

CS631 - Advanced Programming in the UNIX Environment

—

File Systems, System Data Files, Time & Date

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann

`jschauma@stevens.edu`

`http://www.cs.stevens.edu/~jschauma/631/`

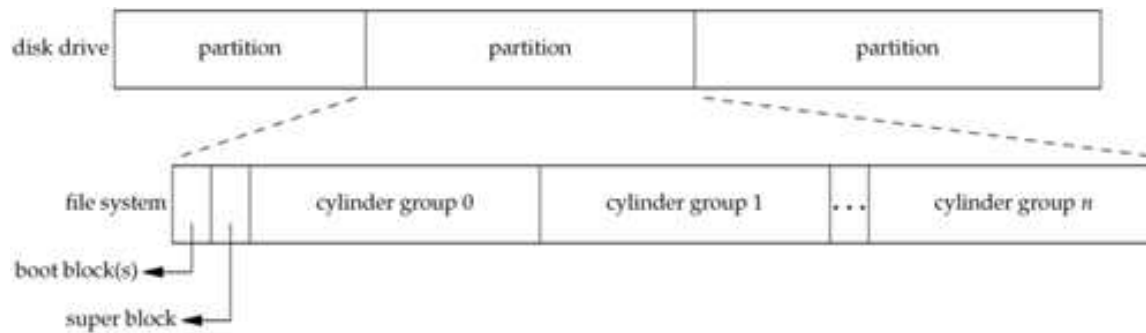
File Systems

- a disk can be divided into logical *partitions*



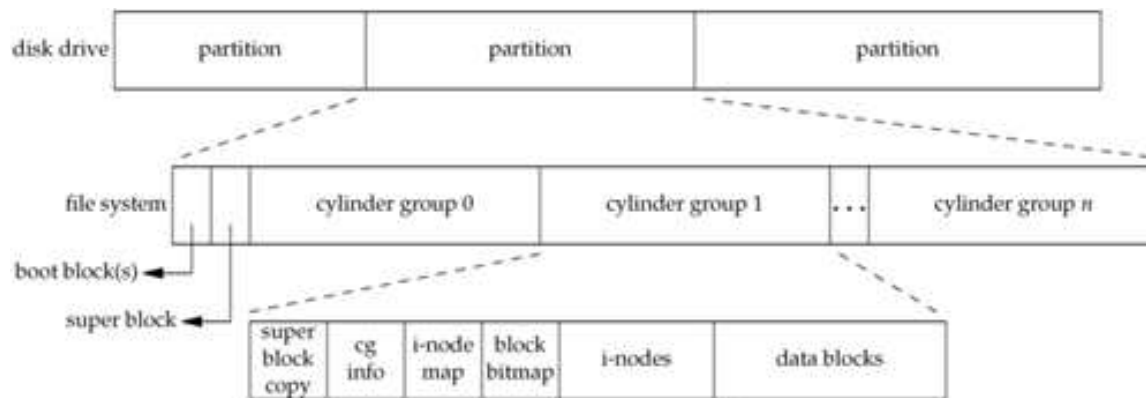
File Systems

- a disk can be divided into logical *partitions*
- each logical *partition* may be further divided into *file systems* containing *cylinder groups*



