

# UNIX System Programming

## Directories and File System

- ❖ Objectives
  - look at how to program with directories
  - briefly describe the UNIX file system

## Overview

1. Directory Implementation
2. UNIX File System
3. Links
4. Subdirectory Creation
5. “.” and “..”
6. `mkdir()`
7. `rmdir()`

8. Reading Directories
9. `chdir()`
10. `getcwd()`
11. Walking over Directories
12. `telldir()` and `seekdir()`
13. `scandir()`

## 1. Directory Implementation

- ❖ A UNIX directory is a *file*:
  - it has an owner, group owner, size, access permissions, etc.
  - many file operations can be used on directories
- ❖ Differences:
  - modern UNIXs have special directory operations
  - e.g. `opendir()`, `readdir()`

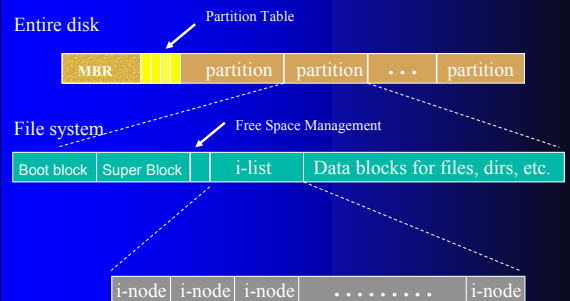
## Directory Structure

- ❖ A directory ‘file’ is a sequence of lines; each line holds an *i-node number* and a file name.
- ❖ The data is stored as binary, so we cannot simply use `cat` to view it
  - but some UNIXs (Solaris on `ajax`) allow:

```
$ od -c dir-name
```

```
      :
120  "maria.html"
207  "bob"
135  "FindIt.c"
      :
```

## 2. UNIX Memory Structure



## Entire Disk & Booting Computer

- ❖ Master Boot Record (sector 0)
  - used to boot computer
- ❖ Partition Table
  - starting and ending address of each partition
- ❖ “A program” reads in and executes the MBR
  - Search for the active partition
  - Reads in its first block (the **boot block**) and executes it.

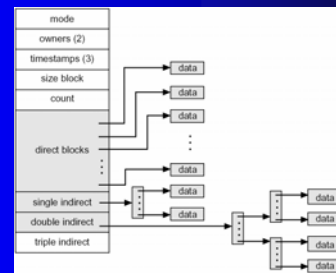
- ❖ *Boot block*: a hardware specific program that is called automatically to load UNIX at system startup time.

- ❖ *Super block*: it contains two lists:
  - a chain of free data block numbers
  - a chain of free inode number

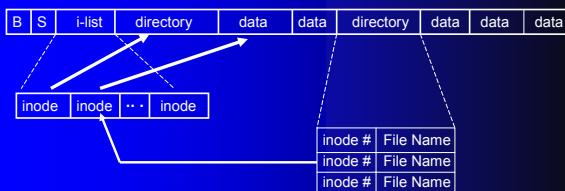
### ❖ *I-node*:

- The administrative information about a file is kept in a structure known as an *inode*.
  - ❖ Inodes in a file system, in general, are structured as an array known as an *inode table*.
- An inode number, which is an index to the inode table, *uniquely identifies* a file in a file system.

## i-node and Data Blocks



## File Systems Expanded



## 3. Links

- 3.1 What is a Link?
- 3.2 Creating a Link
- 3.3 Seeing Links
- 3.4 Removing a Link
- 3.5 Symbolic Links
- 3.6 Implementation

## 3.1. What is a Link?

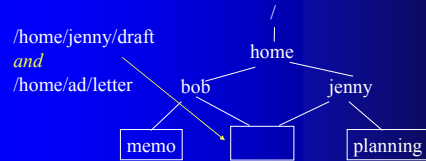
- ❖ A link is a pointer to a file.
- ❖ Useful for sharing files:
  - a file can be shared by giving each person their own link (pointer) to it.

