

stat(), fstat() and lstat() file information

- int stat(const char *file_name, struct stat *buf);
- int fstat(int filedes, struct stat *buf);
- int lstat(const char *file_name, struct stat *buf);

```

struct stat {
    dev_t      st_dev;      /* device */
    ino_t      st_ino;     /* inode */
    mode_t     st_mode;    /* protection */
    nlink_t    st_nlink;   /* number of hard links */
    uid_t      st_uid;     /* user ID of owner */
    gid_t      st_gid;     /* group ID of owner */
    dev_t      st_rdev;    /* device type (if inode device) */
    off_t      st_size;    /* total size, in bytes */
    unsigned long st_blksize; /* blocksize for filesystem I/O */
    unsigned long st_blocks; /* number of blocks allocated */
    time_t     st_atime;   /* time of last access */
    time_t     st_mtime;   /* time of last modification */
    time_t     st_ctime;   /* time of last change */
};

```

stat() fstat() and lstat() (continued)

- stat() return information about the named file
- fstat() return information about an open file
- lstat() return information about the symbolic link, not the file referenced by the symbolic link

File types

- **Regular file** : The most common type of file, there is no distinction to the Unix kernel whether the data is text or binary
- **Directory file** : A file that contains the names of other files and pointers to information on these files
- **Character special file** : A type of file used for certain types of devices
- **Block special file** : A type of file used for disk devices
- **FIFO** : A type of file used for interprocess communication between processes (also known as named pipe)
- **Socket** : A type of file used for network communication between processes
- **Symbolic link** : A type of file that points to another file

File types (cont...)

- The type of a file is encoded in the **st_mode** member of the stat structure
- The following macros can be used to determine the file type:
 - S_ISREG(m) is it a regular file?
 - S_ISDIR(m) directory?
 - S_ISCHR(m) character device?
 - S_ISBLK(m) block device?
 - S_ISFIFO(m) fifo?
 - S_ISLNK(m) symbolic link? (Not in POSIX.1-1996.)
 - S_ISSOCK(m) socket? (Not in POSIX.1-1996.)

Set-User-ID and Set-Group-ID

- Every **process** has six IDs associated with it
 - real user ID who we really are
 - real group ID
 - effective user ID used for file access
 - effective Group
 - saved set-user-ID saved by exec function
 - saved set-group-ID
- Normally the effective user id equal the real user id
- Every file has an owner, the owner is specified by the **st_uid** member
- There is a capability to set a special flag in the file's mode word (**st_mode**) that says "when the file is executed set the effective uid of the process to the owner of the file"

Set-User-ID (cont..)

- This feature is useful when we want a user to execute a certain program in **super-user** permission
- For example passwd is a set-uid file

```

T1or@mos214:~/teach/os/samples> ls -la /usr/bin/passwd
-r-S--x--x 1 root  root  13476 Aug 7 2001
 /usr/bin/passwd*

```

- The set-user-id (set-group-id) bit are contained in the **st_mode** member of the stat structure
- The constant S_ISUID (S_ISGID) can be used to test this bit

File Access Permissions

- The *st_mode* also encode the file permissions bits
- from the shell the command `chmod` can be used to change file permissions
- whenever we want to open any type of file by name, we must have **execute** permissions on the **directories** in the name
- we cannot **create** a new file in a directory unless we have **write** and **execute** permissions in the directory
- to delete a file we need write and execute permission in the directory

st_mode mask	Meaning
S_IRUSR	User read
S_IWUSR	User write
S_IXUSR	User execute
S_IRGRP	Group read
S_IWGRP	Group write
S_IXGRP	Group execute
S_IROTH	Other read
S_IWOTH	Other write
S_IXOTH	Other execute

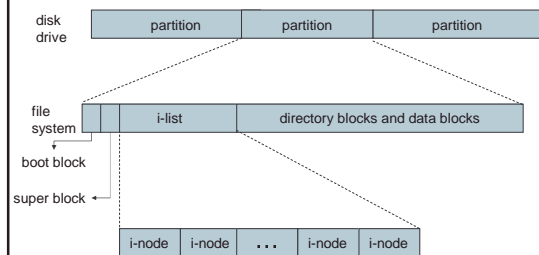
access()

- `int access(const char *pathname, int mode);`
- `access` checks whether the process would be allowed to read, write, execute the file or test for existence
- `mode` is a mask of one or more 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
- The check is done with the process's real uid and gid, rather than with the effective id's. This is to allow set-UID programs to easily determine the invoking user authority

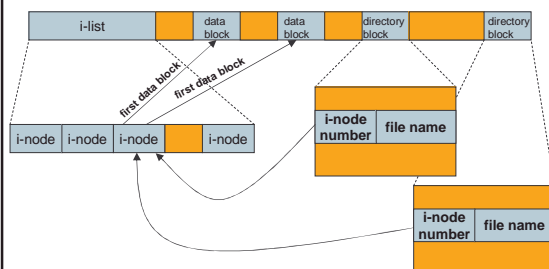
umask()

- `mode_t umask(mode_t cmask);`
- Every process has a **file mode creation mask**
- The `umask()` function set the **file mode creation mask** for the process and return the previous value
- The **file mode creation mask** is used whenever the process creates a new file or a new directory
- recall that the `open()` functions get a mode argument
- Any bits that are **ON** in the **file mode creation mask** are turned **Off** in the file's mode

File system structure



File System structure (cont...)



