System Programming
(ELEC462)

Focus on File Systems

Dukyun Nam

HPC Lab@KNU

Contents

- Introduction
- A User’s View of the File System
- Internal Structure of the Unix File System
- Understanding Directories
- Writing `pwd`
- Multiple File Systems: A Tree of Trees
- Summary

Introduction

- Files and directories
  - Files have content and properties
  - Directories are list of files
    - Organized into a tree-like structure
    - Can contain other directories
- What do the following mean?
  - “A file is in a directory.”
  - “You are in your home directory when logging into a Linux machine.”
  - What does it mean for a person or a file to be “in a directory?”

Introduction (cont.)

- A hard disk is a stack of metal platters
    - Each platter has a magnetically responsive coating
    - A disk stores files, file info, and dirs in a tree-structure
- How does this stack of spinning metal appear to be a tree of files, properties, and directories?
    - This chapter will let you learn how files are physically organized in a disk with some hand-on experiences
- Question) What is the internal structure of the file system?
    - Write `pwd`

Introduction (cont.)

- `pwd` reports your current location with in directory tree.
- System calls to be studied in this chapter:
  - `mkdir, rmdir, chdir, link, unlink, rename, symlink`
- Besides, we will learn:
  - How directories are connected,
  - How `cat/pwd` works, and
  - Mounting file systems

A User’s View of the File System

- Build the tree with this sequence of commands
  - Can you draw a tree of directories?

```
$ mkdir demodir
$ cd demodir
$ pwd
/home/yourname/demodir
$ mkdir b oops
$ mv b c
$ rmdir oops
$ cd c
$ mkdir d1 d2
$ cd ../..
$ mkdir demodir/a
```

A User’s View of the File System (cont.)

- Directory Commands:
  - `mkdir`: creates a directory with a specified name
  - `rmdir`: removes a directory
  - `mv`: renames or moves directories (or files)
  - `cd`: moves from one to another directory
- File Handling Commands:
  - ‘`..`’ : parent directory.
  - Example:

```
$ cd ../..
```

A User’s View of the File System (cont.)

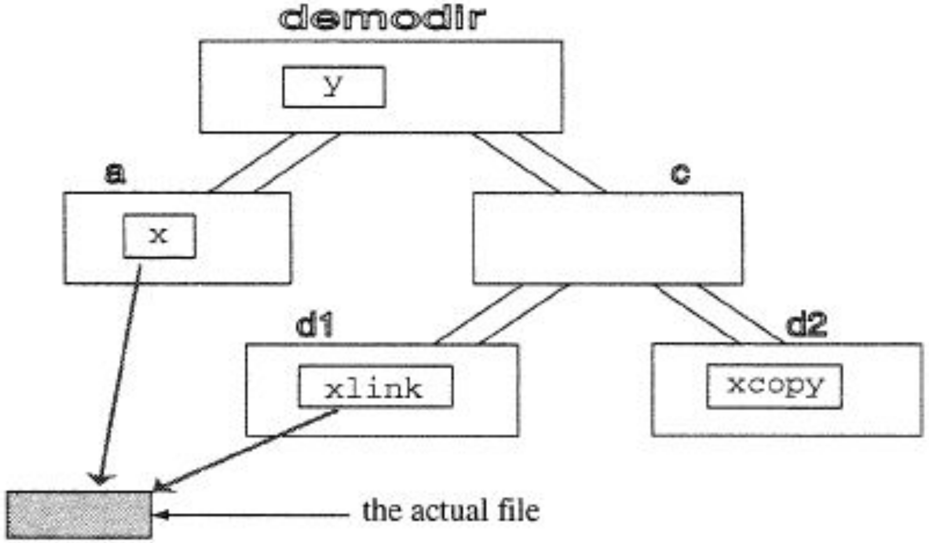
- Create some files in this directory tree
    - Where are files?

```
$ cd demodir
$ cp /etc/group x
$ cat x
$ cp x copy.of.x
$ mv copy.of.x y
$ mv x a
$ cd c
$ cp ../a/x d2/xcopy
```

```
$ ln ../a/x d1/xlink #link
$ ls > d1/xlink # overwrite
$ cp d1/xlink z
$ rm ../../demodir/c/d2/../z
$ cd ../..
$ cat demodir/a/x
(what appears here?)
```

A User's View of the File System (cont.)

- A snapshot of a filesystem
    - x and xlink are called "links"



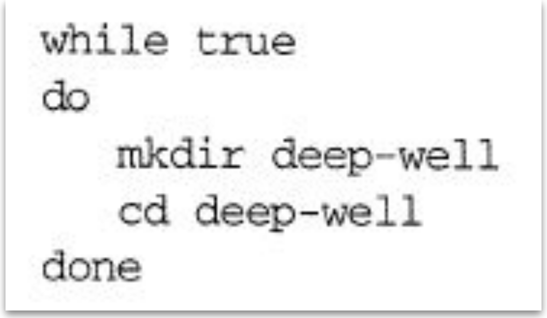
< Two links to the same file >

A User’s View of the File System (cont.)

- Tree commands
  - `ls –R`: can list the items of an entire tree
    - Directories + Subdirectories
  - `chmod –R`: changes permission bits of files
    - By utilizing features of “–R,” we “recursively” apply changes to all files in subdirectories
  - `du`: disk usage command
    - It reports the number of the disk blocks used by a specified directory
  - `find`: searches a directory and all its subdirectories for files and descriptions specified on command line
    - e.g., `find . -name textbook`

A User’s View of the File System (cont.)

- The internal structure of the system imposes no limit on the depth of a directory tree
  - An infinite number of directories can be virtually created
- What’s going to happen if you run the following script on Linux?



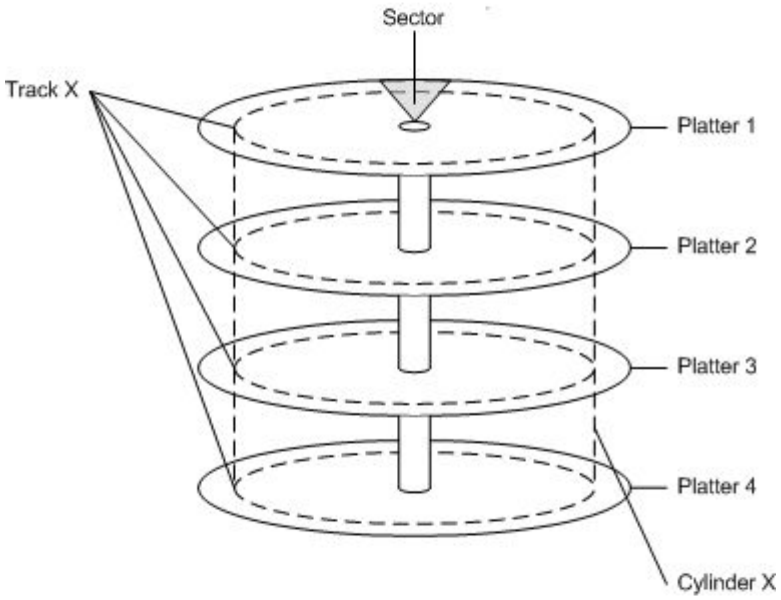
```
while true
do
   mkdir deep-well
   cd deep-well
done
```

A User’s View of the File System (cont.)

- Additional commands
  - `tree`: a recursive directory listing program
    - List contents of directories in a tree-like format
  - `find` with “`2>/dev/null`”
    - `> file`: redirect stdout to file
    - `1> file`: redirect stdout to file
    - `2> file`: redirect stderr to file
    - `/dev/null`: the null device it takes any input you want and throws it away
    - examples
      - `find . -name` *`keyword`* `-print`
      - `find . -name` *`keyword`* `-print 2> /dev/null`

Internal Structure of the Unix File System

- Abstraction zero: from platters to partitions
  - A hard disk can contain one or more logical regions called *partitions*
  - Disk can store a huge amount of data; can be divided into partitions
    - To create separate regions within a larger entity for different purposes.
      - e.g. On Windows: “C:\”, “D:\”, …
      - c.f. Country vs. provinces (or states) / cities / villages
    - We can treat each partition as a separate (but virtual) disk

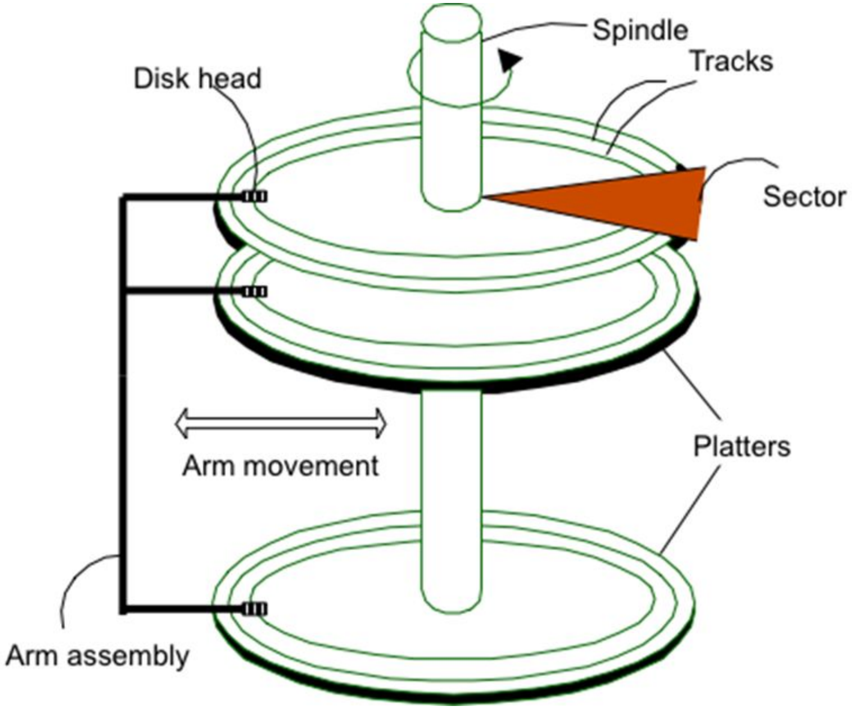


*** Image from https://learn.microsoft.com/en-us/windows/win32/fileio/disk-devices-and-partitions**

Internal Structure of the Unix File System (cont.)

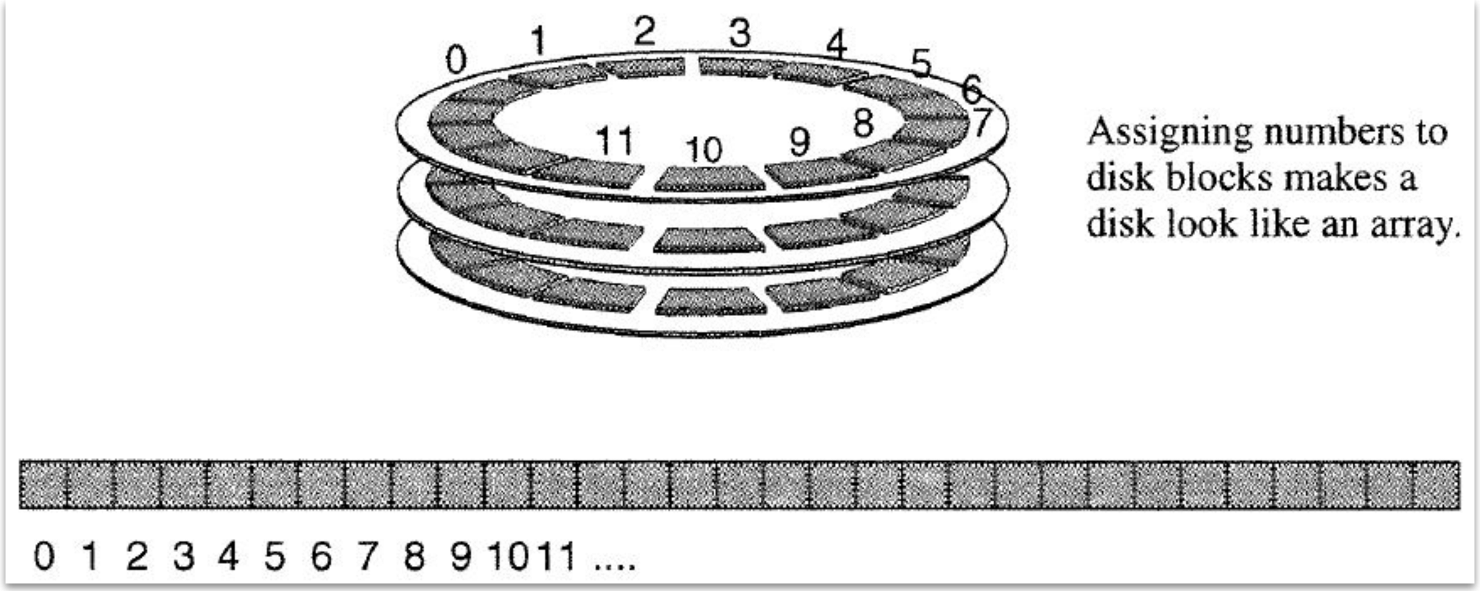
- Abstraction one: from platters to an array of blocks
  - *Disk* is a stack of magnetic *platters*
  - The surface of each platter is organized into *tracks*
  - Tracks divided into *disk sectors*
  - Sector: the basic unit of storage on the disk
    - Typically, 512 Bytes
  - (Disk) *Block* (or page): a set of sectors

    (in unit of read/write)
    - Typically, 4KBytes
  - * To read or write data into disk, (mechanical) arm moves and the spindle spins until the requested sector comes under the disk head



Internal Structure of the Unix File System (cont.)