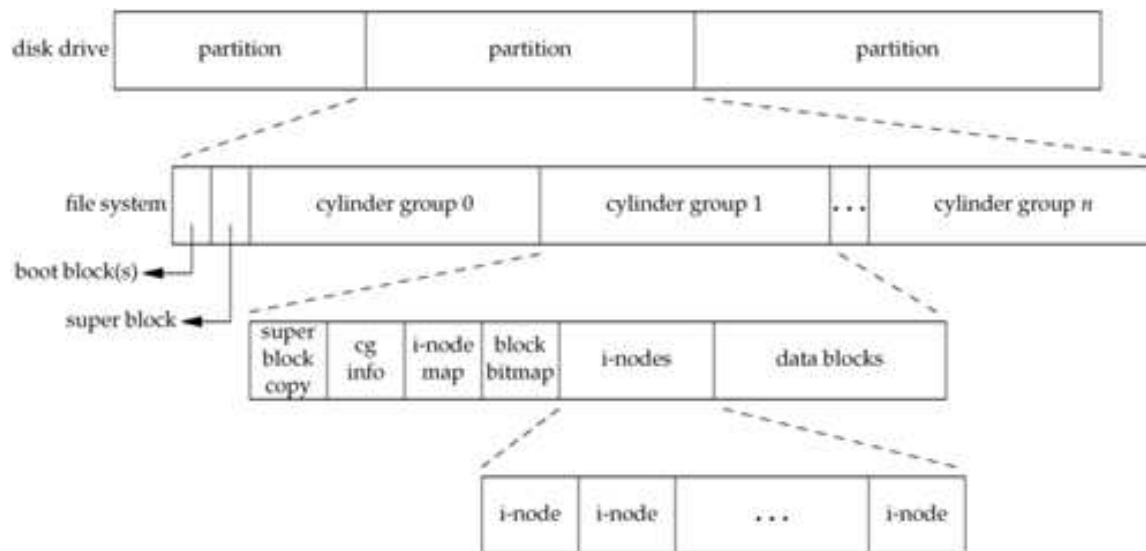
File Systems

- a disk can be divided into logical *partitions*
- each logical *partition* may be further divided into *file systems* containing *cylinder groups*
- each *cylinder group* contains a list of *inodes* (*i-list*) as well as the actual *directory*- and *data blocks*



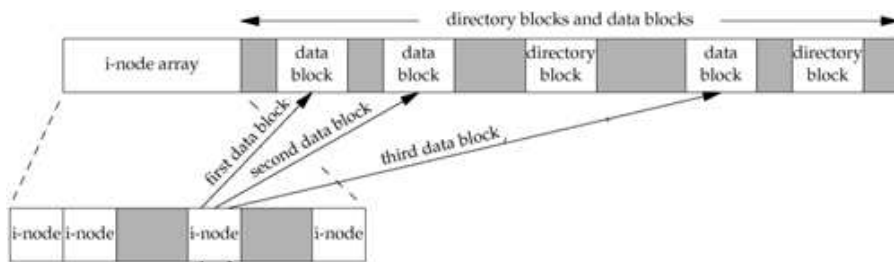
File Systems

- a disk can be divided into logical *partitions*
- each logical *partition* may be further divided into *file systems* containing *cylinder groups*
- each *cylinder group* contains a list of *inodes* (*i-list*) as well as the actual *directory*- and *data blocks*



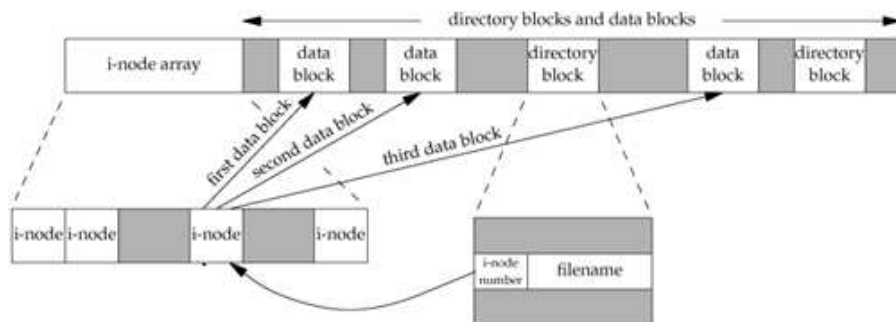
File Systems

- a disk can be divided into logical *partitions*
- each logical *partition* may be further divided into *file systems* containing *cylinder groups*
- each *cylinder group* contains a list of *inodes* (*i-list*) as well as the actual *directory-* and *data blocks*



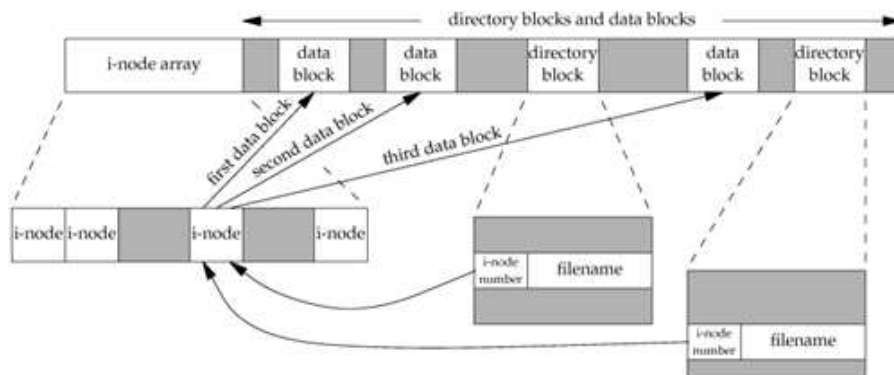
File Systems

- a disk can be divided into logical *partitions*
- each logical *partition* may be further divided into *file systems* containing *cylinder groups*
- each *cylinder group* contains a list of *inodes* (*i-list*) as well as the actual *directory*- and *data blocks*
- a directory entry is really just a *hard link* mapping a “filename” to an inode



