CH04: Focus On File Systems
Writing pwd

Objectives

Ideas and Skills

- User's view of the Unix file system tree
- Internal structure of Unix file system: inodes and data blocks
- How directories are connected
- Hard links, symbolic links: ideas and system calls

Ideas and Skills

- How pwd works
- Mounting file systems

System Calls and Functions

- mkdir, rmdir, chdir
- link, unlink, rename, symlink

Commands

- pwd

Introduction

- Files contain data
- Directories are lists of files
- Directories are organized into a tree-like structure
- Directories can contain other directories

Introduction

- What does it mean to be “in a directory”?
- The tree-like structure is an abstraction, hard disks are a stack of spinning metal.
- How does this appear to be a tree of files and directories?
- Let's study the pwd command to find out.

Introduction

- *pwd* reports your current location within the directory tree.
- The sequence of directories and subdirectories from the top of the tree to your location is called the ***path*** to your working directory.
- To write path, we're gonna need to know about the file system.

A User's View of the File System

Directories and Files

- Users see a tree of directories.
- Each directory can contain files and other directories.

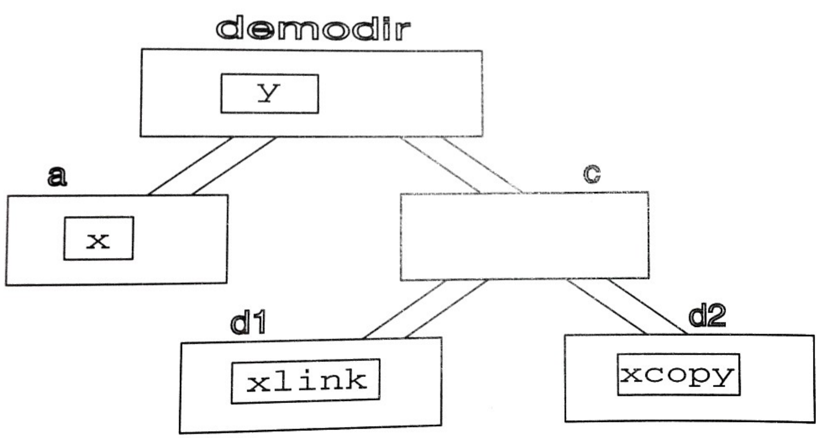


FIGURE 4.1

A tree of directories.

Directory Commands

- Let's build a tree:

```
hank@netbook: ~
hank@netbook:~$ mkdir demodir
hank@netbook:~$ cd demodir
hank@netbook:~/demodir$ pwd
/home/hank/demodir
hank@netbook:~/demodir$ mkdir b oops
hank@netbook:~/demodir$ mv b c
hank@netbook:~/demodir$ rmdir oops
hank@netbook:~/demodir$ cd c
hank@netbook:~/demodir/c$ mkdir d1 d2
hank@netbook:~/demodir/c$ cd ../..
hank@netbook:~$ mkdir demodir/s
hank@netbook:~$ 
```

Directory Commands

- *mkdir* creates a new directory or directories with the specified names
- *rmdir* removes a directory or directories
- *mv* renames a directory or moves a directory
- *cd* moves you from one directory to another

File Commands

- Let's create some files in this tree:

```
hank@netbook: ~
hank@netbook:~/demodir$ cp /etc/group x
hank@netbook:~/demodir$ cp x copy.of.x
hank@netbook:~/demodir$ mv copy.of.x y
hank@netbook:~/demodir$ mv x a
hank@netbook:~/demodir$ cd c
hank@netbook:~/demodir/c$ cp ../a/x d2/xcopy
hank@netbook:~/demodir/c$ ln ../a/x d1/xlink
hank@netbook:~/demodir/c$ ls > d1/xlink
hank@netbook:~/demodir/c$ cp d1/xlink z
hank@netbook:~/demodir/c$ rm ../../demodir/c
/d2/../z
hank@netbook:~/demodir/c$ cd ../..
hank@netbook:~$
```

File Commands

- *cp* makes a copy of a file
- *mv* renames a file or moves a file to a different directory
- *rm* deletes files
- “**..**” stands for the directory one level up, the *parent directory*

File Commands

- A sequence of slash-separated directory names specifies a path to follow that leads to the named object
- *ln* creates a link to a file (a pointer to a file)

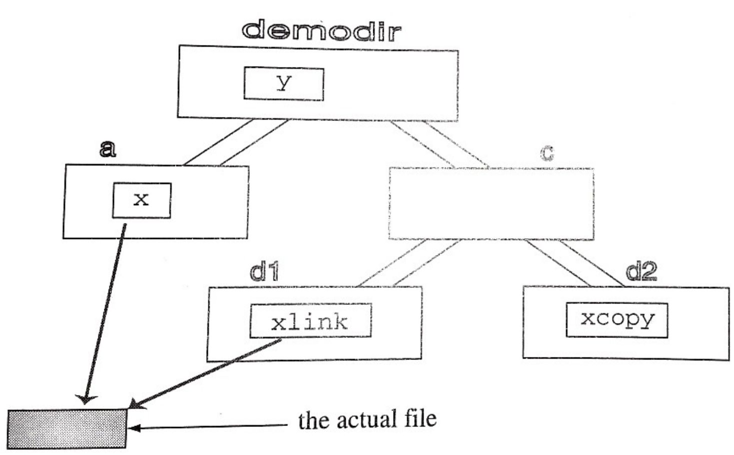


FIGURE 4.2
Two links to the same file.

Tree Commands

- *ls – R* tells ls to list the contents of the specified directory and all its subdirectories
- *chmod -R* tells chmod to apply permission changes to all files in subdirectories
- *du* reports the number of disk blocks used by a directory, the files it contains, and all files below it.
- *find* searches a directory and all subdirectories for specified items

Almost No Limits to Tree Structures

- Internal structure of the system imposes no limit on the depth of a directory tree.
- But, you can create directories so deep they exceed capacity of many commands that operate on trees.

Internal Structure of the Unix File System

Note...

- A disk is a stack of magnetic platters.
- Some levels of abstraction convert that stack into the file system we explored.

Abstraction 0: From Platters to Partitions

- Disks can store a lot of data.
- They can be divided into separate regions called *partitions*.
- Each partition is treated as a separate disk.

Abstraction 1: From Platters to an Array of Blocks

- A disk is a stack of magnetic platters.
- The surface of each platter is organized into concentric circles called *tracks*.
- Each track is divided into *sectors*.

