Files and Directories

‘15H2

Inshik Song

stat(2) family of functions

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

int stat(const char *restrict pathname, struct stat *restrict buf);

int fstat(int fd, struct stat *buf);

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

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

All four return: 0 if OK, –1 on error

- All these functions return extended attributes about the referenced file (in the case of symbolic links, lstat(2) returns attributes of the link, others return stats of the referenced file).
  - stat : returns a structure of information about the named file
  - fstat : obtains information about the file that is already open on the descriptor fd
  - lstat : returns information about the symbolic link, not the file referenced by the symbolic link
  - fstatat : provides a way to return the file statistics for a pathname relative to an open directory represented by the fd argument

File attributes: struct stat

```
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 */
  struct timespec  st_atim;    /* time of last access */
  struct timespec  st_mtim;    /* time of last modification */
  struct timespec  st_ctim;    /* time of last file status change */
  blksize_t        st_blksize; /* best I/O block size */
  blkcnt_t         st_blocks;  /* number of disk blocks allocated */
};
```

*) The definition of the structure can differ among implementations

File types

- The st_mode field of the struct stat encodes the type of file:
  - regular – most common, interpretation of data is up to application
  - directory – contains names of other files and pointer to information on those files. Any process can read, only kernel can write.
  - character special – used for certain types of devices
  - block special – used for disk devices (typically). All devices are either character or block special.
  - FIFO – used for interprocess communication (sometimes called named pipe)
  - socket – used for network communication and non-network communication (same host).
  - symbolic link – Points to another file.
- Find out more in <sys/stat.h>.

File types

- Macros used to determine the file type:

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

- Macros used to determine the type of IPC object:

Macro	Type of object
S_TYPEISMQ()	message queue
S_TYPEISSEM()	semaphore
S_TYPEISSHM()	shared memory object

```
#include "apue.h"

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/log /dev/tty \
> /var/lib/oprofile/opd_pipe /dev/sr0 /dev/cdrom
/etc/passwd: regular
/etc: directory
/dev/log: socket
/dev/tty: character special
/var/lib/oprofile/opd_pipe: fifo
/dev/sr0: block special
/dev/cdrom: symbolic link
```

Counts and percentages of different file types

File type	Count	Percentage
regular file	415,803	79.77 %
directory	62,197	11.93
symbolic link	40,018	8.25
character special	155	0.03
block special	47	0.01
socket	45	0.01
FIFO	0	0.00

Set-User-ID and Set-Group-ID

- Every process has six or more IDs associated with it:

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

- Whenever a file is setuid, set the effective user ID to st_uid. Whenever a file is setgid, set the effective group ID to st_gid.
- As an example, the UNIX System program that allows anyone to change his or her password, passwd(1), is a set-user-ID program. This is required so that the program can write the new password to the password file, typically either /etc/passwd or /etc/shadow, files that should be writable only by the superuser

File Access Permissions

- st_mode also encodes the file access permissions (S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH). Uses of the permissions are summarized as follows:
  - To open a file, need execute permission on each directory component of the path
  - To open a file with O_RDONLY or O_RDWR, need read permission
  - To open a file with O_WRONLY or O_RDWR, need write permission
  - To use O_TRUNC, must have write permission
  - 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
  - To execute a file (via exec family), need execute permission

File Access Permissions

- Which permission set to use is determined (in order listed):
  1. If effective-uid == 0, grant access
  2. If effective-uid == st_uid
     2.1. if appropriate user permission bit is set, grant access
     2.2. else, deny access
  3. If effective-gid == st_gid
     3.1. if appropriate group permission bit is set, grant access
     3.2. else, deny access
  4. If appropriate other permission bit is set, grant access, else deny access

Ownership of New Files and Directories

- st_uid = effective-uid
- st_gid = …either:
  - effective-gid of process
  - gid of directory in which it is being created

access(2)

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

Both return: 0 if OK, –1 on error

- Tests file accessibility on the basis of the real uid and gid. Allows setuid/setgid programs to see if the real user could access the file without it having to drop permissions to do so.
- The mode parameter can be a bitwise OR of:
  - 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

```
#include "apue.h"
#include <fcntl.h>

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
```

umask(2)

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

mode_t umask(mode_t cmask);
```

Returns: previous file mode creation mask

- umask(2) sets the file creation mode mask. Any bits that are on in the file creation mask are turned off in the file's mode.
- Important because a user can set a default umask. If a program needs to be able to insure certain permissions on a file, it may need to turn off (or modify) the umask, which affects only the current process.

