed and *oldname* is renamed *newname*
    - if *newname* exists and it is a directory, an error results
    - must have w+x perms for the directories containing oldname and newname
  - If *oldname* refers to a directory:
    - if *newname* exists and is an empty directory (contains only . and ..), it is removed; *oldname* is renamed *newname*
    - if *newname* exists and is a file, an error results
    - if *oldname* is a prefix of *newname*, an error results
    - must have w+x perms for the directories containing *oldname* and *newname*

Symbolic Links



- **Goal**
  - Get around the limitations of hard links: (a) file system boundary (b) link to a dir.
  - Initially introduced by 4.2BSD

```
#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.

Functions follow symbolic links or not

Function	Does not follow symbolic link	Follows symbolic link
access		•
chdir		•
chmod		•
chown	•	•
creat		•
exec		•
lchown	•	
link		•
lstat	•	
open		•
opendir		•
pathconf		•
readlink	•	
remove	•	
rename	•	
stat		•
truncate		•
unlink	•	

No functions take filedes as input

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2006
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

File Times

- **Three Time Fields:**

Field	Description	Example	ls-option
st_atime	last-access-time	read	-u
st_mtime	last-modification-time	write	default
st_ctime	last-i-node-change-time	chmod, chown	-c

- **Changing the access permissions, user ID, link count, etc, only affects the i-node!**
- **ctime is modified automatically! (stat & access are for reading)**

Effect of functions on times

Function	Referenced file or directory			Parent directory of referenced file or directory			Section	Note
	a	m	c	a	m	c		
chmod, fchmod			•				4.9	
chown, fchown			•				4.11	
creat	•	•	•		•	•	3.4	O_CREAT new file
creat		•	•				3.4	O_TRUNC existing file
exec	•						8.10	
lchown			•				4.11	
link			•		•	•	4.15	parent of second argument
mkdir	•	•	•		•	•	4.20	
mkfifo	•	•	•		•	•	15.5	
open	•	•	•		•	•	3.3	O_CREAT new file
open		•	•				3.3	O_TRUNC existing file
pipe	•	•	•				15.2	

Function	Referenced file or directory			Parent directory of referenced file or directory			Section	Note
	a	m	c	a	m	c		
read	•						3.7	
remove			•		•	•	4.15	remove file = unlink
remove					•	•	4.15	remove directory = rmdir
rename			•		•	•	4.15	for both arguments
rmdir					•	•	4.20	
truncate, ftruncate		•	•				4.13	
unlink			•		•	•	4.15	
utime	•	•	•				4.19	
write		•	•				3.8	

Example: reading/writing a file only affects the file, instead of the residing dir

File Times

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

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

int utime(const char *pathname, const struct utimbuf *times);

- 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.

Year 2038 problem

- On most 32-bit systems, the *time_t* data type used to store the UNIX time number is a signed 32-bit integer.
- Being 32 bits means that *time_t* covers a range of about 136 years in total
- The minimum representable time is 20:45:52 UTC on Dec. 13, 1901.
- The maximum representable time is 03:14:07 UTC on Jan. 19, 2038.
- Solutions: unsigned? 64-bit?

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Example of utime()

```
int
main(int argc, char *argv[])
{
    int             i, fd;
    struct stat     statbuf;
    struct utimbuf  timebuf;

    for (i = 1; i < argc; i++) {
        if (stat(argv[i], &statbuf) < 0) { /* fetch current times */
            err_ret("%s: stat error", argv[i]);
            continue;
        }
        if ((fd = open(argv[i], O_RDWR | O_TRUNC)) < 0) { /* truncate */
            err_ret("%s: open error", argv[i]);
            continue;

        }
        close(fd);
        timebuf.actime  =  statbuf.st_atime;
        timebuf.modtime =  statbuf.st_mtime;
        if (utime(argv[i], &timebuf) < 0) {     /* reset times */
            err_ret("%s: utime error", argv[i]);
            continue;
        }
    }
    exit(0);
}
```

```
$ ls -l changemod times              look at sizes and last-modification times
-rwxrwxr-x 1 sar    15019    Nov  18   18:53   changemod
-rwxrwxr-x 1 sar    16172    Nov  19   20:05   times
$ ls -lu changemod times      look at last-access times
-rwxrwxr-x 1 sar    15019    Nov  18   18:53   changemod
-rwxrwxr-x 1 sar    16172    Nov  19   20:05   times
$ date              print today's date
Thu Jan 22 06:55:17 EST 2004
$ ./a.out changemod times     run the program in Figure 4.21
$ ls -l changemod times                and check the results
-rwxrwxr-x 1 sar         0   Nov  18   18:53   changemod
-rwxrwxr-x 1 sar         0   Nov  19   20:05   times
$ ls -lu changemod times             check the last-access times also
-rwxrwxr-x 1 sar         0   Nov  18   18:53   changemod
-rwxrwxr-x 1 sar         0   Nov  19   20:05   times
$ ls -lc changemod times             and the changed-status times
-rwxrwxr-x 1 sar         0   Jan  22   06:55   changemod
-rwxrwxr-x 1 sar         0   Jan  22   06:55   times
```

Functions – mkdir and rmdir

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

- umask, UID/GID setup (the same with file)
- . and .. are automatically created

```
#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!
- Implementation-dependent!