Abstraction 1: From Platters to an Array of Blocks

- Each sector stores a number of bytes, 512 for example.
- On older hardware, inner tracks had fewer sectors than outer tracks due to less surface area. Not a problem with newer hardware.

Abstraction 1: From Platters to an Array of Blocks

- The sector is the basic unit of storage on the disk.

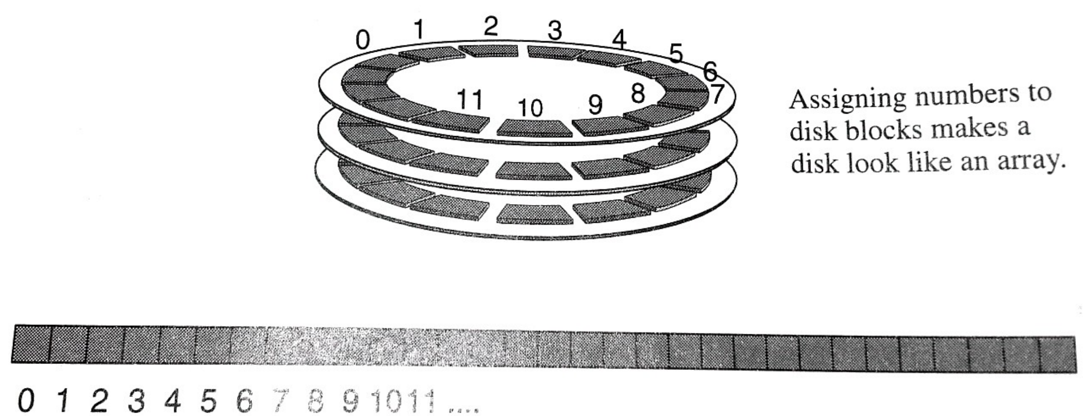


FIGURE 4.3
Assigning numbers to disk blocks.

Abstraction 1: From Platters to an Array of Blocks

- Disk sectors are also known as *blocks*.
- The blocks are assigned numbers.
- This numbering system allows us to treat a disk as an <u>array of blocks</u>.

Abstraction 2: From an Array of Blocks to Three Regions

- A file system stores files: it's contents, properties, and directories that hold them.
- Where in this sequence of blocks are the contents, properties, and directories?

Abstraction 2: From an Array of Blocks to Three Regions

- Unix divides the array of blocks into three sections.

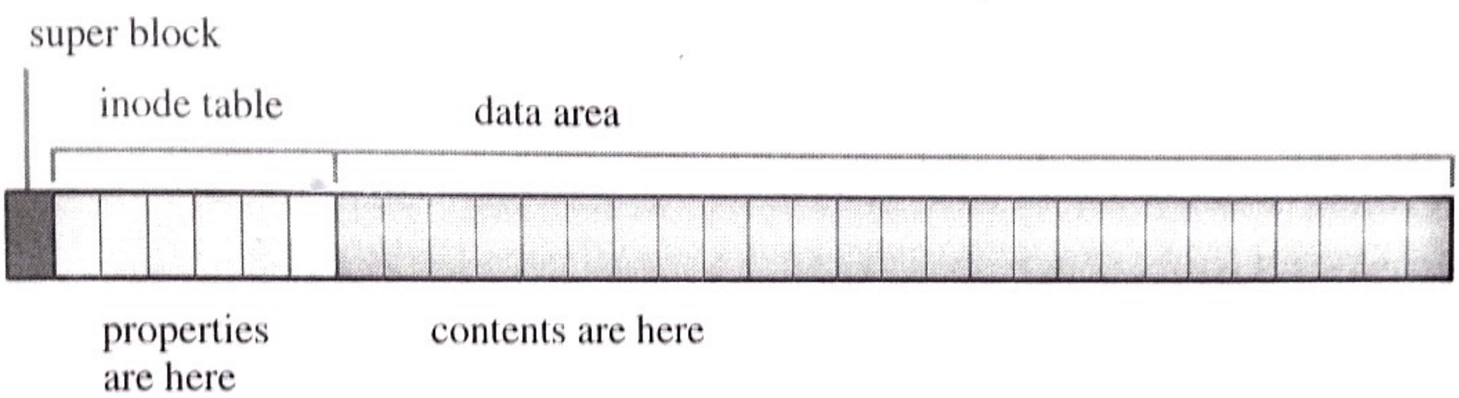


FIGURE 4.4

The three regions of a file system.

Abstraction 2: From an Array of Blocks to Three Regions

- The *data area* holds the contents of the files
- The *inode table* holds the file properties
- The *superblock* holds information about the file system itself.

The Superblock

- The first block in the file system. Contains information about the organization of the file system itself.
- For example, records the size of each area.
- Also holds info about location of unused data blocks.

The Inode Table

- Each file has a set of properties: size, owner user ID, etc.
- These properties are stored in a struct called an *inode*.
- All inodes are the same size.
- The inode table is an array of these structs.
- Every file has an inode in the table.
- Each inode is identified by its position in the table.

The Data Area

- Contents of files are kept here.
- All blocks on the disk are the same size.
- If a file has more bytes than a block can hold, the contents are stored in as many blocks as needed.
- A large file can occupy thousands of disk blocks.
- How are the blocks kept track of?

The file system in practice: creating a file

- Consider the command *who > userlist*
- When the command finishes, there's a new file containing the output of who.
- The kernel has to store the contents in the data area, the properties in an inode, and the name in a directory.

The file system in practice: creating a file

- Here's an example of creating a file requiring three blocks of storage:

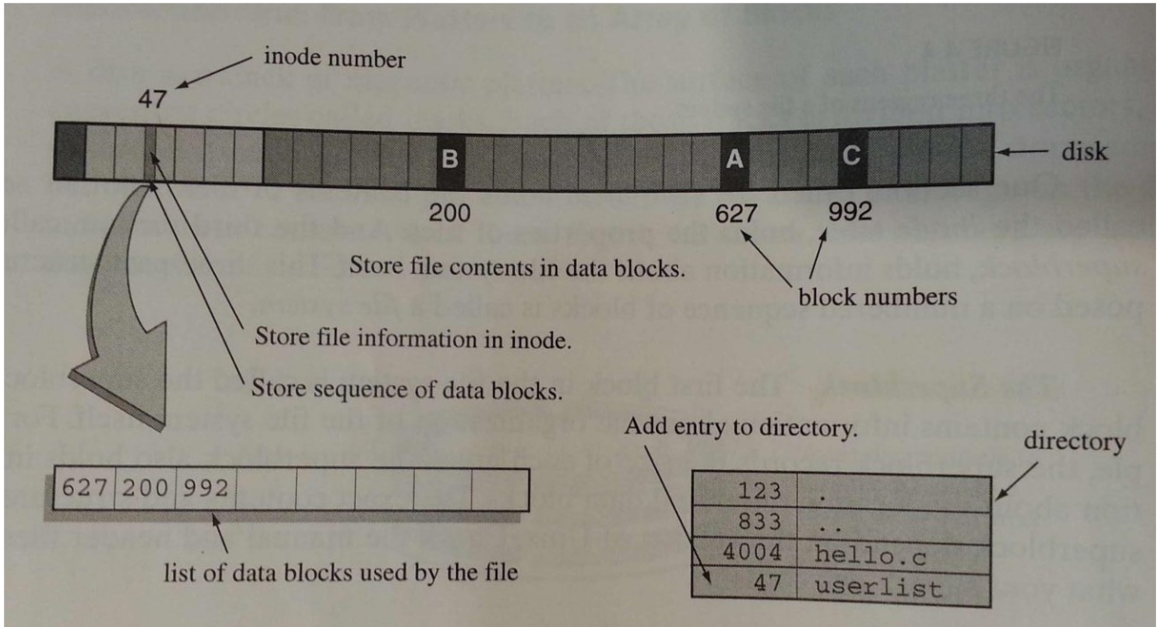


FIGURE 4.5

Internal structure of a file.

The file system in practice: creating a file

- Creating a new file involves the following four main operations:
- ***Store properties***
  The kernel locates a free inode (here, 47). It then stores the file info in this inode.
- ***Store Data***
  The kernel locates 3 blocks from its free blocks list(627, 200, 992).

The file system in practice: creating a file

- ***Record Allocation***
  File contents are in blocks 627, 200, 992 (in that order).

  The kernel records that sequence in disk allocation section of the inode (an array of block numbers).

  Stored in first 3 locations.

The file system in practice: creating a file

- ***Add Filename to Directory***
  The new file is called userlist.

  The kernel adds (47, userlist) do the directory.

  This association between file name and inode number is the link connecting the file name and contents and properties of a file.

The file system in practice: How directories work

- A directory is a special file that contains a list of names of files. Here's an abstract model:

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

Looking Inside a Directory

- Let's see the directory contents

```
hank@netbook: ~
hank@netbook:~$ ls -1ia demodir
403333 .
391684 ..
400613 a
403393 c
422507 d1
422506 d2
422176 s
422508 y
hank@netbook:~$
```

- A list of filenames and their inode numbers

Looking Inside a Directory

- So, the info about the size, owner, group, etc about the current directory is in struct number 403333 in the inode table.

Multiple Links to the Same File

- Check this out:

```
hank@netbook: ~/demodir
hank@netbook:~/demodir$ ls -i
400613 a   422507 d1   422176 s   422508 z
403393 c   422506 d2   422508 y
hank@netbook:~/demodir$
```

- y and z have the same inode number: 422508.
- Both filenames refer to the same inode.

Multiple Links to the Same File

- The inode *is* the file; it contains the properties and list of data blocks for the file.
- y and z are two names for the same file.

Multiple Links to the Same File

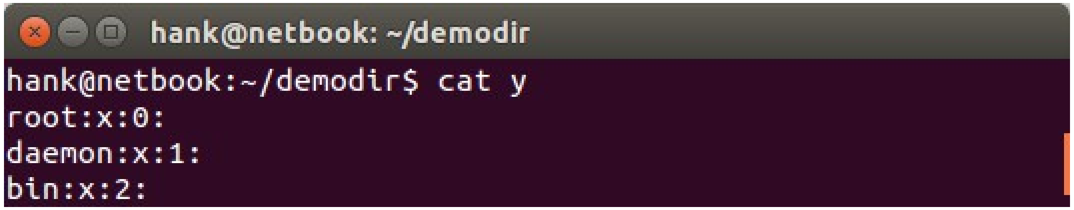
- Now look at this:

```
hank@netbook: ~/demodir
hank@netbook:~/demodir$ ls -ia /
      2 .                     261122 mnt
      2 ..                   1305603 opt
 913921 bin                        1 proc
 522241 boot                 1044482 root
 391683 cdrom                   1235 run
      3 dev                   913922 sbin
```

- In the root directory, . and .. have the same inode number. How can current be the same as parent? mkfs creates a file system and sets the parent of the root directory to point to itself

The File System in Practice: How cat Works

- What happens when you read from a file?