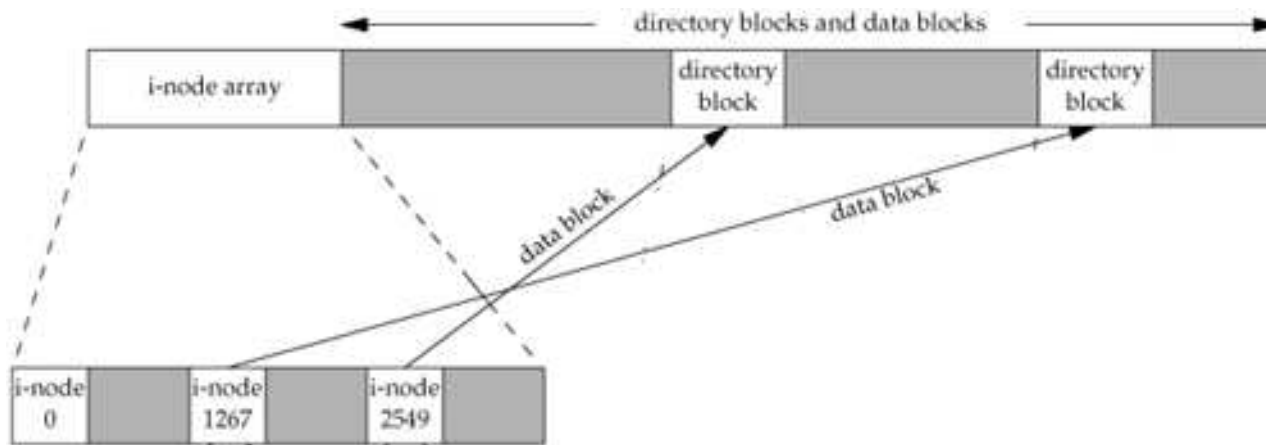
File Systems

- a disk can be divided into logical *partitions*
- each logical *partition* may be further divided into *file systems* containing *cylinder groups*
- each *cylinder group* contains a list of *inodes* (*i-list*) as well as the actual *directory*- and *data blocks*
- 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