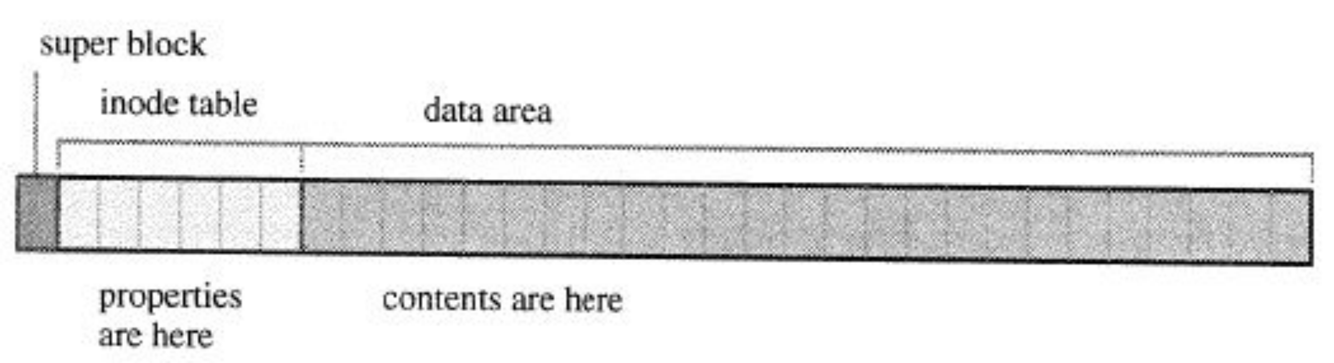
- Abstraction one: from platters to an array of blocks
  - Assigning numbers to disk blocks



Assigning numbers to disk blocks makes a disk look like an array.

Internal Structure of the Unix File System (cont.)

- Abstraction two: from an array of block to three regions
  - A file system stores file contents, file properties (owner, date, etc.), and directories that hold those files
  - Divide the array of blocks into three sections



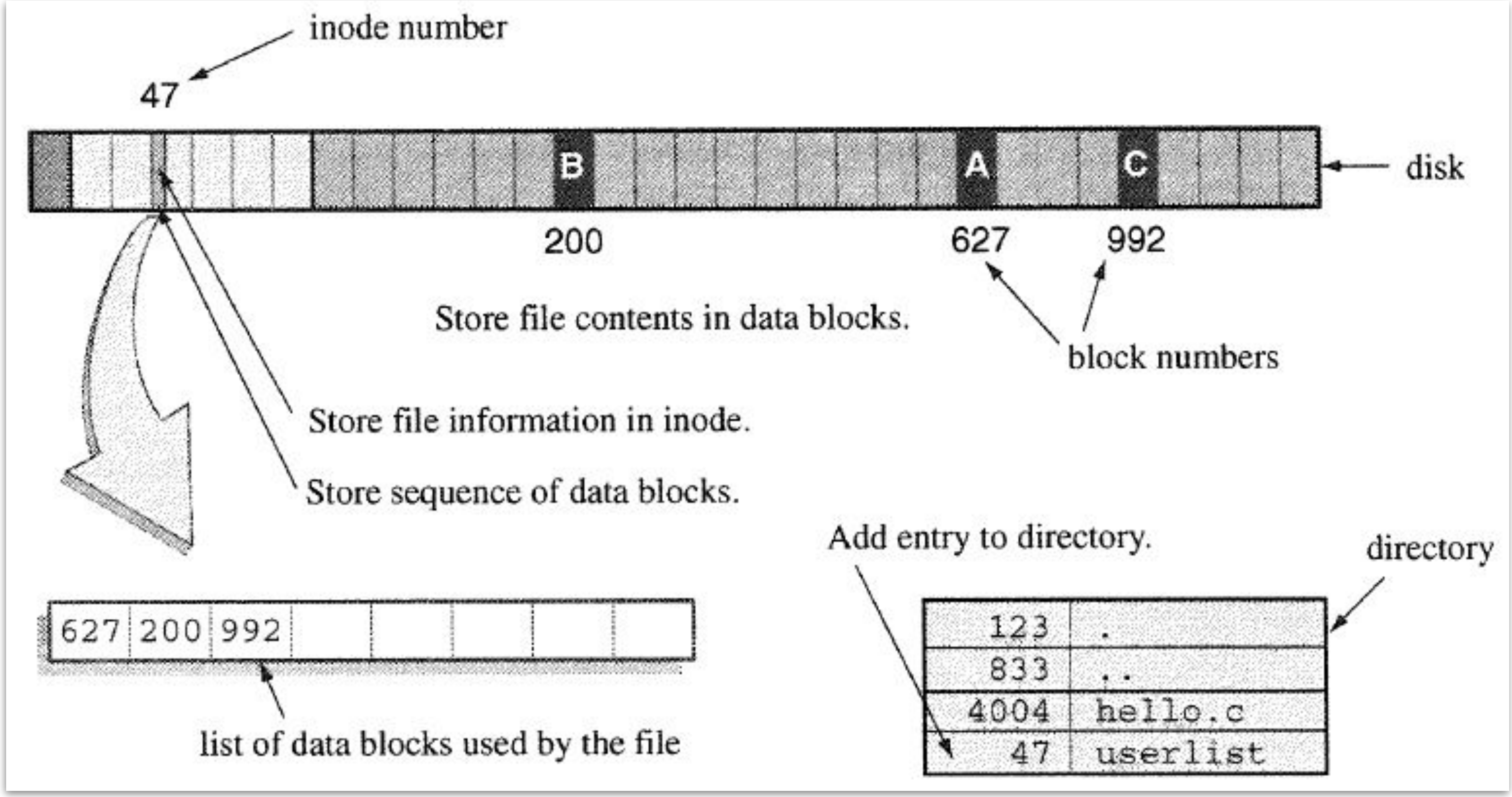
< The three regions of a file system >

Internal Structure of the Unix File System (cont.)

- Three regions
  - 1) The Superblock
    - Contains the information about the organization of the file system itself.
      - e.g. the size of each area, the location of unused data blocks, …
  - 2) The Inode Table
    - Each file has a set of properties (or, metadata): size, user ID of the owner, and last modification time
    - Those properties are recorded in a struct called an *inode*
    - All inodes are the same size, and the inode table is simply an array of those structs
  - 3) The Data Area
    - The actual contents of files are kept in this section
    - All blocks on the disk are the same size

The File System in Practice: Creating a File

- Internal structure of a file
  - Example) `$ who > userlist`



inode number
47
B
A
C
disk
200
627
992
Store file contents in data blocks.
block numbers
Store file information in inode.
Store sequence of data blocks.
Add entry to directory.
directory
list of data blocks used by the file

627	200	992					

123	.
833	..
4004	hello.c
47	userlist

Four Main Operations in Creating a New File

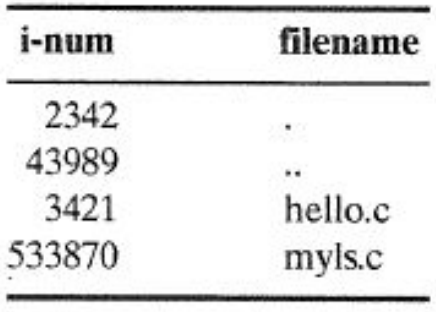
- 1) Store Properties
    - The file has properties.
    - The kernel locates a free inode.
    - The kernel gets inode number 47.
    - The kernel records information about the file in this inode.
- 2) Store Data
    - The file has contents; this file requires 3 blocks of storage.
    - The kernel locates 3 free blocks: 627, 200, 992.
    - Each chunk of bytes is copied from the kernel buffers to these 3 blocks.

Four Main Operations in Creating a New File (cont.)

- 3) Record Allocation
  - The contents of this file are in blocks 627, 200, and 992, in that order
  - The kernel records that sequence of block numbers in the disk allocation section of the inode
  - The disk allocation section is an array of block numbers
- 4) Add Filename to Directory
  - The kernel adds the entry (47, `userlist`) to the directory

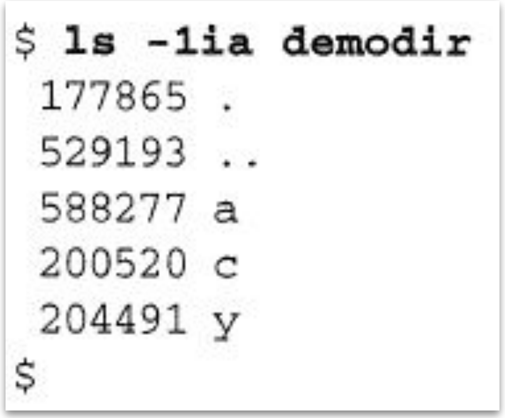
The File System in Practice: How Directories Work

- A directory is a “special kind of file,” containing a list of names of files



i-num	filename
2342	.
43989	..
3421	hello.c
533870	myls.c

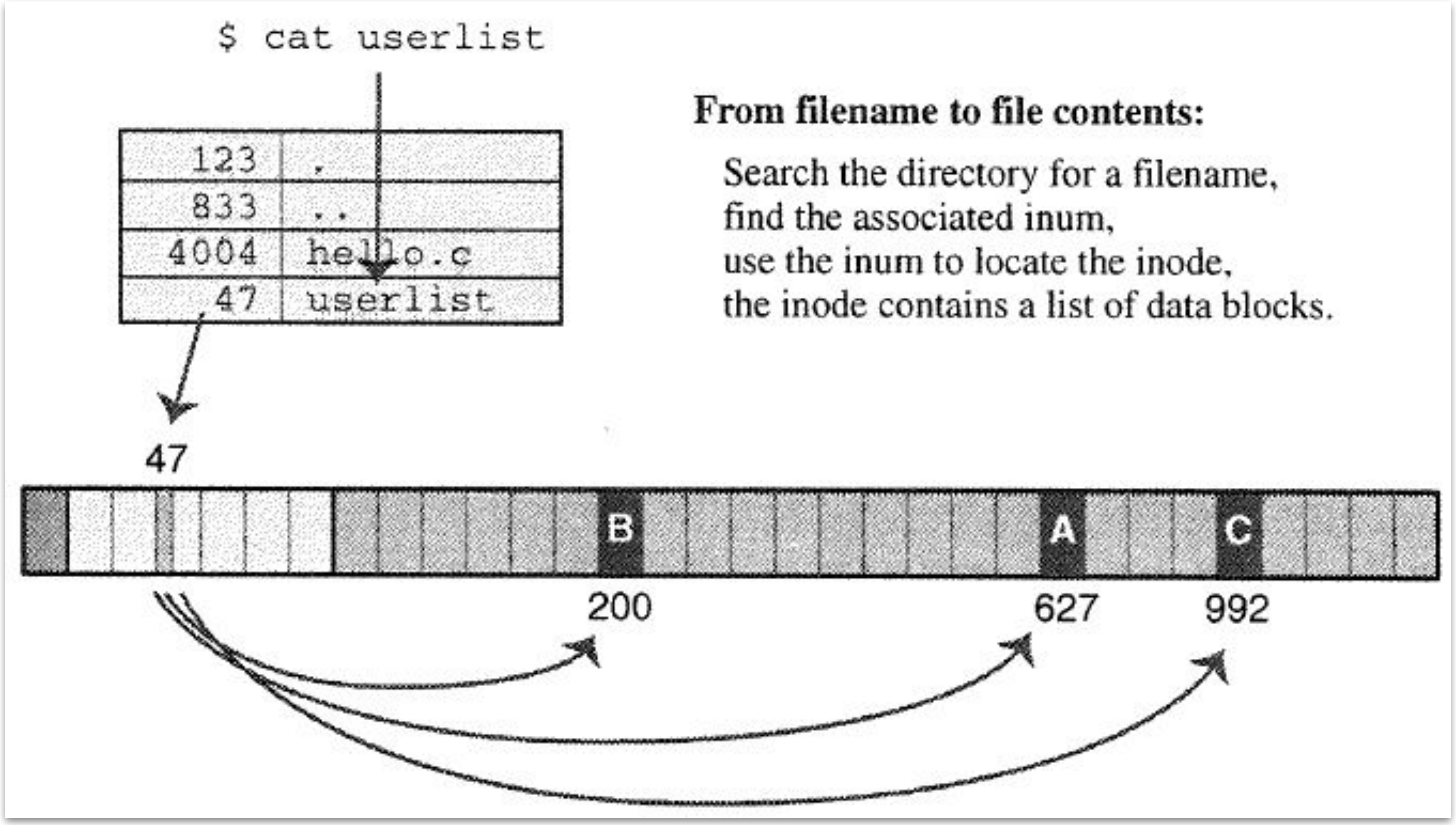
- Looking inside a directory
  - `-1`: list one file per line (one-column output)
  - `-i`: print the index number of each file
  - `-a`: do not ignore entries
  - e.g., Let’s see i-nums of . and .. in root directory
    - `$ ls -1ia /`



```
$ ls -1ia demodir
 177865 .
 529193 ..
 588277 a
 200520 c
 204491 y
$
```