Hard Links

- i-nodes are fixed-length entries that contain most of the information about a file
- directory entries contain the i-node number and the file-name
- we can see 2 directory entries that point to the same i-node entry. This is a **hard link**
- from the shell the command `"ln"` can be used to create hard links
- Every i-node has a **link-count**, which is the number of directory entries that point to this i-node
- Only when the link count reach 0 the file can be deleted
- What is the initial link-count of a new empty directory??

link(), unlink() and rename()

- `int link(const char *oldpath, const char *newpath);`
- `int unlink(const char *pathname);`
- `int rename(const char *oldpath, const char *newpath);`
- `link()` create a new directory entry that point to the same inode `oldpath` is pointing
- The creation of the directory entry and the increment of the link count must be atomic why??
- `unlink()` remove a directory entry and decrement the i-node link count
- If the link count reach 0 the i-node is removed
- not exactly, only when no process holds the file open that file can be removed

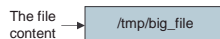
Temporary files

- A process can open a temporary file, unlink it and continue to work with it.
- The files data will be removed only when the process exit
- This feature can be used by programs to assure that temporary files won't be left around in case the program crashes

```
fd = open("/tmp/tmp_file", O_RDWR | O_CREAT | O_TRUNC, 0755);
if(fd == -1)
    SYS_ERROR("Error in open\n");
if(unlink("/tmp/tmp_file") == -1)
    SYS_ERROR("Error in unlink!");
the program continue to work with the file...
```

Symbolic Links

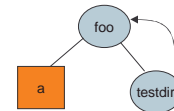
- Hard links have several limitations
 - hard links can not exists across file systems
 - Only the super user can create hard links to directories (in Linux even this is not possible)
- A symbolic link is an indirect pointer to a file



- When using functions that refer to files by name we always need to know whether the function follow a symbolic link or not (written in the function manually)

Symbolic Links (cont...)

- Functions that follow symbolic links
 - `access()`, `chdir()`, `exec()`, `link()`, `open()`, `stat()` ...
- Functions that does not follow symbolic links
 - `chown()`, `lstat()`, `rename()`, `unlink()`
- Symbolic links can exists to files that do not
 - `ln -s /no/such/file myfile` (this is legal)
- Symbolic links can point to directories and create loops
 - `mkdir foo`
 - `touch foo/a`
 - `ln -s ../foo foo/testdir`



Symbolic links (cont ...)

- `int symlink(const char *oldpath, const char *newpath);`
- `symlink()` is used to create symbolic links
- It is not required that `oldpath` exists when the symbolic link is created
- Also `oldpath` and `newpath` need not reside in the same file system
- `int readlink(const char *path, char *buf, size_t bufsiz);`
- Since open follow symbolic links we need a way to open the link itself and read the name in the link
- The function combine `open()`, `read()` and `close()`

File Times

- Tree time fields are maintained for each file

Field	Description	Example	ls option
<code>st_atime</code>	Last-access time of file data	<code>read</code>	<code>-u</code>
<code>st_mtime</code>	Last-modification time of file data	<code>write</code>	default
<code>st_ctime</code>	Last-change time of i-node status	<code>chmod</code>	<code>-c</code>

- Note the deference between `ctime` and `mtime`
- Which time field `access()` and `stat()` change???

utime()

- `int utime(const char *filename, struct utimbuf *buf);`
- `struct utimbuf {`
 - `time_t actime; /* access time */`
 - `time_t modtime; /* modification time */``};`
- `utime()` change the access and modification time of a file
- If `buf` is `NULL` the time is set to the current time
- There is no way to change `ctime`

mkdir() and rmdir()

- `int mkdir(const char *pathname, mode_t mode);`
- `mkdir()` created new empty directory
- `.` and `..` are automatically added to the new dir

- `int rmdir(const char *pathname);`
- An **empty** directory is deleted with `rmdir()`

chdir(), fchdir() and getcwd()

- Every process has a current working directory
- This directory is where the search for all relative pathnames starts
 - A relative path name is a path name that do not start with a `"/"`
- We can change the current working directory using
 - `int chdir(const char *pathname)`
 - `int fchdir(int filedes);`
- We can get the current working directory using
 - `char *getcwd(char *buf, size_t size);`

statfs() and fstatfs()

- `int statfs(const char *path, struct statfs *buf);`
- `int fstatfs(int fd, struct statfs *buf);`

- Those functions return information about the file system
- `struct statfs {`
 - `long f_type; /* type of filesystem (see below) */`
 - `long f_bsize; /* optimal transfer block size */`
 - `long f_blocks; /* total data blocks in file system */`
 - `long f_bfree; /* free blocks in fs */`
 - `long f_bavail; /* free blocks avail to non-superuser */`
 - `long f_files; /* total file nodes in file system */`
 - `long f_ffree; /* free file nodes in fs */`
 - `fsid_t f_fsid; /* file system id */`
 - `long f_namelen; /* maximum length of filenames */`
 - `long f_spare[6]; /* spare for later */``};`