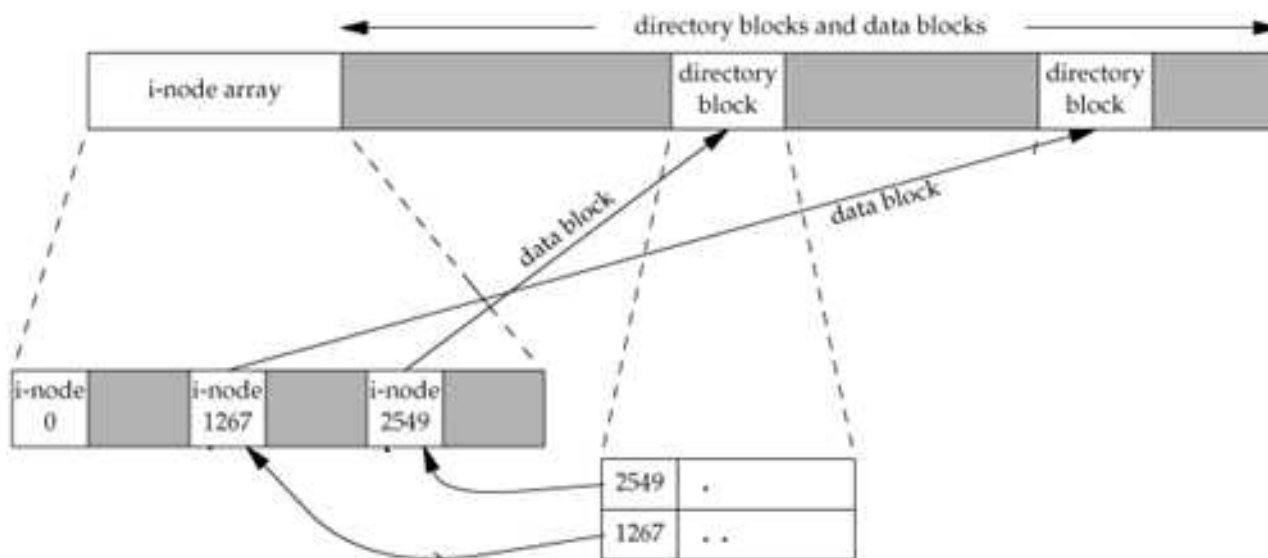
Directories

- directories are special "files" containing hardlinks



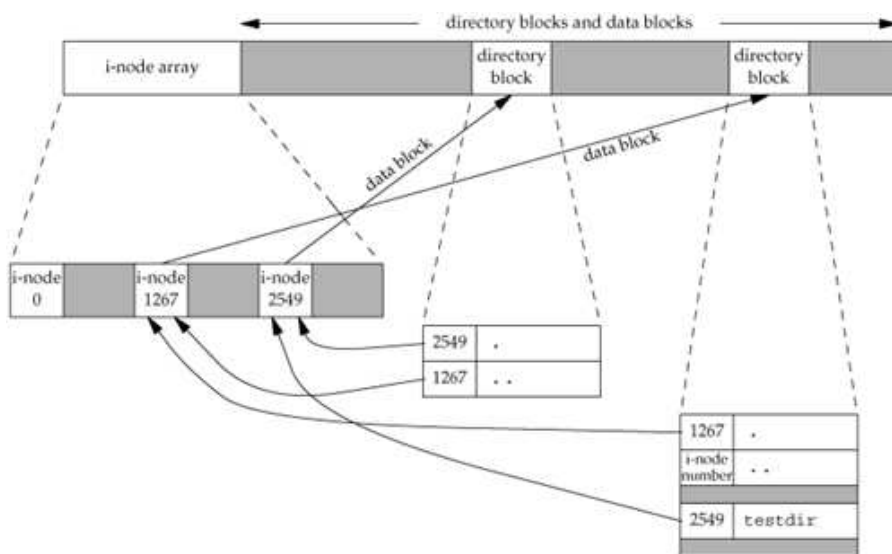
Directories

- directories are special "files" containing hardlinks
- each directory contains at least two entries:
 - . (*this* directory)
 - .. (the parent directory)



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



Inodes

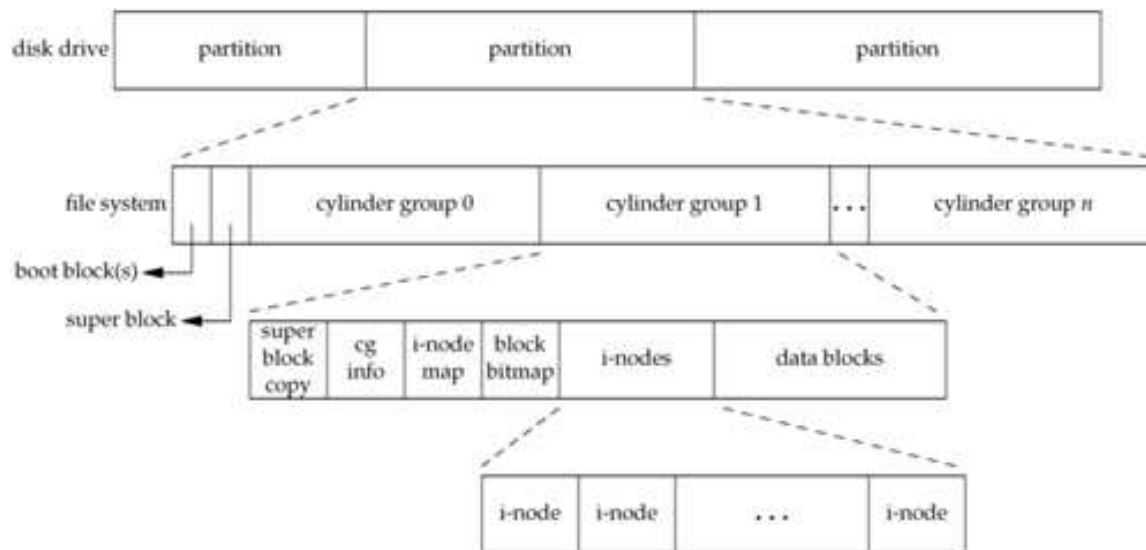
- the *inode* contains most of information found in the `stat` structure.