The File System in Practice: How `cat` Work

- From filename to disk blocks



```
$ cat userlist
```

123	.
833	..
4004	hello.c
47	userlist

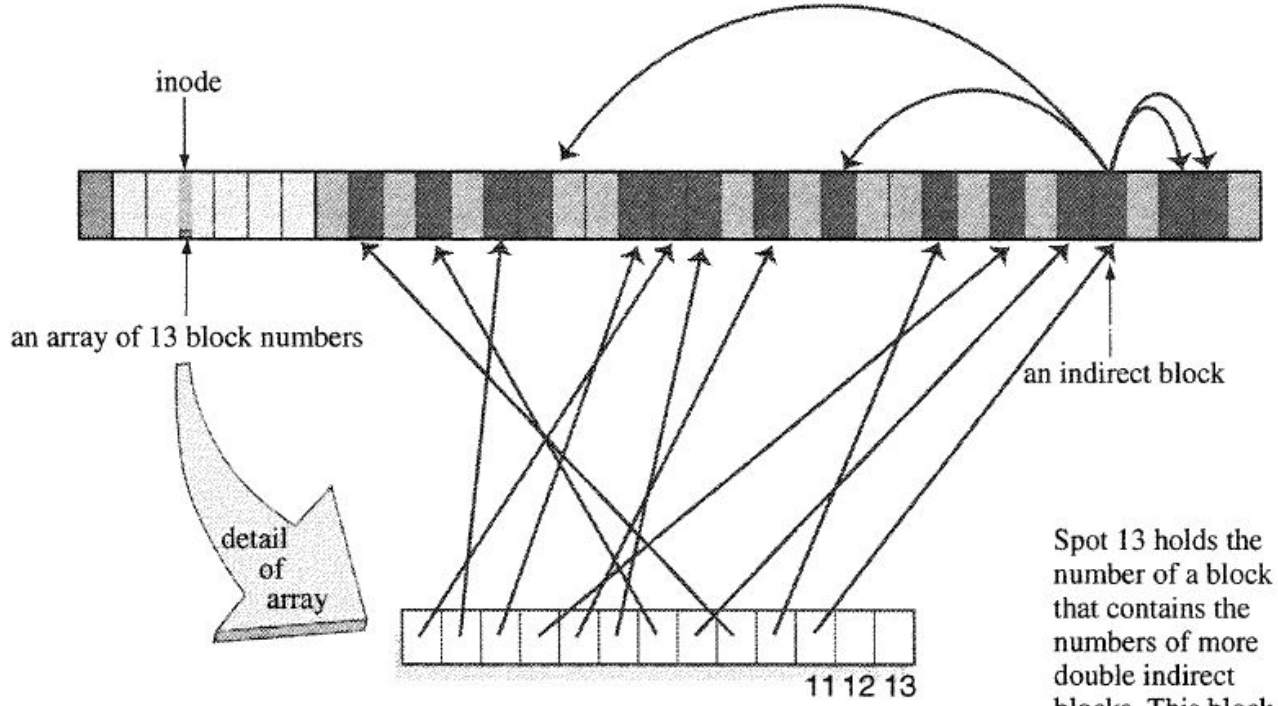
From filename to file contents:

Search the directory for a filename,
find the associated inum,
use the inum to locate the inode,
the inode contains a list of data blocks.

Inodes and Big Files

- Recording data block allocation for a file with 14 blocks

Recording data block allocation for a file with fourteen blocks of data:



The allocation list starts in the first 10 spots in the array in the inode.

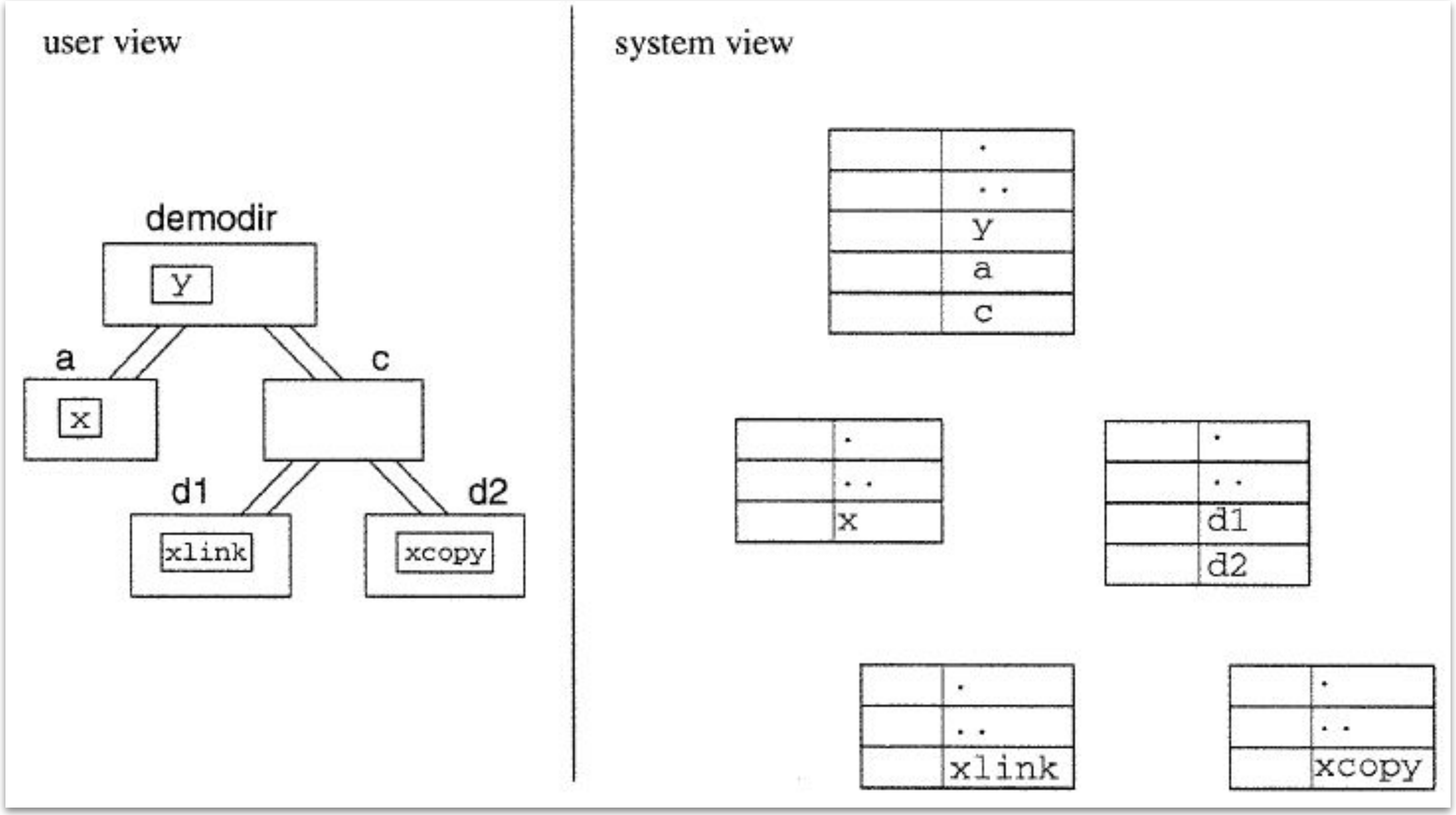
The allocation list continues in an indirect block. Spot 11 holds the number of that block.

Spot 12 holds the number of a block that contains the numbers of more indirect blocks. This block is called a double indirect block.

Spot 13 holds the number of a block that contains the numbers of more double indirect blocks. This block is called a triple indirect block.

Understanding Directories

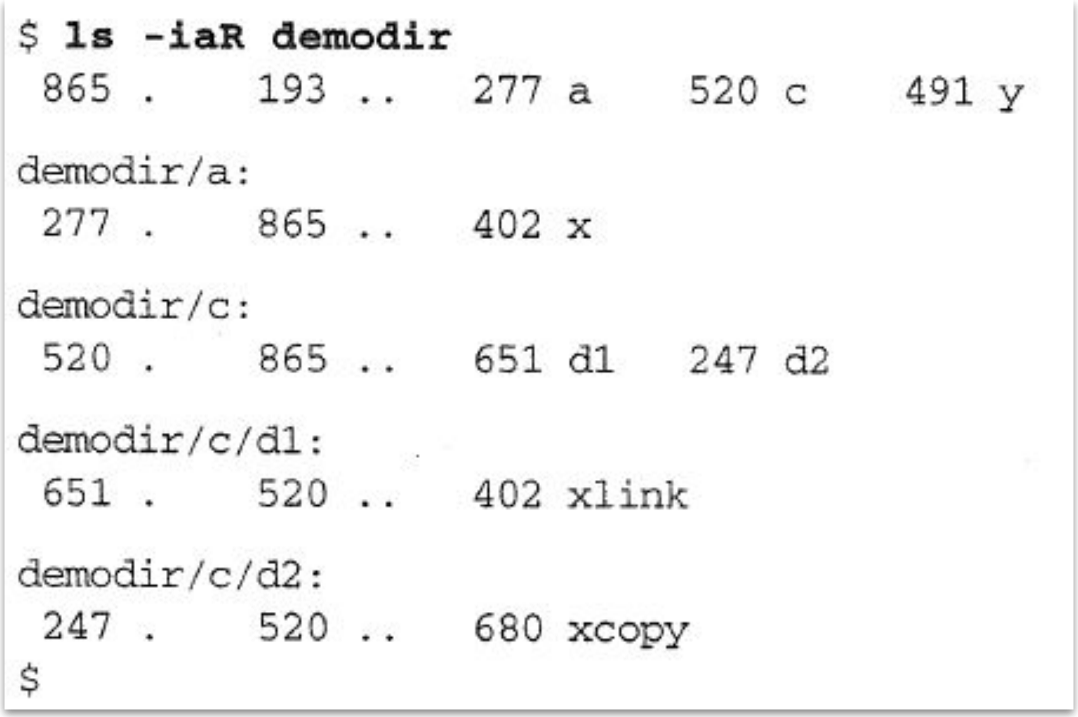
- Internally, a directory is a file that contains a list of pairs:
  - Filename and inode number



< Two views of a directory tree >

Understanding Directories (cont.)