- Let's follow the pointers back from the directory to the data

The File System in Practice: How cat Works

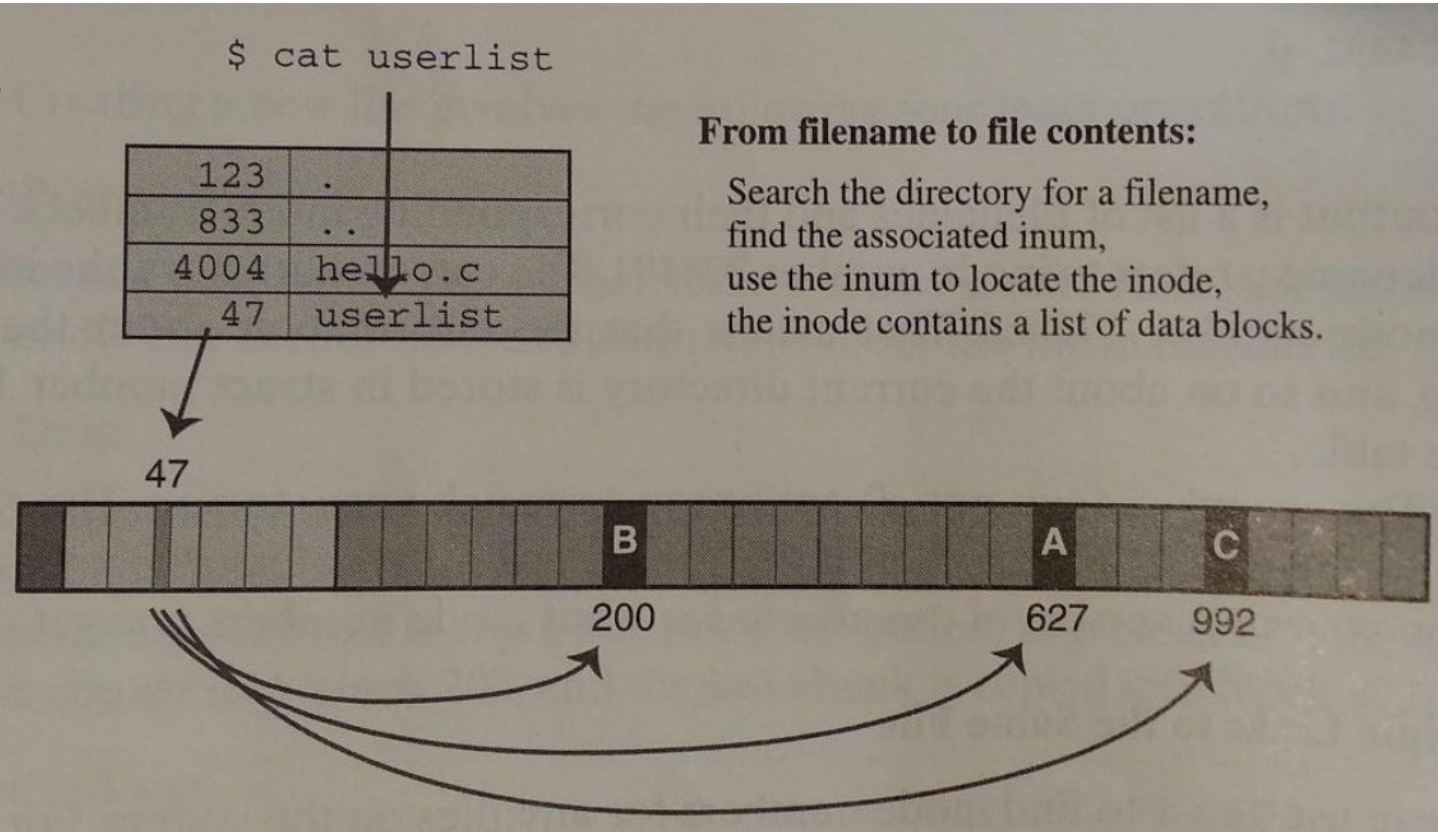


FIGURE 4.6

From filename to disk blocks.

Search the directory for the filename

- Filenames are stored in directories.
- The kernel searches for the entry containing string userlist.
- The matching entry contains inode number 47.

Locate and Read Inode 47

- kernel locates inode 47 in the file system inode region.
- Simple calculation to find it: all inodes are the same size, and each disk block contains a fixed number of inodes.
- Inode might already be in a kernel buffer.
- Inode contains a list of data blocks.

Go to the Data Blocks, One by One

- Kernel now knows which data blocks contain the file contents and their order.
- As cat is repeatedly calling *read*, the kernel is going through each data block, copying bytes from disk to its buffers and back to the array in user space.
- A call to open looks in the directory for the filename, then uses the inode number in the directory to get the file properties and locate the contents.

Go to the Data Blocks, One by One

- What happens when open attempts to open a file a user doesn't have read or write permissions to?
- Kernel uses the filename to get the inode number, then uses the inode number to find the inode.
- There, the kernel gets the permission bits, the user ID of the file owner.

Go to the Data Blocks, One by One

- If your user ID, the user ID for the file and permission bits don't allow access, *open* returns -1 and sets errno to EPERM.

Inodes and Big Files

- How does Unix deal with really big files? Previous explanation leaves out some stuff...
- Here's the problem.
  fact 1 Large files require many blocks.
  fact 2 inode stores block allocation list
  problem How can fixed-sized inode store a long allocation list?

Inodes and Big Files

- Solution: Store most of the allocation list in data blocks, and leave pointers to those blocks in the inode.

Inodes and Big Files

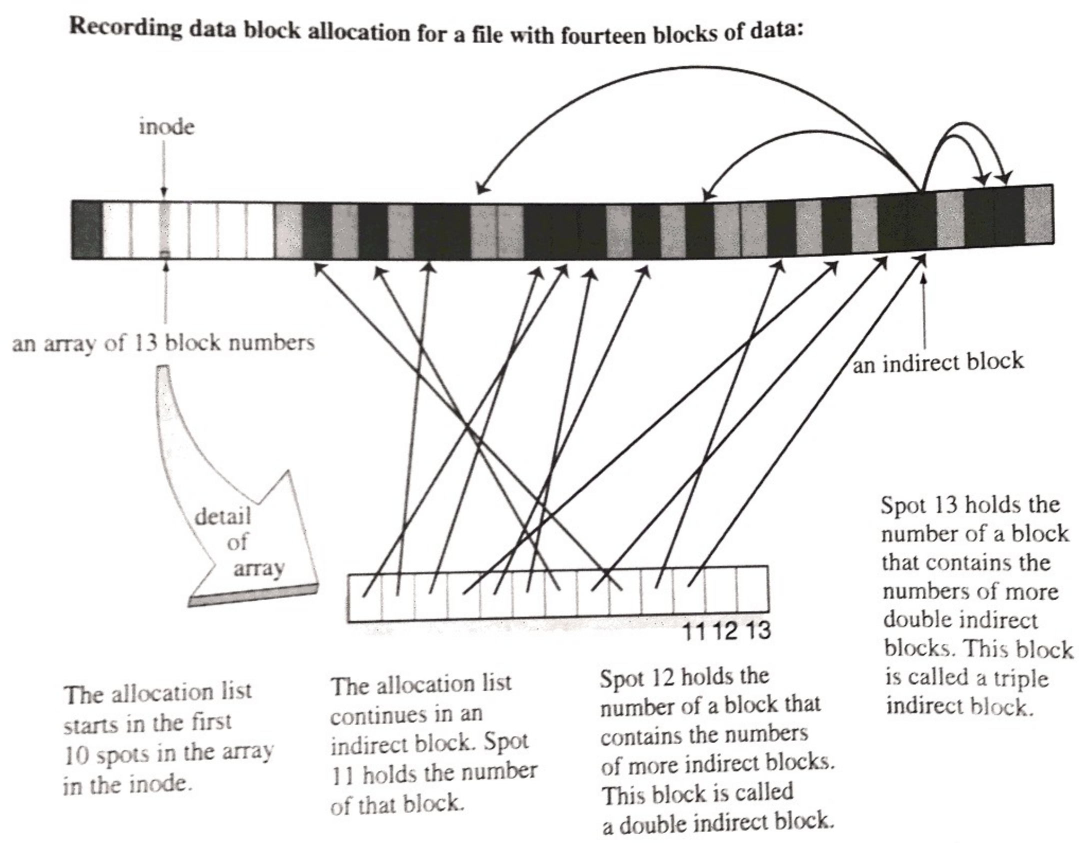


FIGURE 4.7

Block allocation list continues in data region.