Functions – opendir, readdir, rewinddir, closedir

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

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

- **ftw/nftw**
  - recursively traversing the file system

NAME

ftw, nftw - file tree walk

SYNOPSIS

```
#include <ftw.h>

int ftw(const char *dirpath,
        int (*fn) (const char *fpath, const struct stat *sb,
                   int typeflag),
        int nopenfd);

#define _XOPEN_SOURCE 500   /* See feature_test_macros(7) */
#include <ftw.h>

int nftw(const char *dirpath,
        int (*fn) (const char *fpath, const struct stat *sb,
                   int typeflag, struct FTW *ftwbuf),
        int nopenfd, int flags);
```

DESCRIPTION

ftw() walks through the directory tree that is located under the directory dirpath, and calls fn() once for each entry in the tree. By default, directories are handled before the files and subdirectories they contain (preorder traversal).

To avoid using up all of the calling process's file descriptors, nopenfd specifies the maximum number of directories that **ftw**() will hold open simultaneously. When the search depth exceeds this, **ftw**() will become slower because directories have to be closed and reopened. **ftw**() uses at most one file descriptor for each level in the directory tree.

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!

- **cd is a built-in command**

Example of chdir()

```
int
main(void)
{

      if (chdir("/tmp") < 0)
          err_sys("chdir failed");
      printf("chdir to /tmp succeeded\n");
      exit(0);
}
```

```
$ pwd
/usr/lib
$ mycd
chdir to /tmp succeeded
$ pwd
/usr/lib
```

Functions – chdir, fchdir, getcwd

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

- The buffer must be large enough, or an error returns!
- chdir follows symbolic links, and getcwd has not idea of symbolic links!

Example of getcwd()

```
int
main(void)
{

    char    *ptr;
    int     size;

    if (chdir("/usr/spool/uucppublic") < 0)
        err_sys("chdir failed");

    ptr = path_alloc(&size); /* our own function */
    if (getcwd(ptr, size) == NULL)
        err_sys("getcwd failed");

    printf("cwd = %s\n", ptr);
    exit(0);
}
```

```
$ ./a.out
cwd = /var/spool/uucppublic
$ ls -l /usr/spool
lrwxrwxrwx 1 root 12 Jan 31 07:57 /usr/spool -> ../var/spool
```

Example of Directory Traversal

```
struct stat     statbuf;
struct dirent   *dirp;
DIR             *dp;
char            *ptr;

ptr = fullpath + strlen(fullpath)     // point to end of fullpath
*ptr++ = '/';
*ptr = 0;
dp = opendir(fullpath);                // read directory
while ((dirp = readdir(dp)) != NULL) {
    if (strcmp(dirp->d_name, ".") == 0 ||
        strcmp(dirp->d_name, "..") == 0)
            continue;                  // ignore dot and dot-dot

    strcpy(ptr, dirp->d_name);         // append name after slash

    lstat(fullpath, &statbuf);

}
ptr[-1] = 0;     // erase everything from slash onwards
closedir(dp);
```

Limits: ANSI C, POSIX, XPG3, FIPS

- **Compiler-time options and limits (headers)**
  - The largest value of a short?
  - Array bounds at compile time
- **Run-time limits**
  - Related to file/dir
    - pathconf and fpathconf, e.g., the max # of bytes in a filename
  - Not related to file/dir
    - sysconf, e.g., the max # of opened files per process
- **Some of the Run-time limits can be indeterminate! E.g., _SC_CHILD_MAX**
- **Remark: implementation-related**

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Run-Time Limits

- **#include <unistd.h>**
- **long sysconf(int *name*);**
  - _SC_CHILD_MAX, _SC_OPEN_MAX, etc.
- **long pathconf(const char **pathname*, int *name*);**
- **long fpathconf(int **filedes*, int *name*);**
  - _PC_LINK_MAX, _PC_PATH_MAX, _PC_PIPE_BUF, _PC_NAME_MAX, etc.
- **Various *names* and restrictions on arguments**
- **Return –1 and set errno if any error occurs.**
  - EINVAL if the name is incorrect.

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Example of Indeterminate Run-Time Limits

```
#ifdef  PATH_MAX
static int      pathmax = PATH_MAX;
#else
static int      pathmax = 0;
#endif

#define PATH_MAX_GUESS  1024    /* if PATH_MAX is indeterminate */
                                /* we're not guaranteed this is adequate */
char *
path_alloc(int *size)
                                /* also return allocated size, if nonnull */
{
        char    *ptr;

        if (pathmax == 0) {             /* first time through */
                errno = 0;
                if ( (pathmax = pathconf("/", _PC_PATH_MAX)) < 0) {
                        if (errno == 0)
                                pathmax = PATH_MAX_GUESS;       /* it's indeterminate */
                        else
                                err_sys("pathconf error for _PC_PATH_MAX");
                } else
                        pathmax++;              /* add one since it's relative to root */
        }

        if ( (ptr = malloc(pathmax + 1)) == NULL)
                err_sys("malloc error for pathname");

        if (size != NULL)
                *size = pathmax + 1;
        return(ptr);
}
```

`st_ino` (I-node number)

- Each file has a unique *i-node number* (index number).
- The i-node number can be used to look up a file's information (*i-node*) in a system table (the *i-list*).
- A file's i-node contains:
  - user and group ids of its owner
  - permission bits
  - etc.

What You Should Know

- **Properties of Files and Directories**
- **Access Permission**
  - Set-User/Group-ID and Sticky bit
  - Real-User/Group-ID
- **File Systems**
  - I-nodes
  - Hard link vs. Symbolic link
  - Directory traversal
- **Built-in Commands**
  - umask, cd
- **Removing Opened Files/Directories**