```c
#include "apue.h"
#include <fcntl.h>

#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
```

chmod(2), fchmod(2) and fchmodat(2)

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

int chmod(const char *pathname, mode_t mode);

int fchmod(int fd, mode_t mode);

int fchmodat(int fd, const char *pathname, mode_t mode, int flag);
```

All three return: 0 if OK, –1 on error

- Changes the permission bits on the file. Must be either superuser or effective uid == st_uid. mode can be any of the bits from our discussion of st_mode as well as:
  - S_ISUID – setuid
  - S_ISGID – setgid
  - S_ISVTX – sticky bit (aka “saved text”)
  - S_IRWXU – user read, write and execute
  - S_IRWXG – group read, write and execute
  - S_IRWXO – other read, write and execute

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

```
#include "apue.h"

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
```

Sticky bit

- In the early UNIX systems, the sticky bit is used to save the text images in the swap area when the process terminated.
- On contemporary systems, the use of the sticky bit has been extended. The Single UNIX Specification allows the sticky bit to be set for a directory. 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 meets one of the following criteria:
  - Owns the file
  - Owns the directory
  - Is the superuser
- The directories /tmp and /var/tmp are typical candidates for the sticky bit—they are directories in which any user can typically create files. The permissions for these two directories are often read, write, and execute for everyone (user, group, and other). But users should not be able to delete or rename files owned by others.

chown(2), fchown(2), fchownat(2) and lchown(2)

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);

int fchown(int fd, uid_t owner, gid_t group);

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 group);
```

All four return: 0 if OK, –1 on error

- Changes st_uid and st_gid for a file. For BSD, must be superuser.
- Some SVR4's let users chown files they own. POSIX.1 allows either depending on _POSIX_CHOWN_RESTRICTED (a kernel constant).
- owner or group can be -1 to indicate that it should remain the same.
- Non-superusers can change the st_gid field if both:
  - effective-user ID == st_uid and
  - owner == file's user ID and group == effective-group ID (or one of the supplementary group IDs)
- chown and friends clear all setuid or setgid bits.

File Size

- The st_size member of the stat structure contains the size of the file in bytes. This field is meaningful only for regular files, directories, and symbolic links.
  - For a regular file, a file size of 0 is allowed.
  - For a symbolic link, the file size is the number of bytes in the filename.

Holes in a File

```
$ ls -l core
-rw-r--r--  1 sar   8483248 Nov 18 12:18 core
$ du -s core
272     core
```

- The size of the file core is slightly more than 8 MB, yet the du command reports that the amount of disk space used by the file is 272 512-byte blocks (139,264 bytes).
- Obviously, this file has many holes.

```
$ 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
```

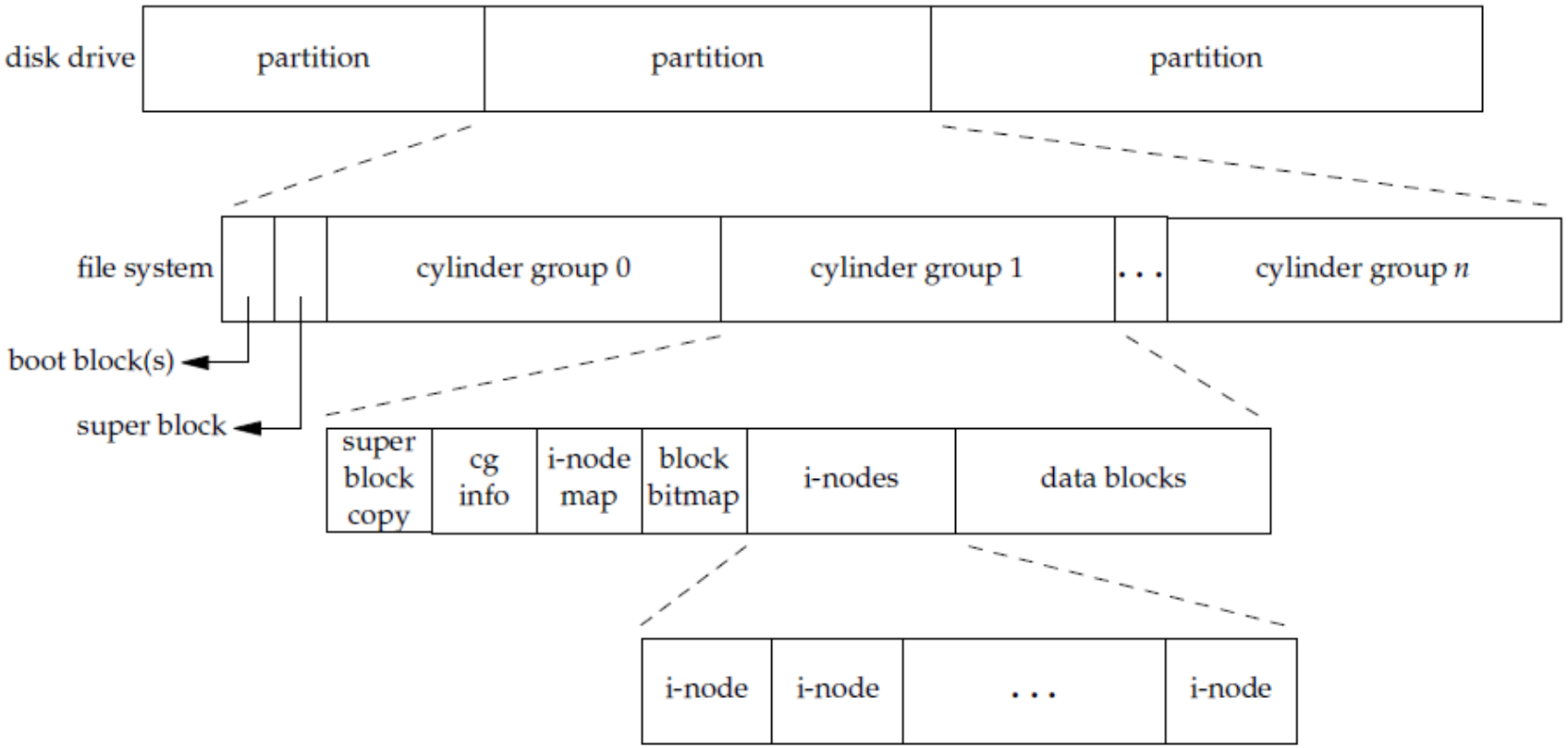
File Truncation