Inodes

- the *inode* contains most of information found in the `stat` structure.
- every *inode* has a *link count* (`st_nlink`): it shows how many “things” point to this inode. Only if this *link count* is 0 (and no process has the file open) are the *data blocks* freed.

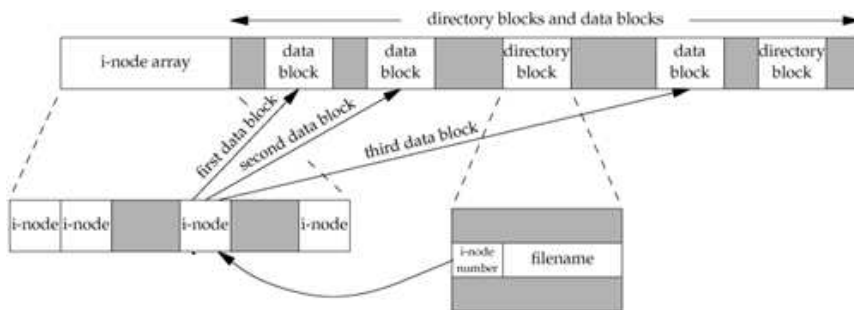
Inodes

- the *inode* contains most of information found in the `stat` structure.
- every *inode* has a *link count* (`st_nlink`): it shows how many “things” point to this inode. Only if this *link count* is 0 (and no process has the file open) are the *data blocks* freed.
- *inode* number in a directory entry must point to an *inode* on the same file system (no hardlinks across filesystems)



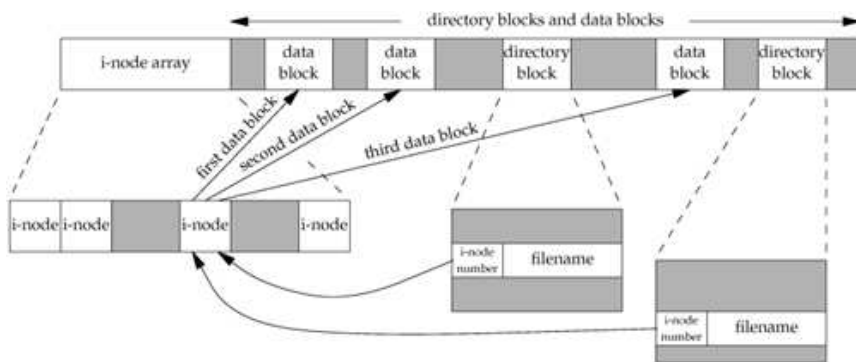
Inodes

- the *inode* contains most of information found in the `stat` structure.
- every *inode* has a *link count* (`st_nlink`): it shows how many “things” point to this inode. Only if this *link count* is 0 (and no process has the file open) are the *data blocks* freed.
- *inode* number in a directory entry must point to an *inode* on the same file system (no hardlinks across filesystems)
- to move a file within a single filesystem, we can just “move” the directory entry (actually done by creating a new entry, and deleting the old one).



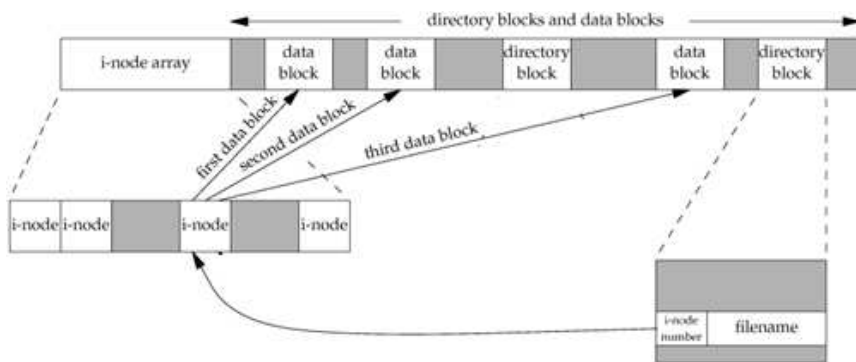
Inodes

- the *inode* contains most of information found in the `stat` structure.
- every *inode* has a *link count* (`st_nlink`): it shows how many “things” point to this inode. Only if this *link count* is 0 (and no process has the file open) are the *data blocks* freed.
- *inode* number in a directory entry must point to an *inode* on the same file system (no hardlinks across filesystems)
- to move a file within a single filesystem, we can just “move” the directory entry (actually done by creating a new entry, and deleting the old one).