Inodes and Big Files

- In this figure, the file requires 14 blocks.
- The allocation list contains 14 block numbers, but there are only 13 spots.
- Solution: Put the first 10 numbers in the inode, and the last 4 in a block.

Inodes and Big Files

- The first 10 spots hold the block numbers for the first 10 blocks.
- Put the other 4 block numbers into a new data block.
- Then, put the new data block holding the block numbers in the 11th spot.

Inodes and Big Files

- Here, note the file actually uses 15 blocks.
- 14 blocks contain the content of the file, and 1 block contains the part of the allocation list that didn't fit in the inode.
- This overflow block is called an *indirect block*.

What happens when the indirect block fills?

- As the file gets bigger, the allocation list gets longer.
- Eventually, the allocation list overflows the indirect block.
- The kernel starts a second indirect block.
- Where does the kernel put the block number of the second indirect block?

What happens when the indirect block fills?

- Instead of putting it in spot 12 in the inode, it puts it inside a new block that is used to hold the block numbers of these new indirect blocks.
- So, spot 12 holds the block number of the block that stores the block numbers of the second, third, fourth, and subsequent overflow blocks.
- This block is called a *double indirect block.*

What happens when the double indirect block fills?

- The kernel starts a new double indirect block.
- The kernel doesn't put the new block's number into the inode array.
- Instead, a *triple indirect block* is created to hold the numbers of the new double indirect block and all future double indirect blocks.

What happens when the double indirect block fills?

- The number of this triple indirect block is recorded in the inode array.

What happens when the triple indirect block fills?

- Well, you're *screwed*. The file has reached its limit.
- If you want huge files, then you have to set your file system up differently, you need to set bigger block sizes.
- The disk allocation system is quick and efficient for small files.

What happens when the triple indirect block fills?

- For large files, the kernel uses more disk space to hold the larger allocation list.
- Seeking a particular file can require fetching several indirect blocks just to get the number of the data block.

Understanding Directories

Understanding Directory Structure

- In what sense is a file *in a directory*?
- What does it mean in technical terms to say “d1 is a subdirectory of c”?
- Internally, a directory is a file that contains of list of pairs: filename and inode number. That's it.
- Users see filenames, Unix sees pointers.

Understanding Directory Structure

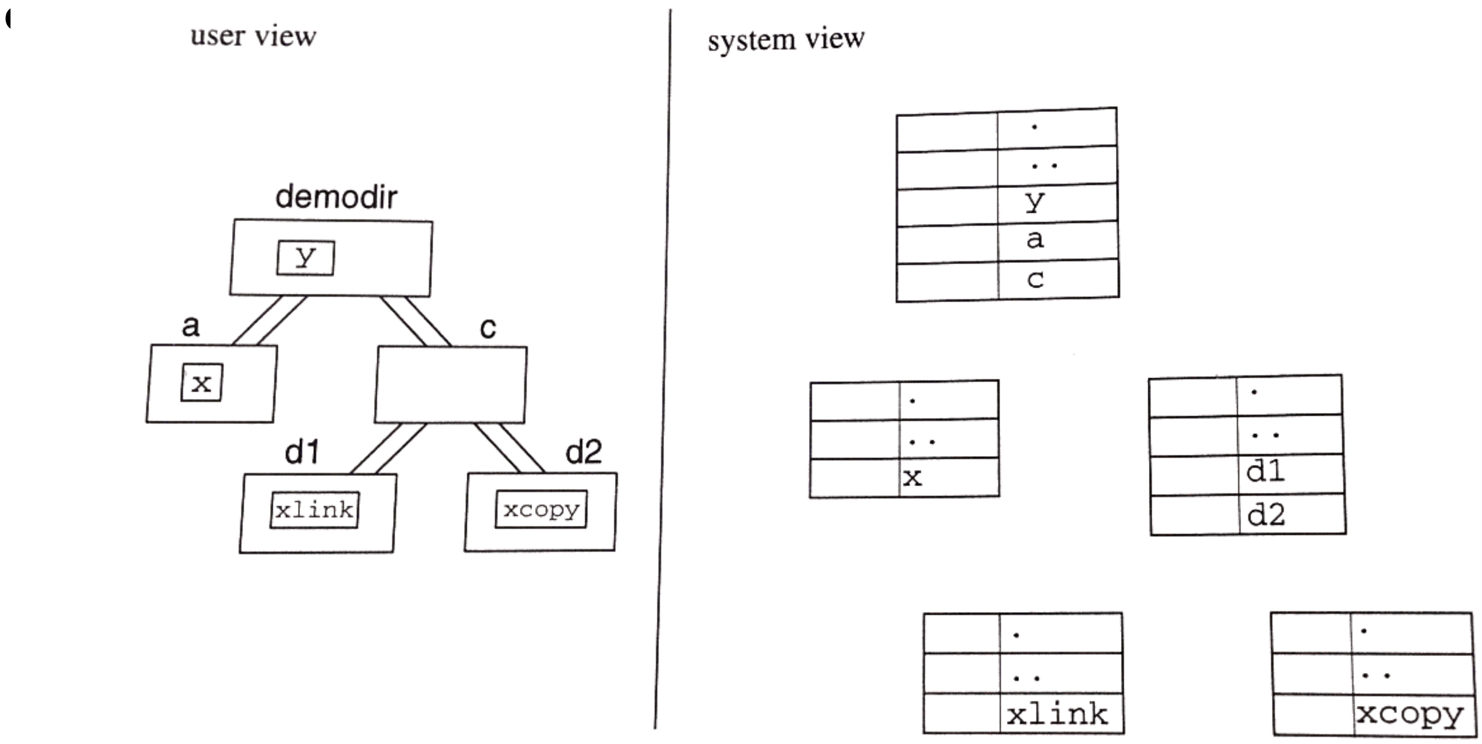


FIGURE 4.8

Two views of a directory tree.