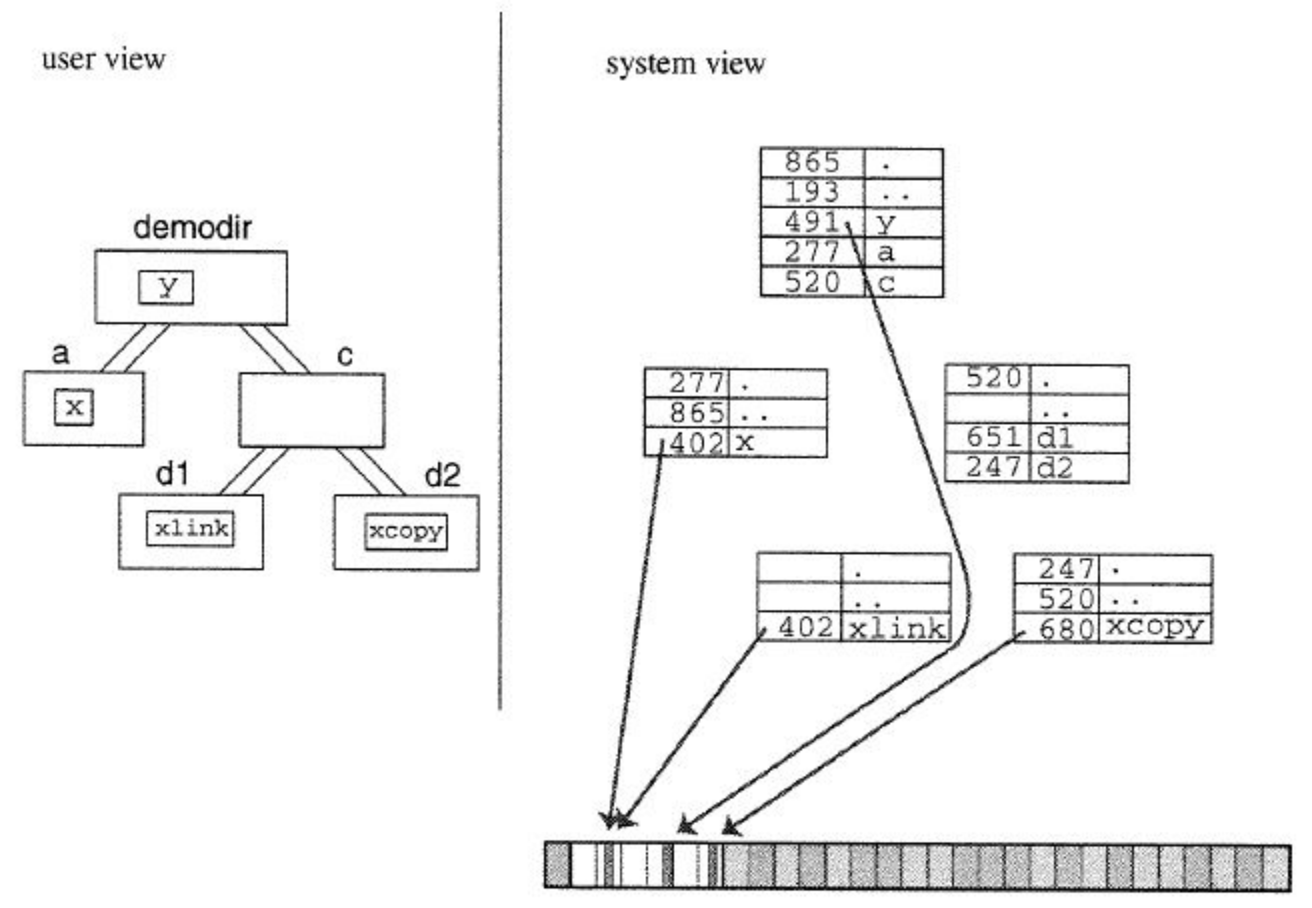
- List inode numbers for all files recursively down a tree
  - `$ ls -iaR demodir`



```
$ ls -iaR demodir
 865 .     193 ..    277 a     520 c     491 y

demodir/a:
 277 .     865 ..    402 x

demodir/c:
 520 .     865 ..    651 d1    247 d2

demodir/c/d1:
 651 .     520 ..    402 xlink

demodir/c/d2:
 247 .     520 ..    680 xcopy
$
```

Understanding Directories (cont.)

- Diagram with most of the inode numbers filled in



865	.
193	..
491	y
277	a
520	c

277	.
865	..
402	x

520	.
	..
651	d1
247	d2

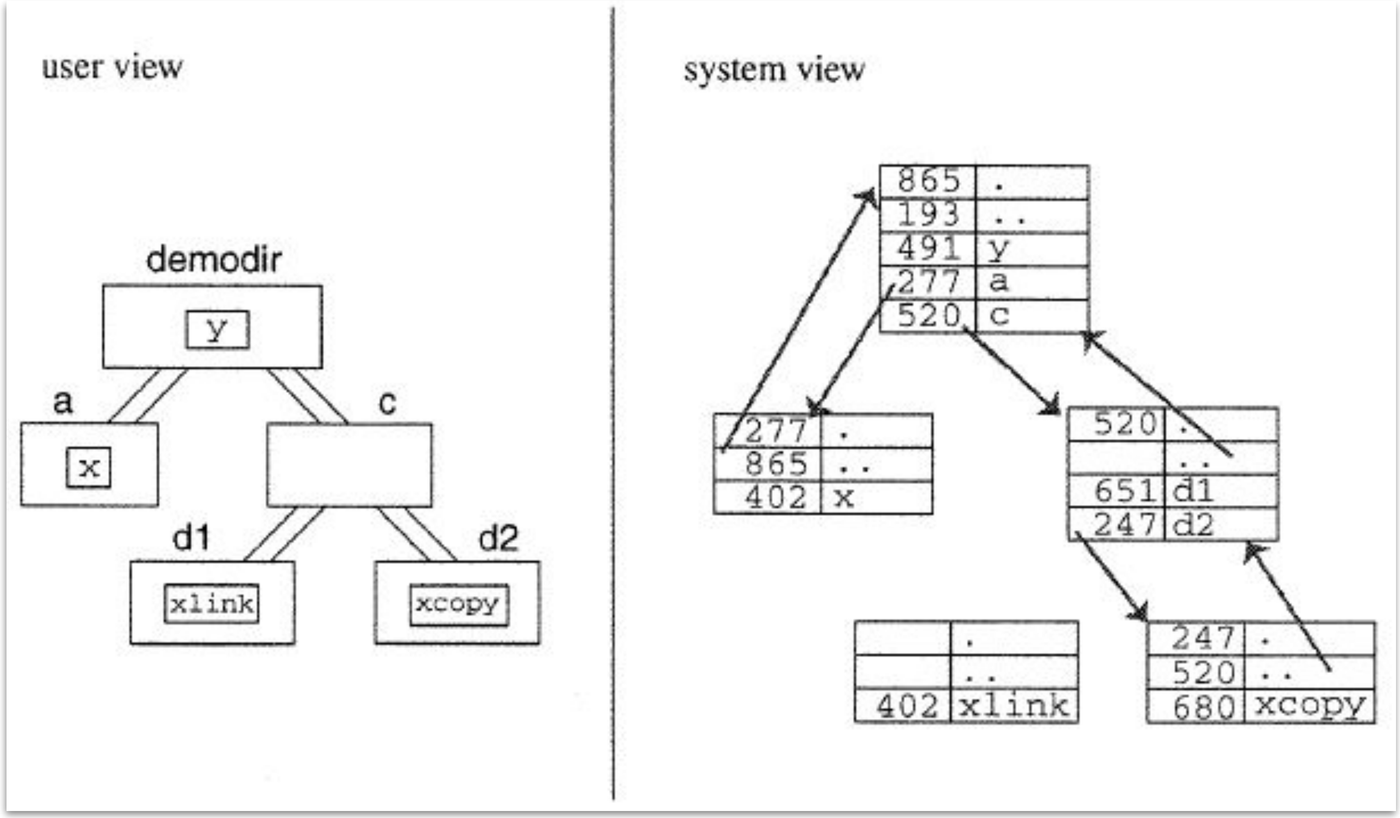
	.
	..
402	xlink

247	.
520	..
680	xcopy

< Filenames and pointers to files >

Understanding Directories (cont.)

- The real meaning of “A file is in a directory.”



865	.
193	..
491	y
277	a
520	c

277	.
865	..
402	x

520	.
	..
651	d1
247	d2

	.
	..
402	xlink

247	.
520	..
680	xcopy

< Directory names and pointers to directories >

Understanding Directories (cont.)

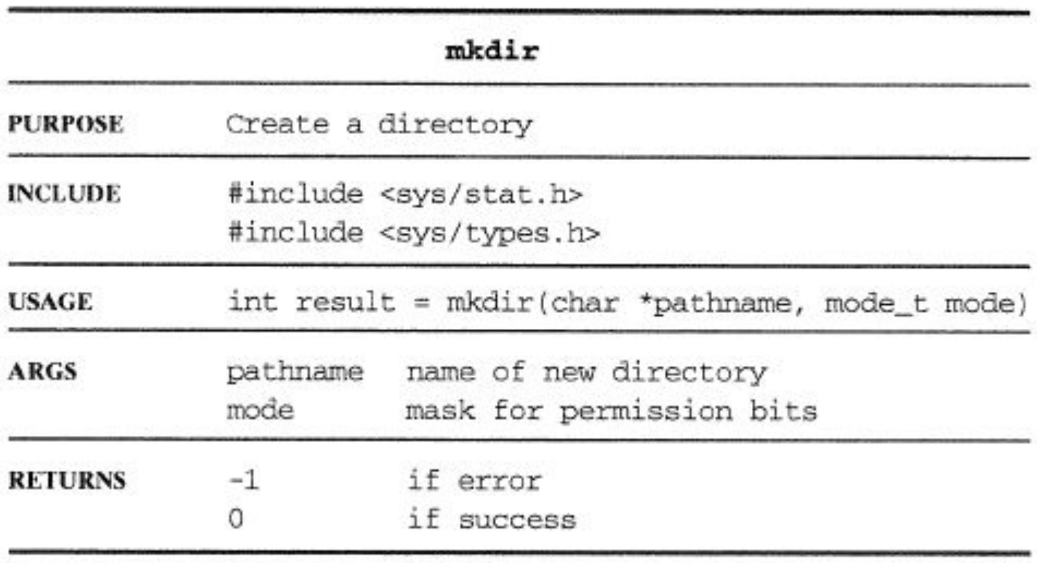
- The real meaning of “A file is in a directory.”
  - “File `x` is in directory `a`” means there is a ‘link’ to inode 402 in the directory called `a`.
  - The filename attached to that link is `x`.
  - It’s important to remember that the directory marked ‘`d1`’ contains a link to inode 402, which is called `xlink`.
  - These two links to inode 402 (`demodir/a/x` and `demodir/c/d1/xlink`) refer to the same file.
- In short, directories contain “references” to files.
  - Each of these **references** is called “**link**.”
  - The **contents** of the file are in “**data blocks**.”
  - The **properties** of the file are recorded in a “**struct in the inode table**.”
  - The **inode number** and a **(link) name** are stored **in a directory**.

Understanding Directories (cont.)

- The real meaning of “A directory contains a subdirectory.”
  - e.g., `a` (inode 277) contained in `demodir`
    - The kernel installs in every directory an entry for its own inode, called “`.`”.
- The real meaning of “A directory has a parent directory.”
  - e.g., `c` (inode 520) is the parent directory of `d2` (inode 247), marked “`..`”.
- In sum, in the Unix/Linux file system,
  - Files do **NOT** have names.
  - Links **DO** have names.
  - Files have **inode numbers**.