Inodes

- the *inode* contains most of information found in the `stat` structure.
- every *inode* has a *link count* (`st_nlink`): it shows how many “things” point to this inode. Only if this *link count* is 0 (and no process has the file open) are the *data blocks* freed.
- *inode* number in a directory entry must point to an *inode* on the same file system (no hardlinks across filesystems)
- to move a file within a single filesystem, we can just “move” the directory entry (actually done by creating a new entry, and deleting the old one).



link(2)

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

```
int link(const char *name1, const char *name2);
```

Returns: 0 if OK, -1 on error

- Creates a link to an existing file (hard link).
- POSIX.1 allows links to cross filesystems, most implementations (SVR4, BSD) don't.
- only uid(0) can create links to directories (loops in filesystem are bad)

link(2) and unlink(2)

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

```
int link(const char *name1, const char *name2);
```

Returns: 0 if OK, -1 on error

- Creates a link to an existing file (hard link).
- POSIX.1 allows links to cross filesystems, most implementations (SVR4, BSD) don't.
- only uid(0) can create links to directories (loops in filesystem are bad)

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

```
int unlink(const char *path);
```

Returns: 0 if OK, -1 on error

- removes directory entry and decrements link count of file
- if file link count == 0, free data blocks associated with file (...unless processes have the file open)

link(2) and unlink(2)

```
$ cc -Wall wait-unlink.c  
$ ./a.out  
$ df .
```

rename(2)

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

```
int rename(const char *from, const char *to);
```

Returns: 0 if OK, -1 on error

If *oldname* refers to a file:

- if *newname* exists and it is not a directory, it's removed and *oldname* is renamed *newname*

rename(2)

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

```
int rename(const char *from, const char *to);
```

Returns: 0 if OK, -1 on error

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

rename(2)

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

```
int rename(const char *from, const char *to);
```

Returns: 0 if OK, -1 on error

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 *old/newname*

rename(2)

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

```
int rename(const char *from, const char *to);
```

Returns: 0 if OK, -1 on error

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 *old/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*

rename(2)

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

```
int rename(const char *from, const char *to);
```

Returns: 0 if OK, -1 on error

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 *old/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

rename(2)

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

```
int rename(const char *from, const char *to);
```

Returns: 0 if OK, -1 on error

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 *old/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

rename(2)

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

```
int rename(const char *from, const char *to);
```

Returns: 0 if OK, -1 on error

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 *old/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 *old/newname*

Symbolic Links

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

```
int symlink(const char *name1, const char *name2);
```

Returns: 0 if OK, -1 on error

- file whose "data" is a path to another file
- anyone can create symlinks to directories or files
- certain functions dereference the link, others operate on the link

How do we get the contents of a symlink? `open(2)` and `read(2)`?

Symbolic Links

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

```
int symlink(const char *name1, const char *name2);
```

Returns: 0 if OK, -1 on error

- file whose "data" is a path to another file
- anyone can create symlinks to directories or files
- certain functions dereference the link, others operate on the link

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

```
int readlink(const char *path, char *buf, size_t bufsz);
```

Returns: number of bytes placed into buffer if OK, -1 on error

This function combines the actions of `open`, `read`, and `close`.

Note: *buf* is not NUL terminated.

File Times

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

```
int utimes(const char *path, const struct timeval times[2]);  
int lutimes(const char *path, const struct timeval times[2]);  
int futimes(int fd, const struct timeval times[2]);
```

Returns: 0 if OK, -1 on error

If *times* is NULL, access time and modification time are set to the current time (must be owner of file or have write permission). If *times* is non-NULL, then times are set according to the `timeval` struct array. For this, you must be the owner of the file (write permission not enough).

Note that `st_ctime` is set to the current time in both cases.

For the effect of various functions on the access, modification and changes-status times see Stevens, p. 117.

Note: some systems implement `lutimes(3)` (library call) via `utimes(2)` syscalls.

mkdir(2) and rmdir(2)

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

int mkdir(const char *path, mode_t mode);
```

Returns: 0 if OK, -1 on error

Creates a new, empty (except for . and .. entries) directory. Access permissions specified by *mode* and restricted by the `umask(2)` of the calling process.