Understanding Directory Structure

- How do we translate from one diagram to the other?
- Filling in the numbers allows us to see how the directory tree is held together.
- If we use ls -iaR, we can list inode numbers for all files recursively down a tree:

Understanding Directory Structure

```
$ 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
$
```

Figure 4.9 is the diagram with most of the inode numbers filled in:

Understanding Directory Structure

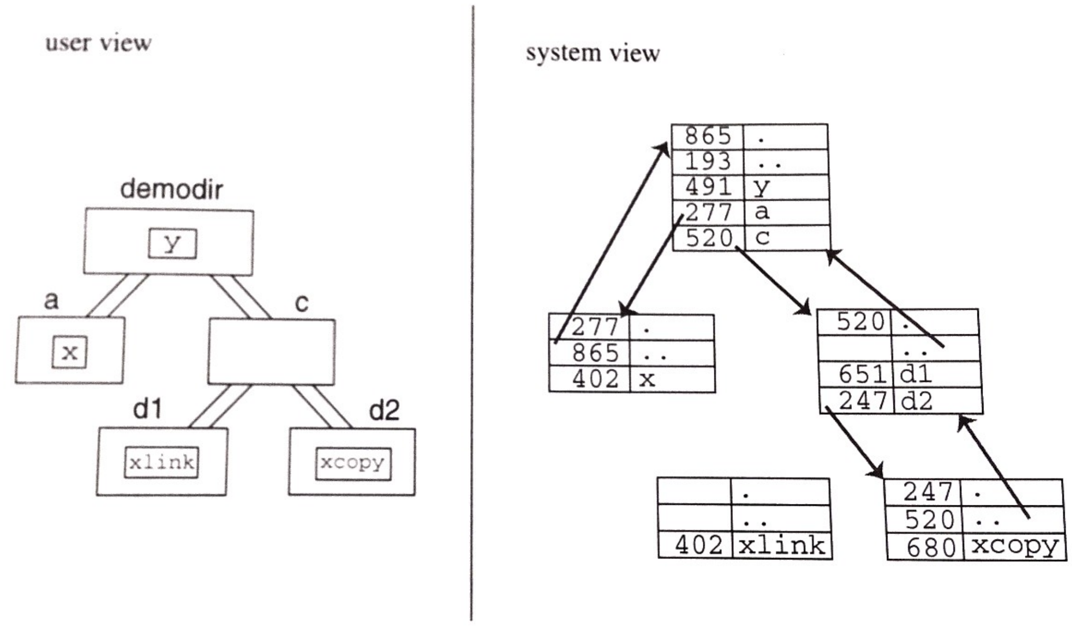


FIGURE 4.10

Directory names and pointers to directories.

Understanding Directory Structure

- From the above figure, we see that file y is in directory demodir in the user view.
- In system view, we see the directory a has an entry with filename x and inode number 402.
- So, to say a file x is in a directory a means there is a link to inode 402 in the directory called a, and the filename attached to that link is x.

Understanding Directory Structure

- Notice also, the directory labeled d1 also contains a link to inode 402.
- This means, that both x and xlink refer to the exact same file.

Understanding Directory Structure

- In summary, directories contain references to files.
- Each reference is called a *link*.
- The contents of the file are in data blocks.
- The file properties are in a struct in the inode table.
- The inode number and a name are stored in a directory.
- All this also applies to a directory having a subdirectory

Understanding Directory Structure

- In system view, the . refers to the directory itself.
- So, directory a, has an inode number of 277.
- .. refers to the parent directory. Directory a has an innode number of 865 for it's ..
- That inode number is the umber of demodir, the directory it's in.

Understanding Directory Structure

- Note both x and xlink have the same inode number.
- So, which one is the original file and which is the link?
- In Unix, they have the same status, they are called *hard links* to the file.
- The file is an inode and a bunch of data blocks; a link is a reference to an inode.

Understanding Directory Structure

- You can make many links to the same file.
- The kernel counts the number of links.
- In the example for inode 402, it's at least 2.
- The *link count* is stored in the inode, and a member of the struct stat returned by system call stat.

Understanding Directory Structure

- In Unix, files don't have names, links have names.
- Files have inode numbers.

Commands and System Calls for Directory Trees

mkdir

- The *mkdir* command creates new directories.
- It accepts one or ore directory names at the command line.
- The *mkdir* command uses the *mkdir* system call:

mkdir

	mkdir	
PURPOSE	Create a directory	
INCLUDE	#include <sys/stat.h> #include <sys/types.h>	
USAGE	int result = mkdir(char* pn, mode_t mode	
ARGS	pn	name of new directory
	mode	mask for permission bits
RETURNS	-1	if error
	0	if success

mkdir

- *mkdir* creates and links a new directory node to the file system tree.
- That is, it creates the inode for the directory, allocates a disk block for its contents, installs the two entries . and .. with inode numbers set to the correct values, and adds a link to that node to its parent directory.

rmdir

- *rmdir* command deletes a directory.
- It accepts one or more directory names at the command line.
- The *rmdir* command uses the *rmdir* system call:

rmdir

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

rmdir

- *remdir* removes a directory node from a directory tree.
- The directory must be empty (except for dot and dotdot).
- May not contain any files or subdirectories.
- The link to the directory is removed from its parent directory.
- If the directory itself isn't being used by another process, the inode and data blocks are freed.

rm

- *rm* command removes entries from a directory.
- *rm* accepts one ore more filenames on the command line.
- *rm* command uses the *unlink* system call:

unlink

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

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.
- If there are other links to the inode, the data blocks and inode are otherwise untouched.
- Can't be used to unlink directories.

ln

- *ln* command creates a link to a file.
- *ln* uses the *link* system call:

link

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

link

- *link* makes a new link to an inode.
- The new link contains the inode number of the original link and has the specified name.
- If a link exists with the new name, *link* will fail.
- Nobody is allowed to use *link* to make new links to directories.

mv

- *mv* command changes the name or location of a file or directory.
- Very flexible command, will look at internal details later.
- Many cases, it just uses the *rename* system call:

rename

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 the original link
	to	name of new link
RETURNS	-1	if error
	0	if success

rename

- *rename* changes the name or location of a file or directory.
- *rename*(“y”, “y.old”) changes the name of the file
- *rename*(“y”, “c/d2/y.old”) changes the name and location of the file.
- Can be used for directories and files.
- Can't move a directory into one of its subdirectories.
- Deletes an existing file or empty directory

cd

- *cd* command changes the current directory of a process.
- *cd* affects the process, not the directory.
- *cd* uses the *chdir* system call:

chdir

	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

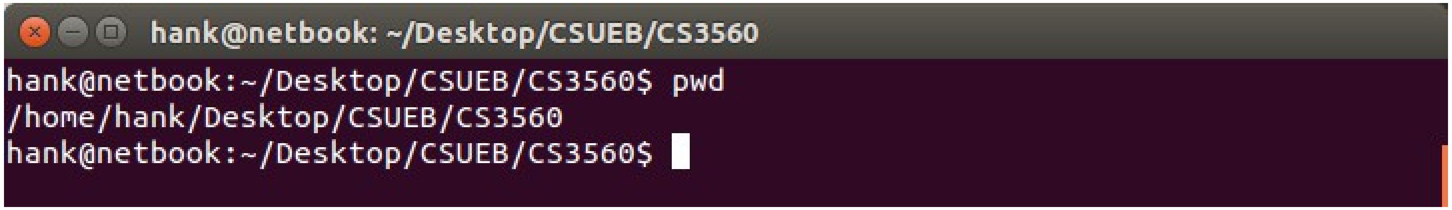
chdir

- Each running program on Unix has a current directory.
- The *chdir* system call changes the current directory of the process.
- The process is now “in that directory”.
- Internally, the process keeps a variable storing the inode number of the current directory.
- When you “change into a new directory”, you're really just changing the value of that

Writing pwd

Writing pwd

- The *pwd* command prints the path to the current directory:



- Where is the long path stored?
- How does *pwd* know the directory is called CS3560, how does it know its parent is CSUEB, etc?

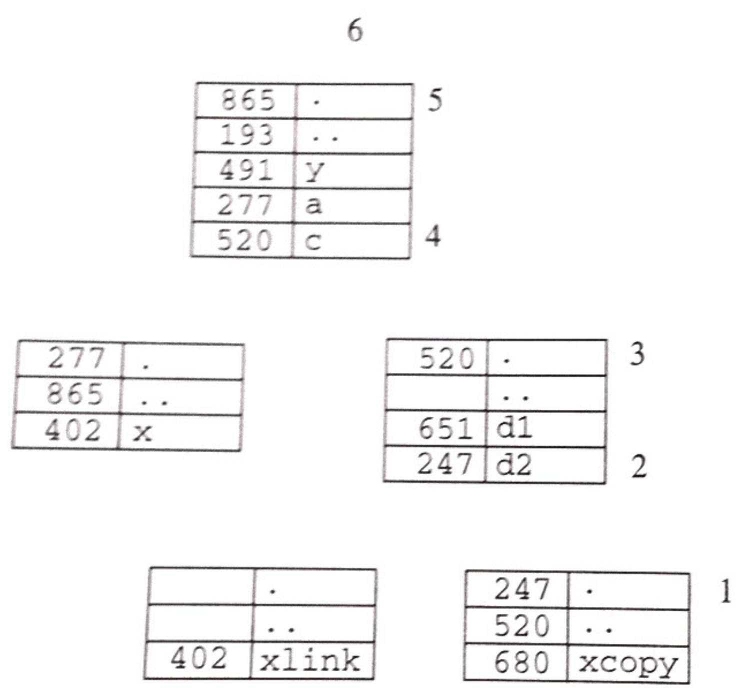
How pwd Works

- The answer: follow the links and read the directories.
- *pwd* climbs up the tree, directory by directory, noting at each step the inode number for dot, then looking through the parent directory for the name assigned to that inode number, until it reaches the top of the tree.

How pwd Works

- Consider:

Computing pwd:



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 ..

FIGURE 4.12

Computing the current path.

How pwd Works

- Let's start in the current directory, the one in the lower right.
- The name of our location is “.” and has inode number 247.
- Now, *chdir* to the parent directory, and look for the entery with inode number 247.
- In the parent, inode 247 is called d2, thus last component in the path is d2.

How pwd Works

- In the parent, its name is “.”, and it has inode number 520.
- *chdir*ing into its parent, we can see inode 520 is listed as c.
- Thuse the last two parts of the path are c/d2.

How pwd Works

- So, here's an algorithm:

1. Note the inode number for “.”, call it *n* (use *stat*)
2. *chdir* .. (use *chdir*)
3. Find the name of the link with inode *n* (use *opendir*, *readdir*, *closedir*)

   Repeat (until you reach the top of the tree)

How pwd Works

- Sounds simple, but two questions:

1. ***How do we know when we reach the top of the tree?***
   In the root directory, dot and dotdot point to the same inode. So our *pwd* version repeats until it gets to a directory where dot and dotdot are equal

How pwd Works

2. ***How do we print the directory names in the correct order?***
   Could write a loop and build up a string of directory names with *strcat* or *sprintf*. Instead, let's use recursion.

- Let's take a look some code, spwd.c.

How pwd Works

- ***Does it work?***

```
hank@netbook: ~/Desktop/CSUEB/CS3560/CH04
hank@netbook:~/Desktop/CSUEB/CS3560/CH04$ pwd
/home/hank/Desktop/CSUEB/CS3560/CH04
hank@netbook:~/Desktop/CSUEB/CS3560/CH04$ ./spwd
/home/hank/Desktop/CSUEB/CS3560/CH04
hank@netbook:~/Desktop/CSUEB/CS3560/CH04$
```

- Perfect-o.

Multiple File Systes: A Tree of Trees

Hmmm.....

- What if a Unix system has two disks or partitions?
- With a few simple abstractions, we organize a single partition into a tree of directories.
- If you have two partions, do you have separate trees?

Hmmm.....Hmmm.....

- Some operating systems assign drive letters or volume names to each disk or partition.
- They then become part of the full path to the file.
- Some systems assign block numbers across all disks to create a virtual single disk.

Hmmmmmmmm.....

- Unix goes a different way.
- Each partition has its own file system tree.
- When there is more than one file system on a computer, these trees are grafted into one larger tree.

Hrrruuuh????