## 3.2. Creating a Link

```
ln existing-file new-pointer
```

- ❖ Jenny types:

```
ln draft /home/bob/letter
```



- ❖ Changes to a file affects every link:

```
$ cat file_a
This is file A.
$ ln file_a file_b
$ cat file_b
This is file A.

$ vi file_b
:

$ cat file_b
This is file B after the change.
$ cat file_a
This is file B after the change.
```

## 3.3. Seeing Links

- ❖ Compare status information:

```
$ ls -l file_a file_b file_c file_d
```

```
-rw-r--r-- 2 maria 33 May 24 10:52 file_a
-rw-r--r-- 2 maria 33 May 24 10:52 file_b
-rw-r--r-- 1 maria 16 May 24 10:55 file_c
-rw-r--r-- 1 maria 33 May 24 10:57 file_d
```

- ❖ Look at inode number:

```
$ ls -li file_a file_b file_c file_d
```

```
3534 file_a 3534 file_b
5800 file_c 7328 file_d
```

- ❖ Directories may appear to more links:

```
{saffron:maria:105} ls -ld dir
drwxr-xr-x 2 maria users 4096 Apr 7 17:57 dir/
{saffron:ingrid:106} mkdir directory/hello
{saffron:ingrid:107} ls -ld directory
drwxr-xr-x 3 ingrid ingrid 4096 Apr 7 17:58 dir/
```

- ❖ This is because subdirectories (e.g. directories inside `dir/`) have a link back to their parent.

## 3.4. Removing a Link

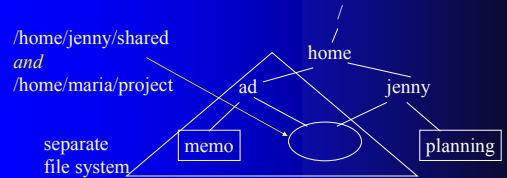
- ❖ Deleting a link does not remove the file.
- ❖ Only when the file *and* every link is gone will the file be removed.

## 3.5. Symbolic Links

- ❖ The links described so far are often called *hard links*
  - a hard link is a pointer to a file which must be on the *same* file system
- ❖ A *symbolic link* is an *indirect pointer* to a file
  - it stores the pathname of the pointed to file
  - it can link across file systems

### ❖ Jenny types:

```
ln -s shared /home/maria/project
```



### ❖ Symbolic links are listed differently:

```
$ ln -s pics /home/mh/img
$ ls -lF pics /home/mh/img
drw-r--r-- 1 maria staff 981 May 24 10:55 pics
lrwxrwxrwx 1 maria staff  4 May 24 10:57 /home/mh/img --> pics
```

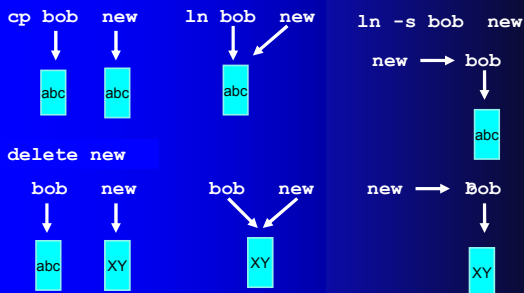
### ❖ Symbolic links can confuse:

```
$ ln -s /home/maria/grades /tmp/grades-old
$ cd /tmp/grades-old
$ pwd
/home/maria/grades

$ echo $cwd           (C Shell only)
/tmp/grades-old

$ cd ..
$ echo $cwd
/home/maria
```

## 3.6 Link Creation, Update & Removal



## 3.7 link() and unlink()

```
#include <unistd.h>
int link( const char *oldpath, const char *newpath );
```

### ❖ Meaning of:

```
link( "abc", "xyz" )
:
120 "fred.html"
207 "abc"
135 "bookmark.c"
207 "xyz"
:
```

- ❖ `unlink()` clears the **directory record**
  - usually means that the `i` node number is set to 0
  - the file may not be affected
- ❖ The `i`-node is only deleted when the **last** link to it is removed; the data block for the file is also deleted (reclaimed) & no process have the file opened

## Example: unlink

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

int main(void)
{
    if( open( "tempfile", O_RDWR ) < 0 )
    {
        perror( "open error" );
        exit( 1 );
    }
    if( unlink( "tempfile" ) < 0 )
    {
        perror( "unlink error" );
        exit( 1 );
    }
    printf( "file unlinked\n" );
    exit(0);
}
```

## remove()

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

- ❖ C Library function (*not* a system call)
- ❖ Delete a name and possibly the file it refers to.
  - It calls `unlink()` for files, and `rmdir()` for directories.

## rename()