```
#include <unistd.h>

int rmdir(const char *path);
```

Returns: 0 if OK, -1 on error

If the link count is 0 (after this call), and no other process has the directory open, directory is removed. Directory must be empty (only . and .. remaining)

Reading Directories

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

DIR *opendir(const char *filename);
                                Returns: pointer if OK, NULL on error

struct dirent *readdir(DIR *dp);
                                Returns: pointer if OK, NULL at end of dir or on error

void rewinddir(DIR *dp);
int closedir(DIR *dp);
                                Returns: 0 if OK, -1 on error
```

- read by anyone with read permission on the directory
- format of directory is implementation dependent (always use readdir and friends)

opendir, readdir and closedir should be familiar from our small `ls` clone. `rewinddir` resets an open directory to the beginning so `readdir` will again return the first entry.

For directory traversal, consider `fts(3)` (not available on all UNIX versions).

Moving around directories

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

Returns: *buf* if OK, NULL on error

Get the kernel's idea of our process's current working directory.

```
#include <unistd.h>

int chdir(const char *path);
int fchdir(int fd);
```

Returns: 0 if OK, -1 on error

Allows a process to change its current working directory. Note that `chdir` and `fchdir` affect only the current process.

```
$ cc -Wall cd.c
```

```
$ ./a.out /tmp
```

Password File

Called a *user database* by POSIX and usually found in `/etc/passwd`, the password file contains the following fields:

Description	struct passwd member	POSIX.1
username	char *pw_name	X
encrypted passwd	char *pw_passwd	
numerical user id	uid_t pw_uid	X
numerical group id	gid_t pw_gid	X
comment field	char *pw_gecos	
initial working directory	char *pw_dir	X
initial shell	char *pw_shell	X

Encrypted password field is a one-way hash of the users password.
(Always maps to 13 characters from `[a-zA-Z0-9./]`.)

Some fields can be empty:

- password empty implies no password
- shell empty implies `/bin/sh`

Password File

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

struct passwd *getpwuid(uid_t uid);
struct passwd *getpwnam(const char *name);
```

Returns: pointer if OK, NULL on error

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

struct passwd *getpwent(void);

void setpwent(void);
void endpwent(void);
```

Returns: pointer if OK, NULL on error

- `getpwent` returns next password entry in file each time it's called, no order
- `setpwent` rewinds to "beginning" of entries
- `endpwent` closes the file(s)

See also: `getspnam(3)`/`getspent(3)` (where available)

Group File

Called a *group database* by POSIX and usually found in `/etc/group`, the group file contains the following fields:

Description	struct group member	POSIX.1
groupname	char *gr_name	X
encrypted passwd	char *gr_passwd	
numerical group id	uid_t gr_uid	X
array of pointers to user names	char **gr_mem	X

The `gr_mem` array is terminated by a NULL pointer.

Group File

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

struct group *getgrgid(gid_t gid);
struct group *getgrnam(const char *name);
```

Returns: pointer if OK, NULL on error

These allow us to look up an entry given a user's group name or numerical GID. What if we need to go through the group file entry by entry? Nothing in POSIX.1, but SVR4 and BSD give us:

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

struct group *getgrent(void);

void setgrent(void);
void endgrent(void);
```

Returns: pointer if OK, NULL on error

- `getgrent` returns next group entry in file each time it's called, no order
- `setgrent` rewinds to "beginning" of entries
- `endgrent` closes the file(s)

Supplementary Groups and other data files

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

int getgroups(int gidsetsize, gid_t *grouplist);
    Returns: returns number of suppl. groups if OK, -1 on error
```

Note: if `gidsetsize == 0`, `getgroups(2)` returns number of groups without modifying `grouplist`.

Other system databases

Similar routines as for password/group for accessing system data files:

Description	Data file	Header	Structure	Additional lookup functions
hosts	/etc/hosts	<netdb.h>	hostent	gethostbyname gethostbyaddr
networks	/etc/networks	<netdb.h>	netent	getnetbyname getnetbyaddr
protocols	/etc/protocols	<netdb.h>	protoent	getprotobyname getprotobynumber
services	/etc/services	<netdb.h>	servent	getservbyname getservbyport

System Identification

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

int uname(struct utsname *name);
    Returns:  nonnegative value if OK, -1 on error
```