```
#include <unistd.h>

int truncate(const char *pathname, off_t length);

int ftruncate(int fd, off_t length);
```

Both return: 0 if OK, –1 on error

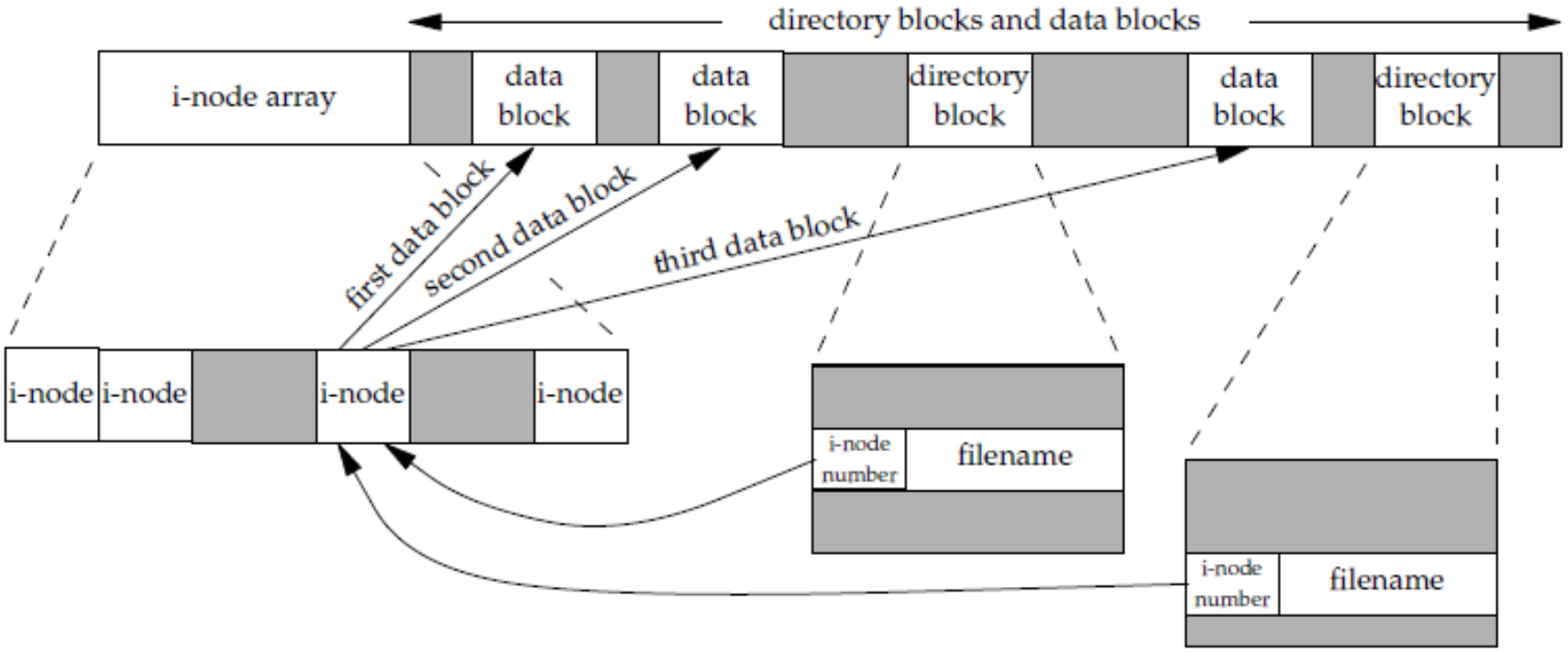
- Truncate an existing file to *length* bytes.
  - If the previous size of the file was greater than *length*, the data beyond *length* is no longer accessible.
  - Otherwise, if the previous size was less than *length*, the file size will increase and the data between the old end of file and the new end of file will read as 0 (i.e., a hole is probably created in the file).
- Use ftruncate when we need to empty a file after obtaining a lock on the file.

File Systems



File Systems

- A directory entry is really just a hard link mapping a “filename” to an inode
- You can have many such mappings to the same file



Directories

- Directories are special "files" containing hardlinks
- Each directory contains at least two entries:
  - . (this directory)
  - .. (the parent directory)
- The link count (st_nlink) of a directory is at least 2