Commands and System Calls for Directory Trees

- `mkdir`: the command to create new directories
    - Uses `mkdir()`
    - `mkdir()`: creates a new directory and links its inode to the file system tree; more specifically,
        - Creates the inode for the directory
        - Allocates a disk block for its contents
        - Installs in the directory "`.`" and "`..`" entries with certain inode numbers, and
        - Finally, adds a link to that node to its parent directory.



	mkdir	
PURPOSE	Create a directory	

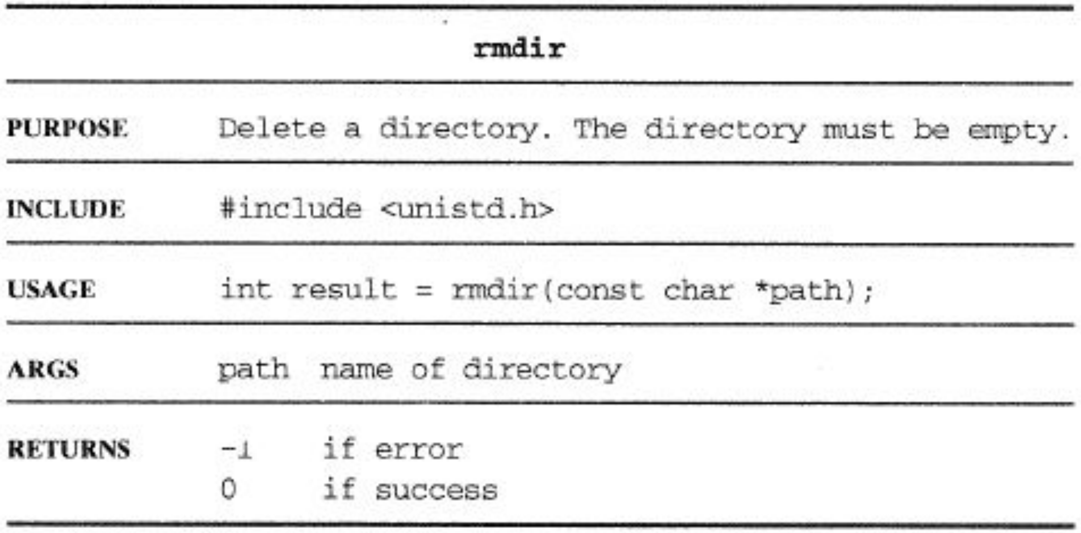
INCLUDE	#include <sys/stat.h>
	#include <sys/types.h>	

USAGE	int result = mkdir(char *pathname, mode_t mode)	
ARGS	pathname mode	name of new directory mask for permission bits

RETURNS	-1
	0
		if error
		if success

Commands and System Calls for Directory Trees (cont.)

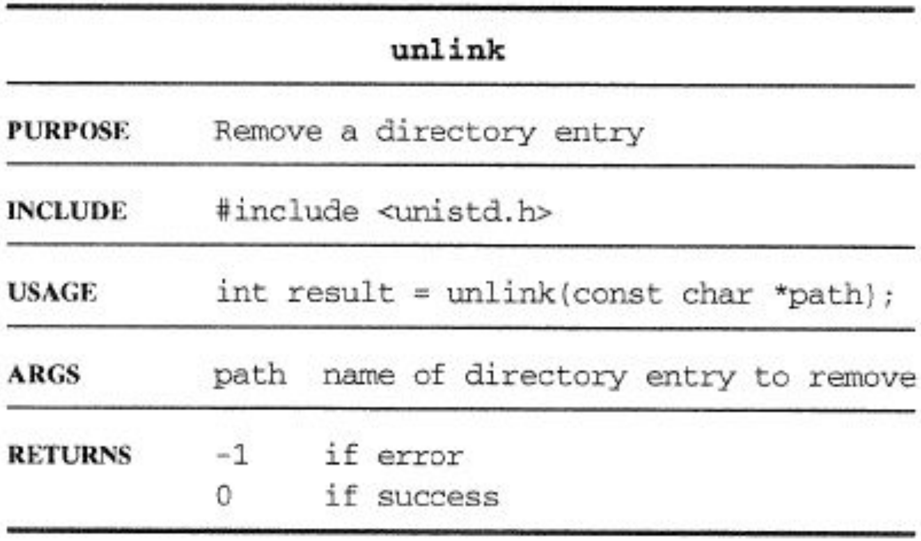
- `rmdir`: the command to delete a directory
  - Uses `rmdir()`
  - `rmdir()`: removes a directory node from a directory tree
    - The directory must be EMPTY.
    - If the directory itself is not used by any other process, then the inode and data are freed.



	rmdir
PURPOSE	Delete a directory. The directory must be empty.
INCLUDE	`#include <unistd.h>`
USAGE	`int result = rmdir(const char *path);`
ARGS	path name of directory
RETURNS	-1 if error 0 if success

Commands and System Calls for Directory Trees (cont.)

- `rm`: the command to remove entries from a directory
  - Uses `unlink()`
  - `unlink()`: deletes a directory entry
    - Decrements the link count for the corresponding inode
    - If the link count for the inode becomes zero, the data blocks and inode are freed
    - `unlink` may not be used to unlink directories

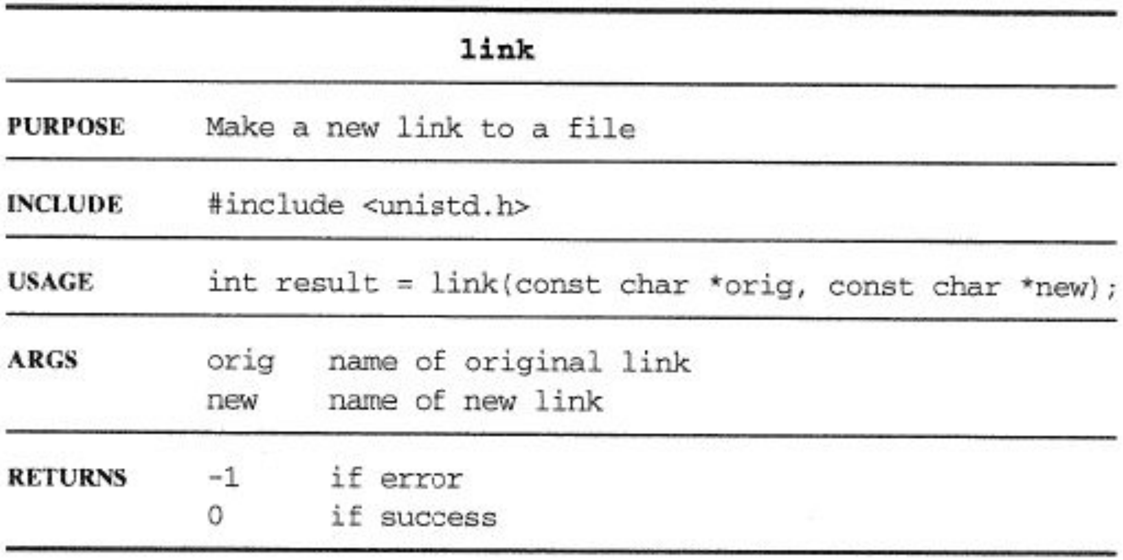


unlink

PURPOSE	Remove a directory entry
INCLUDE	`#include <unistd.h>`
USAGE	`int result = unlink(const char *path);`
ARGS	path name of directory entry to remove
RETURNS	-1 if error 0 if success

Commands and System Calls for Directory Trees (cont.)

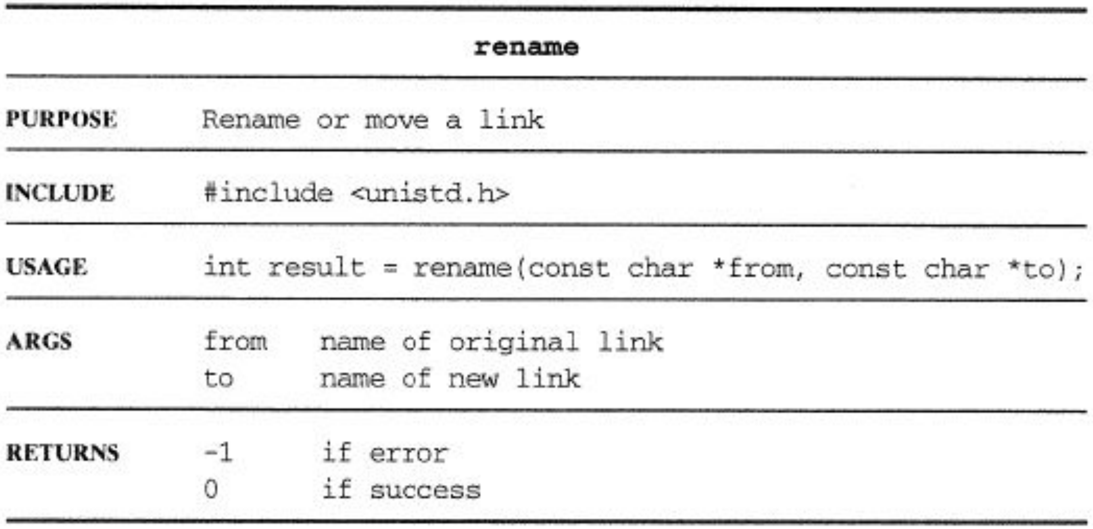
- `ln`: the command to create a link to a file
  - Uses `link()`
  - `link()`: makes a new link to an inode
    - The new link contains the inode number of the original link and has the name specified
    - If there is already a link with the new name, `link` will fail



	link
PURPOSE	Make a new link to a file
INCLUDE	#include <unistd.h>
USAGE	int result = link(const char *orig, const char *new);
ARGS	orig name of original link new name of new link
RETURNS	-1 if error 0 if success

Commands and System Calls for Directory Trees (cont.)

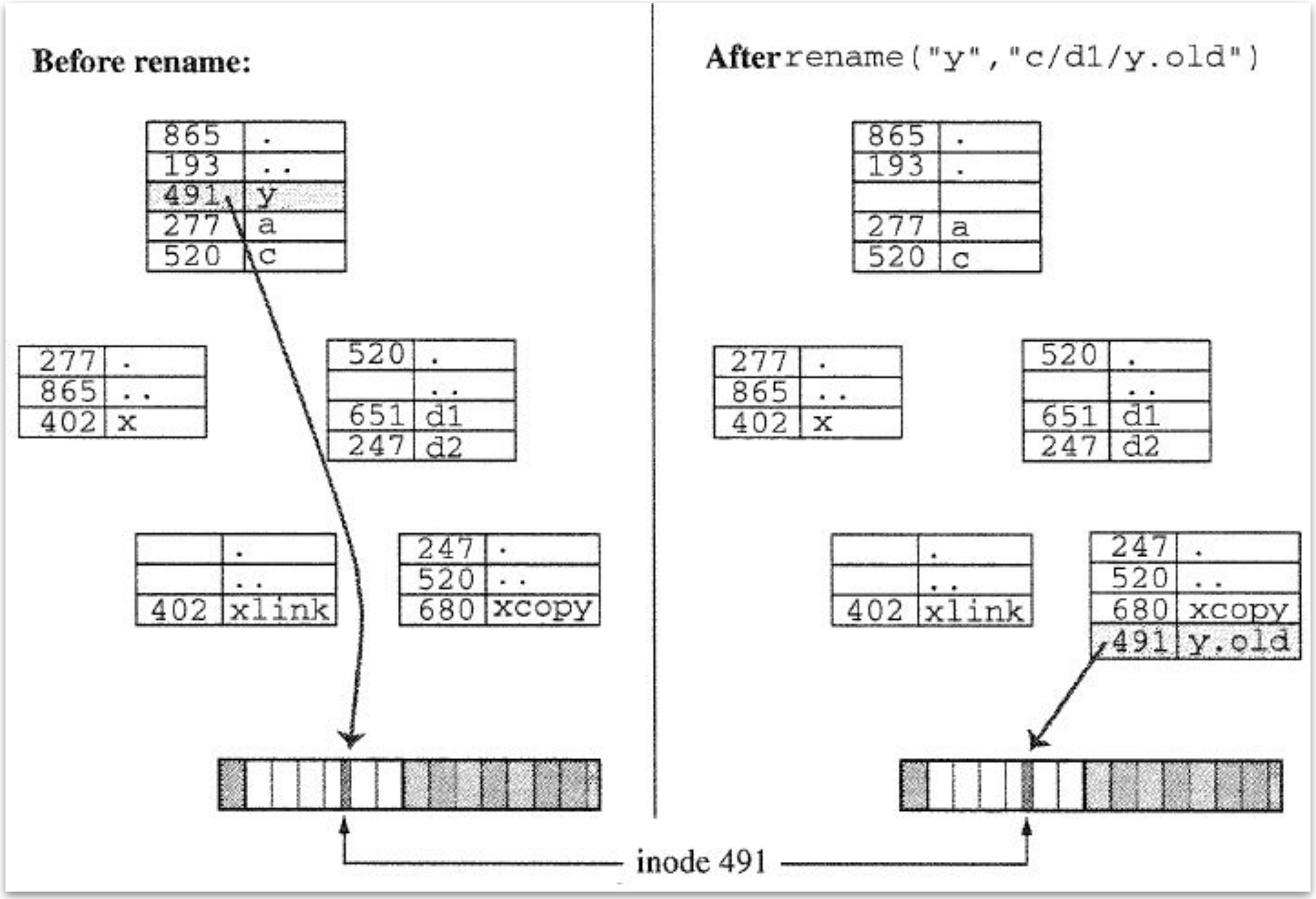
- `mv`: the command to change the name or location of a file or directory
  - Uses `rename()`
  - The basic logic of rename:
    - Copy the original link to new name and/or location
    - Then delete original link



	rename
PURPOSE	Rename or move a link
INCLUDE	`#include <unistd.h>`
USAGE	`int result = rename(const char *from, const char *to);`
ARGS	from name of original link to name of new link
RETURNS	-1 if error 0 if success

Commands and System Calls for Directory Trees (cont.)