- Pass a pointer to a `utsname` struct. This struct contains fields like `opsys` name, version, release, architecture, etc.
- This function used by the `uname(1)` command (try `uname -a`)
- Not that the size of the fields in the `utsname` struct may not be large enough to id a host on a network

To get just a hostname that will identify you on a TCP/IP network, use the Berkeley-derived:

```
#include <unistd.h>

int gethostname(char *name, int namelen);
    Returns:  0 if OK, -1 on error
```


Time and Date

```
#include <time.h>

time_t time(time_t *tloc);
        Returns:  value of time if OK, -1 on error
```

- Time is kept in UTC
- Time conversions (timezone, daylight savings time) handled "automatically"
- Time and date kept in a single quantity (`time_t`)

We can break this `time_t` value into its components with either of the following:

```
#include <time.h>

struct tm *gmtime(const time_t *calptr);
struct tm *localtime(const time_t *calptr);
        Returns:  pointer to broken down time
```

Time and Date

```
#include <time.h>

time_t mktime(struct tm *tm_ptr);
           Returns:  calendar time if OK, -1 on error
```

`localtime(3)` takes into account daylight savings time and the *TZ* environment variable. The `mktime(3)` function operates in the reverse direction. To output human readable results, use:

```
#include <time.h>

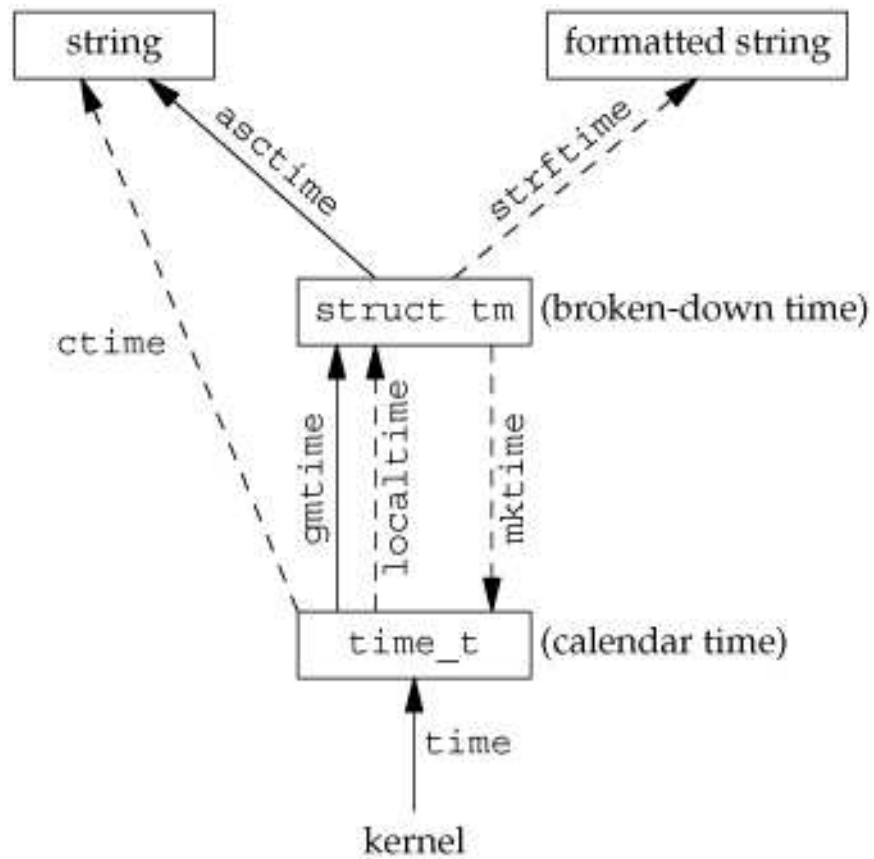
char *asctime(const struct tm *tm_ptr);
char *ctime(const struct tm *tm_ptr);
           Returns:  pointer to NULL terminated string
```

Lastly, there is a `printf(3)` like function for times:

```
#include <time.h>

size_t strftime(char *buf, size_t maxsize, const char *restricted_format, const struct tm *timeptr);
           Returns:  number of characters stored in array if room, else 0
```

Time and Date



Homework

Reading:

- Stevens, Chapter 4 and 6

Other:

- work on your midterm project!