```
#include <stdio.h>
int rename( const char *oldpath, const
char *newpath );
```

- ❖ Changes the name of `oldpath` to `newpath` (for both directories and files)
- ❖ If `oldpath` names open or nonexistent file, or if `newpath` names a file that already exists, then the action of `rename()` is *implementation-dependent*.

## symlink()

```
#include <unistd.h>
int symlink(const char *oldpath, const char
*newpath);
```

- ❖ Creates a symbolic link named `newpath` which contains the string `oldpath`.
- ❖ Symbolic links are interpreted at run-time.
- ❖ Dangling link – may point to a non-existing file.
- ❖ If `newpath` exists it will not be overwritten.

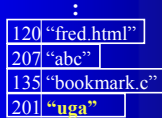
## readlink()

```
#include <unistd.h>
int readlink( const char *path, char *buf,
size_t bufsiz );
```

- ❖ Read value of a **symbolic link** (does *not* follow the link).
  - Places the contents of the symbolic link `path` in the buffer `buf`, which has size `bufsiz`.
  - Does not append a NULL character to `buf`.
- ❖ Return value
  - The count of characters placed in the buffer if it succeeds.
  - -1 if an error occurs.

## 4. Subdirectory Creation

- ❖ `mkdir uga` CAUSES:
  - the creation of a `uga` directory file and an `i` node for it
  - an `i` node number and name are added to the parent directory file



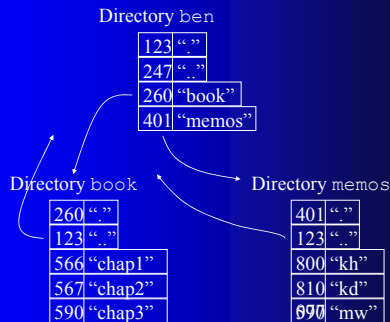
## 5. "." and ".."

- ❖ "." and ".." are stored as ordinary file names with `i`-node numbers pointing to the correct directory files.

- ❖ Example:



## In more detail:



## 6. mkdir()

- ❖ 

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

int mkdir(char *pathname, mode_t mode);
```
- ❖ Creates a new directory with the specified `mode`: return 0 if ok, -1 on error

- ❖ "." and ".." entries are added automatically
- ❖ `mode` must include execute permissions so the user(s) can use `cd`.  
e.g. 0755

## 7. rmdir()

- ❖ 

```
#include <unistd.h>
int rmdir(char *pathname);
```
- ❖ Delete an empty directory; return 0 if ok, -1 on error.
- ❖ Will delay until other processes have stopped using the directory.

## 8. Reading Directories

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

DIR *opendir(char *pathname);

struct dirent *readdir(DIR *dp);

void rewinddir(DIR *dp);

int closedir(DIR *dp);
```

returns a pointer if ok, NULL on error

returns a pointer if ok, NULL at end or on error

return 0 if ok, -1 on error

## dirent and DIR

```
❖ struct dirent
{
    long d_ino; /* i-node number */
    char d_name[NAME_MAX+1]; /* fname */
    off_t d_off; /* offset to next rec */
    unsigned short d_reclen; /* record length */
}
```

❖ DIR is a directory stream (similar to FILE)

- when a directory is first opened, the stream points to the first entry in the directory

## Example: listdir.c

List the contents of the current directory.

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

int main()
{
    DIR *dp;
    struct dirent *dir;

    if( (dp = opendir(".")) == NULL )
    {
        fprintf( stderr, "Cannot open dir\n" );
        exit(1);
    }
}
```

```
/* read entries */
while( (dir = readdir(dp)) != NULL )
{
    /* ignore empty records */
    if( dir->d_ino != 0 )
        printf( "%s\n", dir->d_name );
}

closedir( dp );
return 0;
} /* end main */
```

## 9. chdir()

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

❖ Change the current working directory (*cwd*) of the calling process; return 0 if ok, -1 on error.

## Example: cd to /tmp

❖ Part of `to_tmp.c`:

```

;
if( chdir("/tmp") < 0
    printf( "chdir error\n" );
else
    printf( "In /tmp\n" );
```

## Directory Change is Local

- ❖ The directory change is limited to within the program.

❖ e.g.

```
$ pwd
/usr/lib
$ to_tmp      /* from last slide */
In /tmp
$ pwd
/usr/lib
```

## 10. `getcwd()`

- ❖ `#include <unistd.h>`  
`char *getcwd(char *buf, int size);`
- ❖ Store the *cwd* of the calling process in `buf`; return `buf` if ok, `NULL` on error.
- ❖ `buf` must be big enough for the pathname string (`size` specifies the length of `buf`).