- How `rename` works, why `rename` exists



Before rename:

865	.
193	..
491	y
277	a
520	c

277	.
865	..
402	x

520	.
	..
651	d1
247	d2

	.
	..
402	xlink

247	.
520	..
680	xcopy

After `rename("y","c/d1/y.old")`

865	.
193	.
277	a
520	c

277	.
865	..
402	x

520	.
	..
651	d1
247	d2

	.
	..
402	xlink

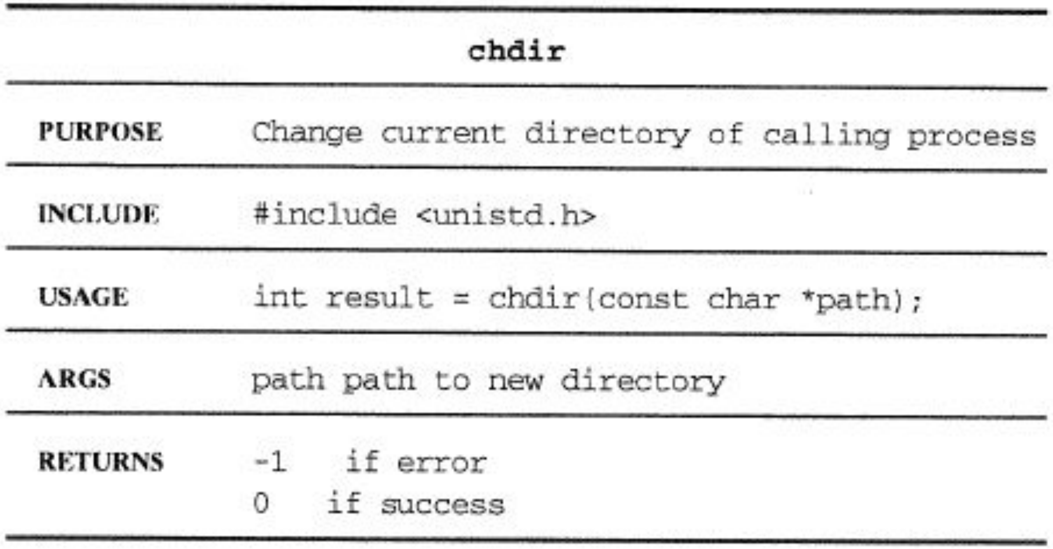
247	.
520	..
680	xcopy
491	y.old

Commands and System Calls for Directory Trees (cont.)

- Advantages of having `rename()`
  - Makes it possible to rename or relocate directories “safely.”
    - In old days, no regular users are allowed to `link` and `unlink` directories; no method of renaming directories
- Supports non-Unix file systems
  - Other systems may not work in the way of changing a link to rename a file or directory
  - The kernel hides all the details of implementation, such that the same code can be allowed to operate on all kinds of different file systems

Commands and System Calls for Directory Trees (cont.)

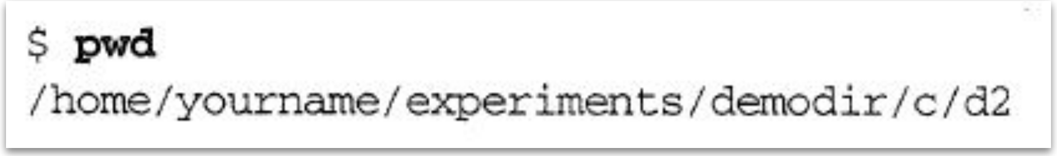
- `cd`: the command to change the current directory of a process
- Uses `chdir()`
  - Internally, the process keeps a variable that stores the inode number of the current directory
  - When you “change into a new directory,” you just change the value of that variable



	chdir
PURPOSE	Change current directory of calling process
INCLUDE	#include <unistd.h>
USAGE	int result = chdir(const char *path);
ARGS	path path to new directory
RETURNS	-1 if error 0 if success

Understanding pwd

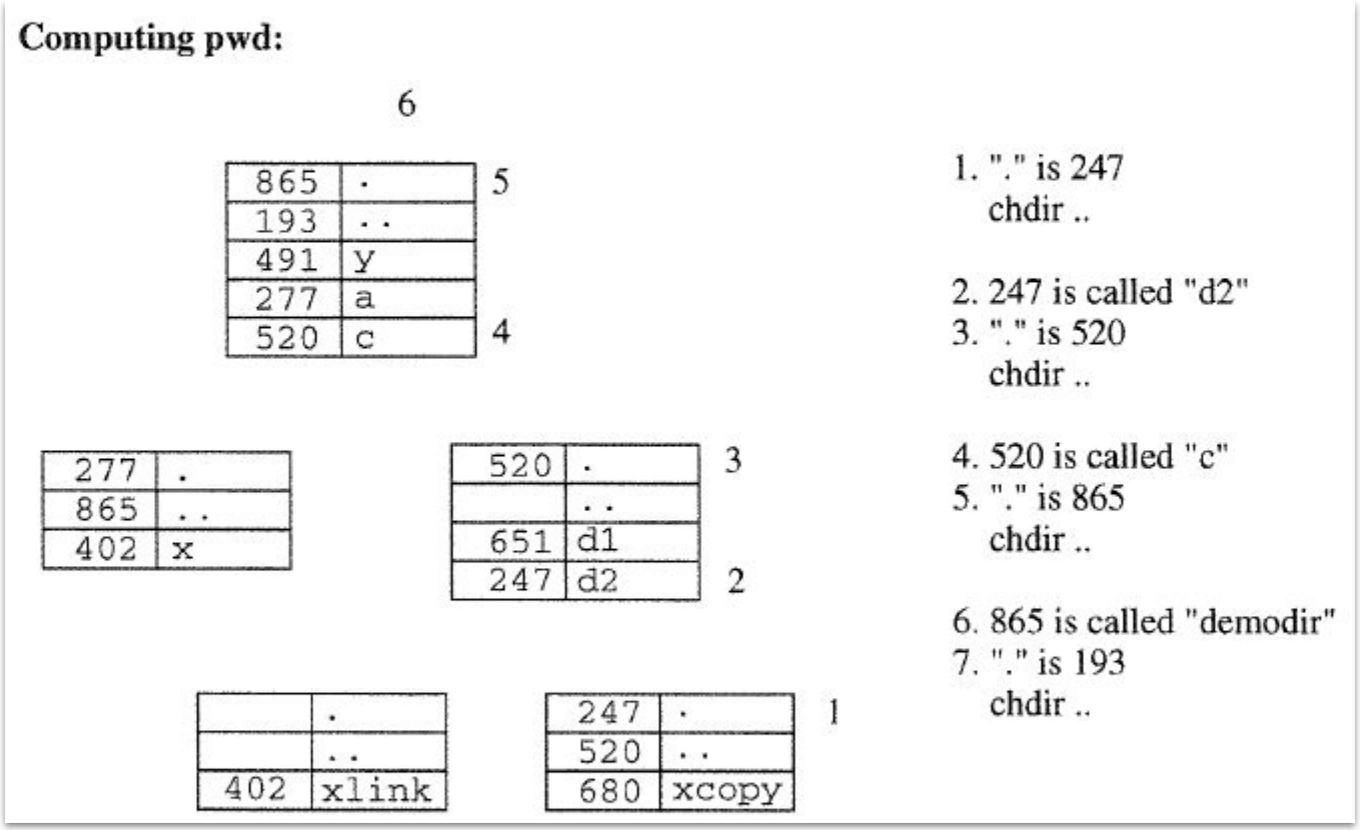
- Type pwd
  - What can you see?
  - Let’s first get into the ‘d2’ directory; then do pwd



```
$ pwd
/home/yourname/experiments/demodir/c/d2
```

How `pwd` Works

- Follow the links and read the directories



Computing pwd:

865	.
193	..
491	y
277	a
520	c

277	.
865	..
402	x

520	.
	..
651	d1
247	d2

	.
	..
402	xlink

247	.
520	..
680	xcopy

1. "." is 247
chdir ..

2. 247 is called "d2"
3. "." is 520
chdir ..

4. 520 is called "c"
5. "." is 865
chdir ..

6. 865 is called "demodir"
7. "." is 193
chdir ..

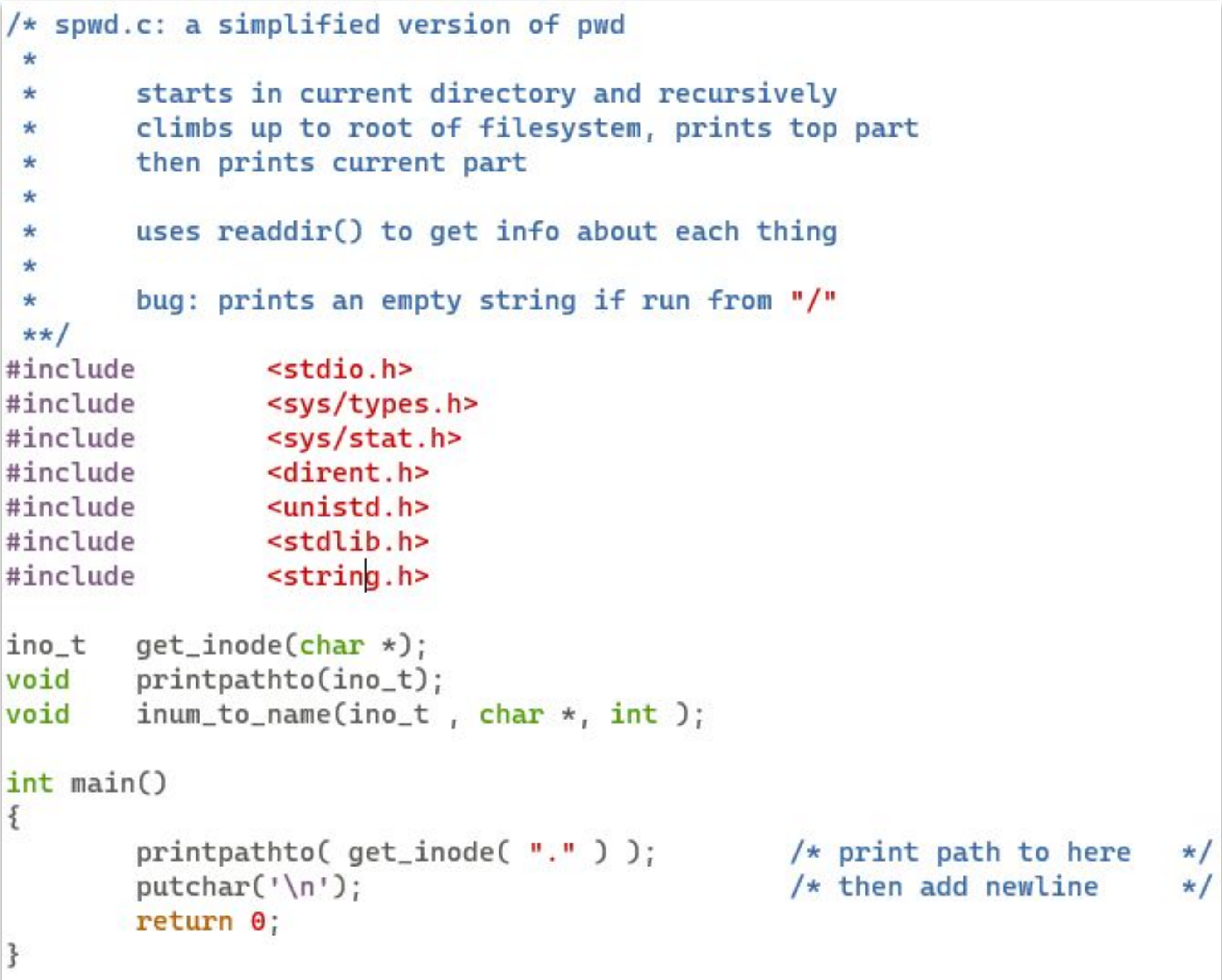
< Computing the current path >

How `pwd` Works (cont.)

- The algorithm is a repetition of these three steps:
  - Step 1: Find the inode number for “.”, call it *n*
    - use `stat()`
  - Step 2: Do “`chdir ..`”
    - use `chdir()`
  - Step 3: Find the name for the link with inode *n*
    - use `opendir(), readdir(), closedir()`
  - Repeat until you reach the top of the tree
- Q1) How do we know when we reach the top of the tree?
- Q2) How do we print the directory names in the correct order?

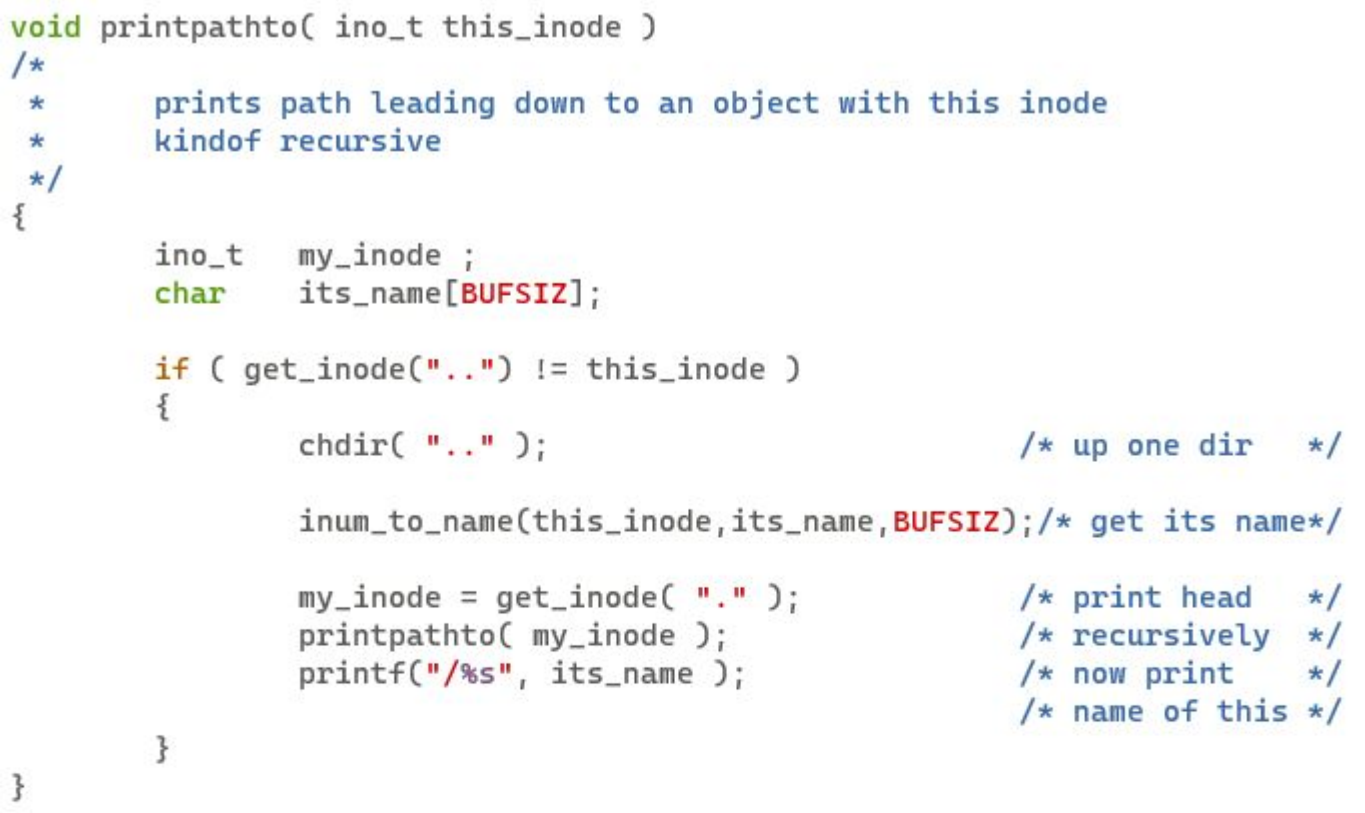
Writing pwd

- Simplified version (spwd.c)

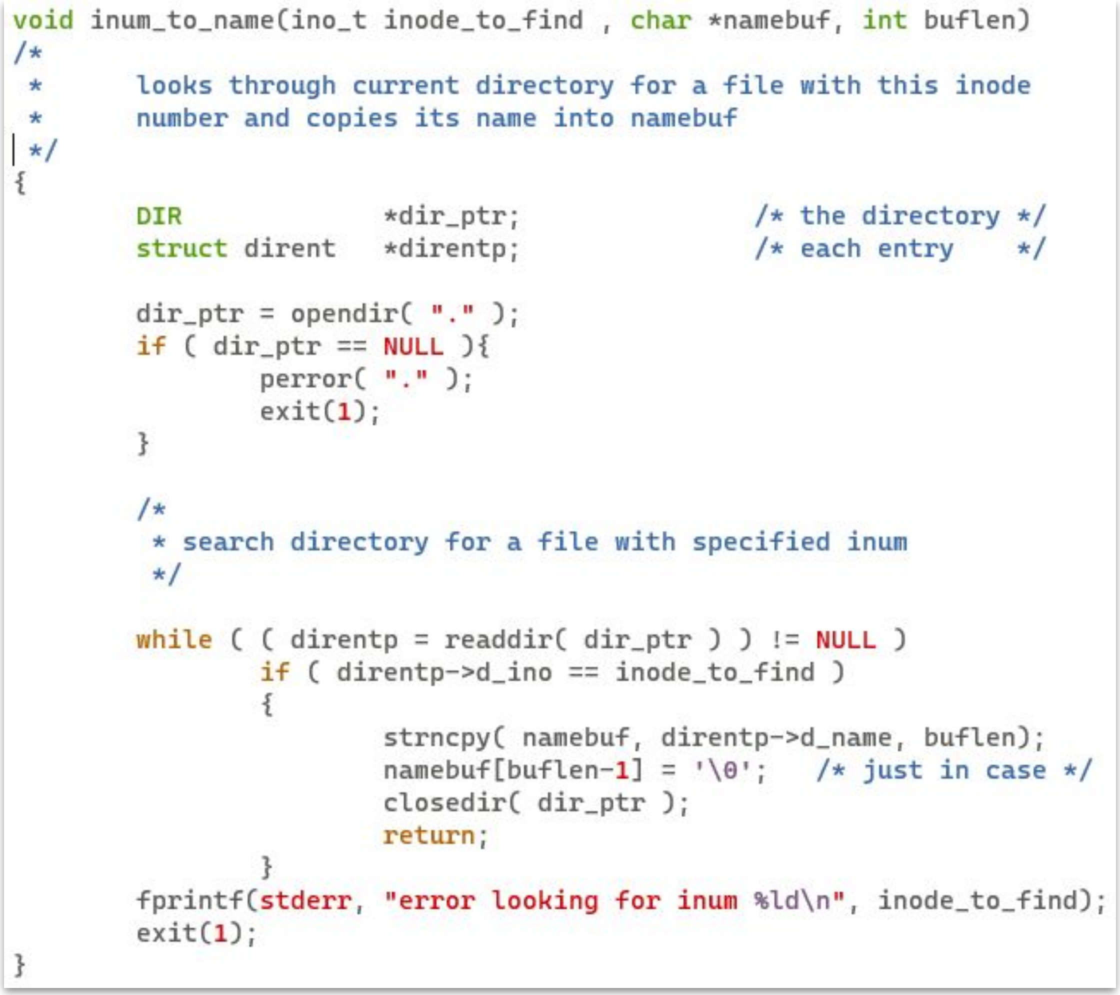


```
/* spwd.c: a simplified version of pwd
 *
 *      starts in current directory and recursively
 *      climbs up to root of filesystem, prints top part
 *      then prints current part
 *
 *      uses readdir() to get info about each thing
 *
 *      bug: prints an empty string if run from "/"
 **/
#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/stat.h>
#include        <dirent.h>
#include        <unistd.h>
#include        <stdlib.h>
#include        <string.h>

ino_t   get_inode(char *);
void    printpathto(ino_t);
void    inum_to_name(ino_t , char *, int );

int main()
{
        printpathto( get_inode( "." ) );        /* print path to here    */
        putchar('\n');                          /* then add newline      */
        return 0;
}
```

Writing pwd (cont.)

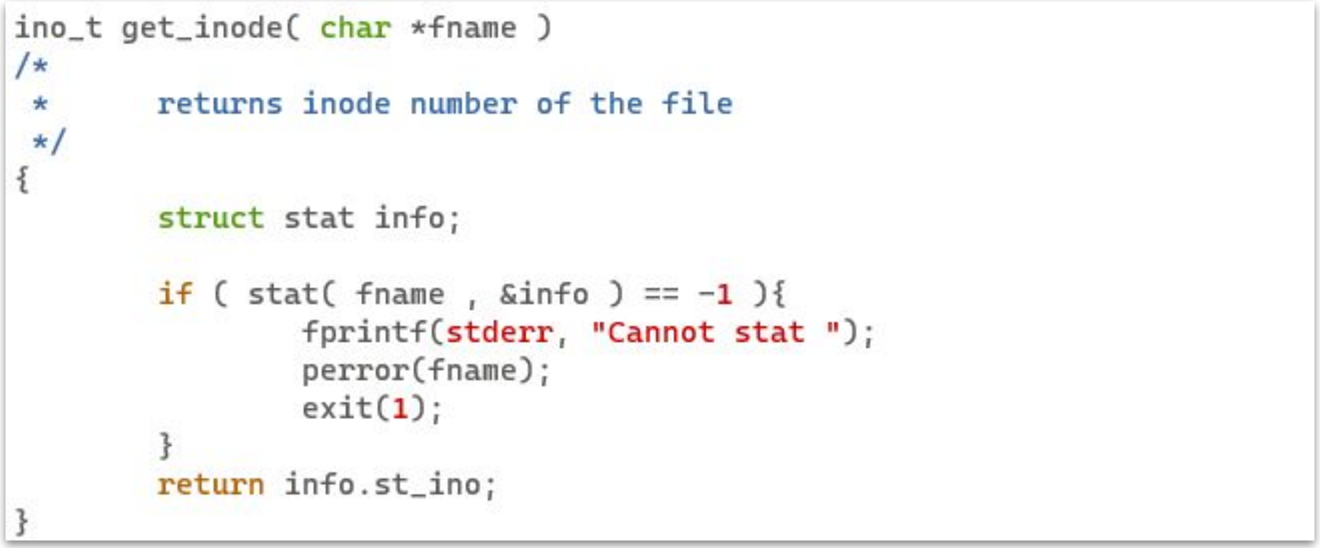


```
void printpathto( ino_t this_inode )
/*
 *      prints path leading down to an object with this inode
 *      kindof recursive
 */
{
        ino_t   my_inode ;
        char    its_name[BUFSIZ];

        if ( get_inode("..") != this_inode )
        {
                chdir( ".." );                  /* up one dir   */

                inum_to_name(this_inode,its_name,BUFSIZ);/* get its name*/

                my_inode = get_inode( "." );    /* print head   */
                printpathto( my_inode );        /* recursively  */
                printf("/%s", its_name );       /* now print    */
                                                /* name of this */
        }
}
```



```
void inum_to_name(ino_t inode_to_find , char *namebuf, int buflen)
/*
 *      looks through current directory for a file with this inode
 *      number and copies its name into namebuf
 */
{
        DIR             *dir_ptr;               /* the directory */
        struct dirent   *direntp;               /* each entry    */

        dir_ptr = opendir( "." );
        if ( dir_ptr == NULL ){
                perror( "." );
                exit(1);
        }

        /*
         * search directory for a file with specified inum
         */

        while ( ( direntp = readdir( dir_ptr ) ) != NULL )
                if ( direntp->d_ino == inode_to_find )
                {
                        strncpy( namebuf, direntp->d_name, buflen);
                        namebuf[buflen-1] = '\0';   /* just in case */
                        closedir( dir_ptr );
                        return;
                }
        fprintf(stderr, "error looking for inum %ld\n", inode_to_find);
        exit(1);
}
```

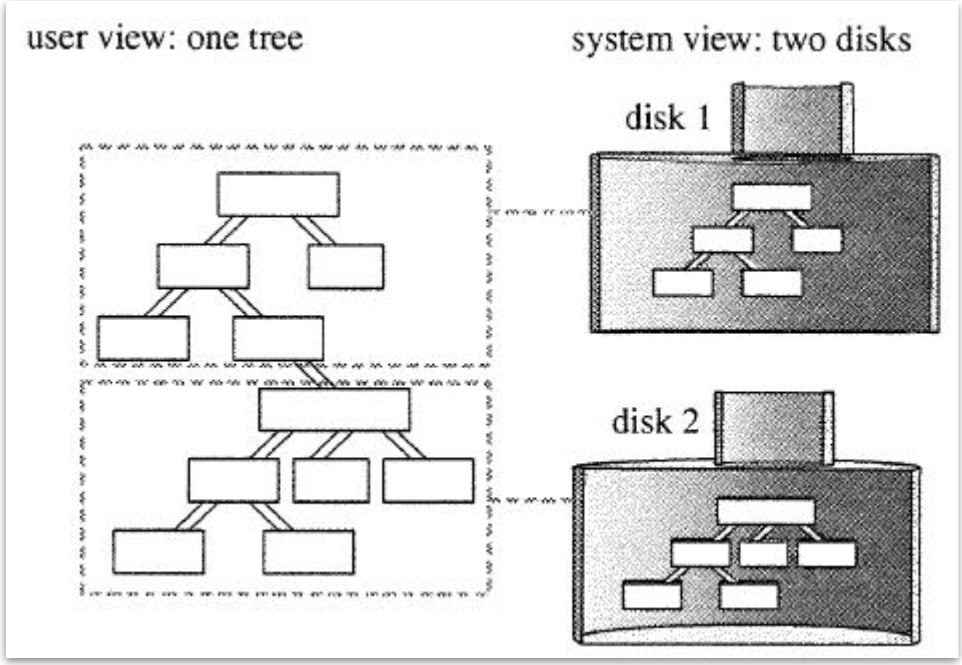
Writing `pwd` (cont.)