## Example

```
#include <stdio.h>
#include <unistd.h>
#include <dirent.h> /* for NAME_MAX */

int main()
{
    char name[NAME_MAX+1];

    if( getcwd( name, NAME_MAX+1 ) == NULL )
        printf( "getcwd error\n" );
    else
        printf( "cwd = %s\n", name );
    :
}
```

## 11. Walking over Directories

- ❖ 'Visit' every file in a specified directory *and* all of its subdirectories
  - visit means get the name of the file
- ❖ Apply a user-defined function to every visited file.

## Function Prototypes

```
❖ #include <ftw.h>

/* ftw means file tree walk,
   starting at directory */
int ftw( char *directory, MyFunc *fp,
         int depth );

/* apply MyFunc() to each visited file */
typedef int MyFunc( const char *file,
                   struct stat *sbuf, int flag );
```

- ❖ `depth` is the maximum number of directories that can be open at once. Safest value is 1, although it slows down `ftw()`.
- ❖ Result of `ftw()`: 0 for a successful visit of *every* file, -1 on error.

## MyFunc Details

- ❖ The `file` argument is the pathname relative to the start directory
  - it will be passed to `MyFunc()` automatically by `ftw()` as it visits each file
- ❖ `sbuf` argument is a pointer to the `stat` information for the file being examined.

- ❖ The `flag` argument will be set to one of the following for the item being examined:
  - `FTW_F` Item is a regular file.
  - `FTW_D` Item is a directory.
  - `FTW_NS` Could not get stat info for item.
  - `FTW_DNR` Directory cannot be read.
- ❖ If the `MyFunc` function returns a non-zero value then the `ftw()` walk will terminate.

## Example: shower.c

Print the names of all the files found below the current directory.

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

int shower(const char *file,
           const struct stat *sbuf,
           int flag);

void main()
{
    ftw(".", shower, 1);
}
```

```
int shower(const char *file,
           const struct stat *sbuf,
           int flag)
{
    if (flag == FTW_F) /* is a file */
        printf("Found: %s\n", file);
    return 0;
}
```

## 12. telldir() and seekdir()

- ❖ `#include <dirent.h>`  
`off_t telldir(DIR *dp);`
- ❖ Returns the location of the current record in the directory file, -1 on error.
- ❖ `#include <dirent.h>`  
`void seekdir(DIR *dp, off_t loc);`
- ❖ Set the location in the directory file.

## 13. scandir()

- ❖ Scan the directory file calling the selection function (`sfp`) on each directory item.
- ❖ Items for which `sfp` returns non-zero are stored in an array pointed to by `pitems`.
- ❖ The items in the array are sorted using `qsort()` using the comparison function (`cfp`).

## Function Prototypes

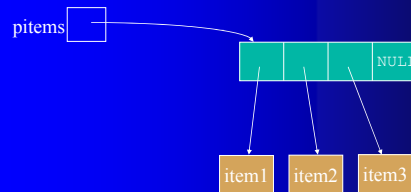
```
❖ #include <dirent.h>

int scandir(char *directory,
            struct dirent ***pitems,
            SelectFnc *sfp,
            CompareFnc *cfp);

/* selection function for searching */
typedef int SelectFnc(const struct dirent *d);

/* comparison function for sorting */
typedef int CompareFnc(const struct dirent *d1,
                      const struct dirent *d2);
```

## pitems Pictured



- ❖ `dirent.h` includes one pre-defined comparison function, which compares directory item *names*:

```
int alphasort(const struct dirent *d1,
             const struct dirent *d2);
```

- ❖ This causes the `pitems` array to be sorted into increasing alphabetical order by item name.

## Example: `geth.c`

List the files/subdirectories in the current directory which begin with 'h', sorted in alpha order.

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

/* our selection function */
int selectH(const struct dirent *d);

void main()
{ struct dirent **pitems;

  scandir(".", &pitems, selectH, alphasort);
  while (*pitems != NULL) {
    printf("%s\n", (*pitems)->d_name);
    pitems++;
  }
}
```

```
int selectH(const struct dirent *d)
/* Does item name start with 'h'? */
{
  if (d->d_name[0] == 'h')
    return 1;
  return 0;
}
```