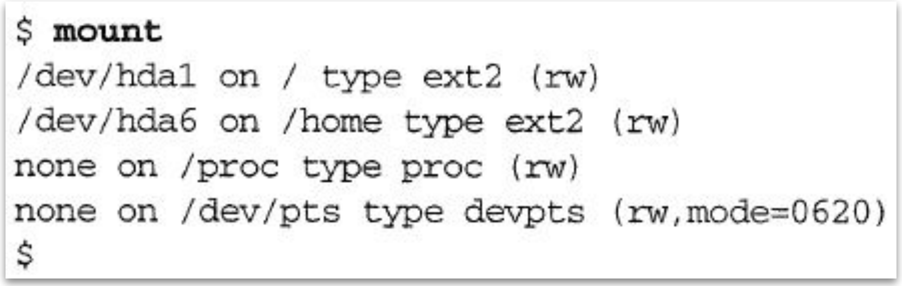
```c
ino_t get_inode( char *fname )
/*
 *      returns inode number of the file
 */
{
        struct stat info;

        if ( stat( fname , &info ) == -1 ){
                fprintf(stderr, "Cannot stat ");
                perror(fname);
                exit(1);
        }
        return info.st_ino;
}
```

Multiple File Systems: A Tree of Trees

- Each partition has its own file system tree
  - We can have the other file system attached to some subdirectory of the root file system
  - The kernel associates a pointer to disk 2's file system with a directory of disk 1's file system having the root
  - Of course, we can detach (unmount) Partition B from Partition A
    - e.g., external HDD / USB



Mount Points

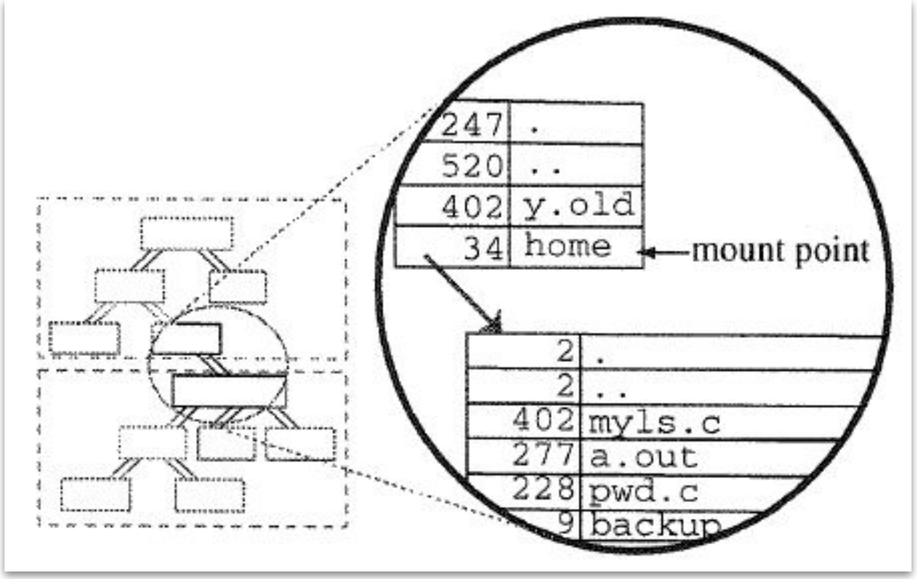


```
$ mount
/dev/hda1 on / type ext2 (rw)
/dev/hda6 on /home type ext2 (rw)
none on /proc type proc (rw)
none on /dev/pts type devpts (rw,mode=0620)
$
```

- Linux uses the phrase “to mount a file system”
  - “To pin it to some existing support.”
  - The root directory of the subtree is pinned onto a directory on the root file system
  - The directory to which the subtree is attached is called the mount point for that second file system
- The mount command:
  - Lists currently mounted file systems and their mount points

Duplicate Inode Numbers and Cross-Device Links

- Q) How can the kernel know which file with the inode number 402 to use while one of the two is mounted to another?
  - Suppose that two different disks have files with the same inode number, say 402
    - Several directories may have filenames associated with the inode number 402
  - It appears that these two links point to the same file, but they are actually not; they refer to two different files in its own filesystem



247	.
520	..
402	y.old
34	home

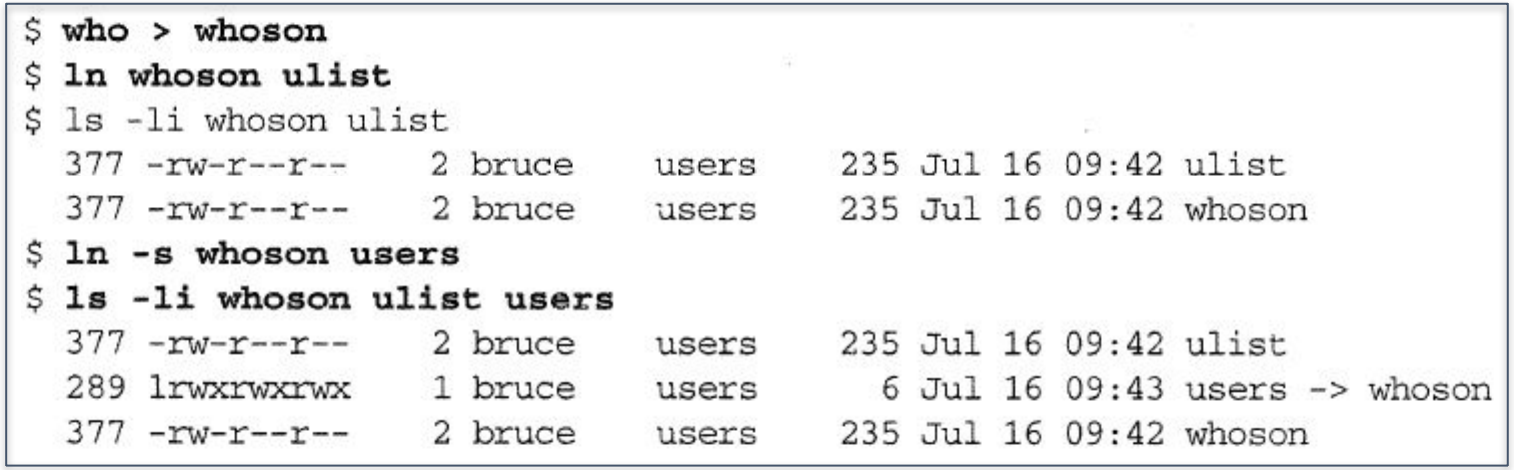
2	.
2	..
402	myls.c
277	a.out
228	pwd.c
9	backup

Duplicate Inode Numbers and Cross-Device Links (cont.)

- Hard links (that we’ve discussed so far)
  - Pointers that connect directories into a tree
  - Pointers that link filenames to the files themselves
    - Cannot point to inodes in other file systems
    - Note that even root cannot make hard links to directories in other file systems.
    - There’s no way of achieving pointing to the same file with hard links from one file system to another filesystem, and vice versa.
- Solution
  - Let’s use another type of link supported by Unix/Linux, ***soft link***
  - That way, in the current file system we can reach the file with the same inode number from another filesystem

Symbolic Links

- “Symbolic” (soft) links: “`ln -s`”
  - Refers to a file by “name” not by “inode” number
  - Similar to a shortcut in that it is a path name contained in a file
- This symbolic linked file (users) behaves like the original file (whoson)
  - It’s not the original file, though!

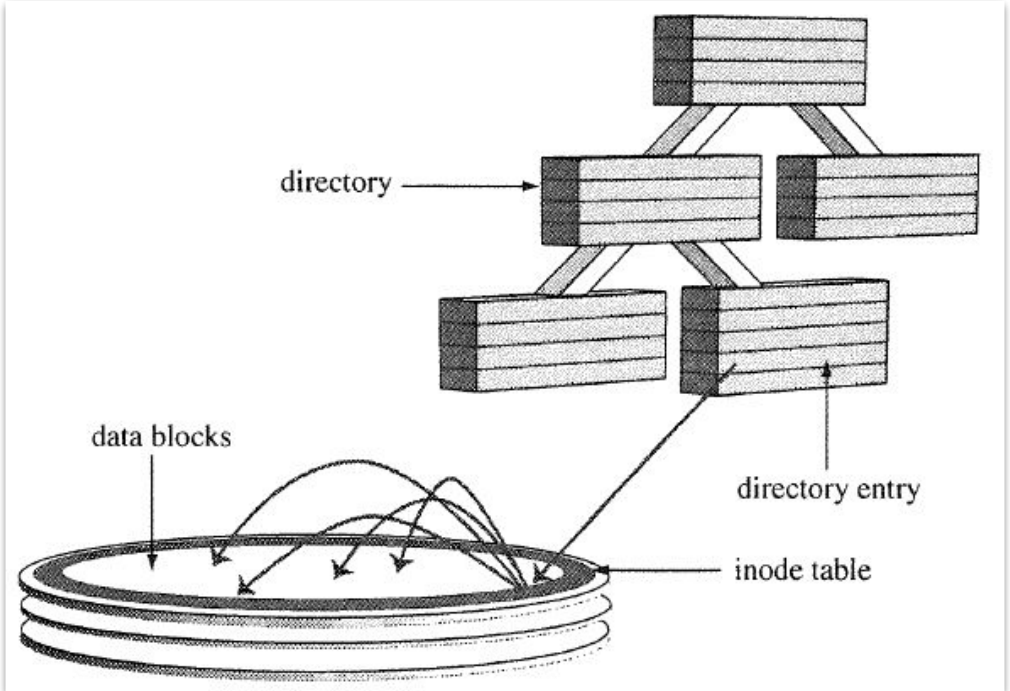


```
$ who > whoson
$ ln whoson ulist
$ ls -li whoson ulist
  377 -rw-r--r--    2 bruce    users    235 Jul 16 09:42 ulist
  377 -rw-r--r--    2 bruce    users    235 Jul 16 09:42 whoson
$ ln -s whoson users
$ ls -li whoson ulist users
  377 -rw-r--r--    2 bruce    users    235 Jul 16 09:42 ulist
  289 lrwxrwxrwx    1 bruce    users      6 Jul 16 09:43 users -> whoson
  377 -rw-r--r--    2 bruce    users    235 Jul 16 09:42 whoson
```

Symbolic Links (cont.)

- Symbolic links: relevant to `symlink` and `readlink` system calls
  - May "span" file systems, as they don't store the inode of the original file; it just keeps the reference to the original file by name
  - May "point to" directories (across different file systems)
  - Still suffers from the problems we discussed for the following conditions, the symbolic link will be broken:
    - If the file system containing the original file (pointed by a symbolic link) is removed (or unmounted)
    - If the original file name is changed
    - If a different file with that name is installed
  - But it's OK, though, as
    - We can check with the soft links for lost references or infinite loops in which the links point to parent directories.

Summary



- A directory entry is a filename and an inode number
- The inode number points to a struct on the disk
- That struct contains the file information and the data block allocation