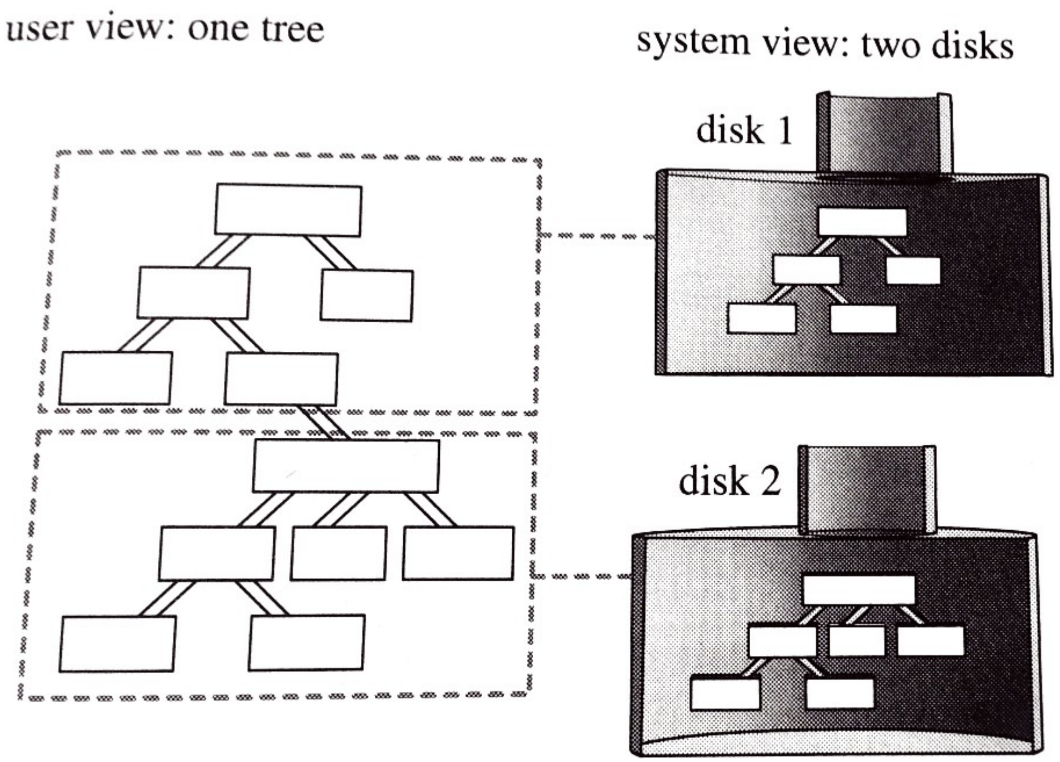


FIGURE 4.13

Tree grafting.

Hrrruuuh???? OH!

- The user sees a seamless tree of directories.
- Really, there are two trees, one on disk 1 and one on disk 2.
- Each tree has a root directory.
- One file system is the *root filesystem*; the top of this tree is the root of the entire tree.

Hrrruuuh???? OH!

- The other tree is attached to some subdirectory of the root file system.
- Internally, the kernel associates a pointer to the other filesystem with a directory on the root file system.

Mount Points

- The phrase *mount a file system* is like *mounting a picture* – that is, 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 attaches is the *mount point* for that second system.

Mount Points

- The *mount* command lists the currently mounted file systems and their mount points:
  $ **mount**
  /dev/hda1 on / type ext (rw)
  /dev/hda6 on /home type ext2 (rw)
  none on /proc type proc (rw)
  none on /dev/pts type devpts (rw,mode=0620)
  $

Mount Points

- First line reports partion 1 on /dev/hda (the first IDE drive) is mounted at the tree root
- This partition is the root file system.
- The second line says the file system on /dev/hda6 is attached to the root file system at the /home directory.

Mount Points

- When a user *chdir*s from / to /home, she crosses from one file system to another.
- When our *pwd* winds its way up the tree, it will stop at /home because it reached the top of its file system.

Mount Points

- Unix allows different types of file systems to be mounted on the root file system.
- A CD-ROM could be mounted.
- A disk containing a Windows or Macintosh file system can be mounted if the kernel contains subroutines that know how to work with their file structures.
- Even file systems from other computers using network connections.

Duplicate Inode Numbers and Cross-Device Links

- Under Unix, every file in the file system has an inode number.
- Two different disks may have files with inode number 402.
- Several directories may contain filenames associated with inode 402.
- How does the kernel figure this out?

Duplicate Inode Numbers and Cross-Device Links

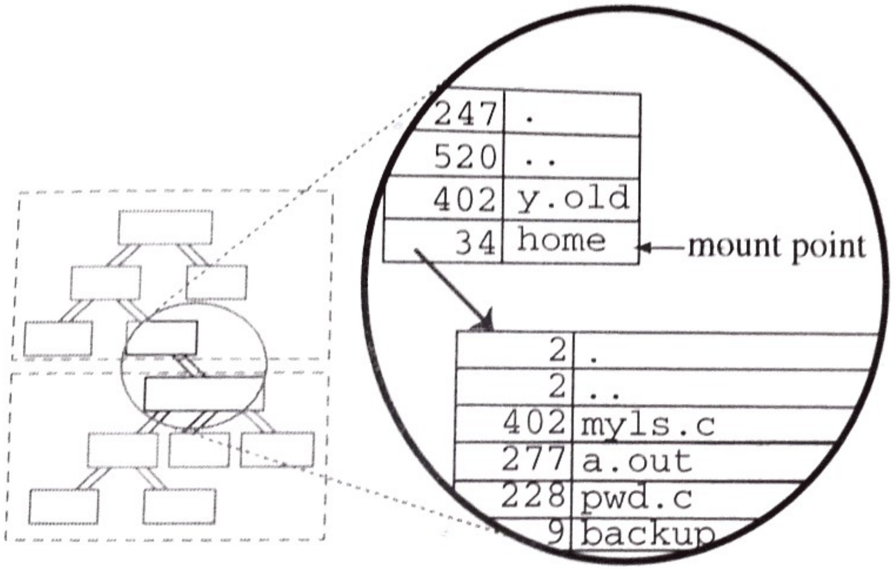


FIGURE 4.14

Inode numbers and file systems.

Duplicate Inode Numbers and Cross-Device Links

- Each directory contains a link to inode 402.
- myls.c and y.old appear to be links to the same inode, but where is that inode?
- Disk 1's file system has an inode 402, and the file system on disk 2 has a different inode 402.
- Don't refer to the same file.
- inode number no longer identifies a file uniquely.
- Looks like link to same file, but they're not

Duplicate Inode Numbers and Cross-Device Links

- ***How can I make links to the same file from different file systems?***
  You can't.
- ***Do the link and rename system calls know about this?***
  Yup. *link* refuses to create cross-device links and rename refuses to transfer an inode number across file systems.

Duplicate Inode Numbers and Cross-Device Links

- Hard llinks are the pointers that connect directories into a tree; hard links are the pointers that link filenames to the files themselves.
- Hard links can't point to inodes in other file systems.
- But, there's another kind of link....

Duplicate Inode Numbers and Cross-Device Links

- The *symbolic link* refers to a file by name, not by inode number.
- Here's a comparison:

```
hank@netbook: ~

hank@netbook:~$ who > whoson
hank@netbook:~$ ln whoson ulist
hank@netbook:~$ ls -li whoson ulist
400613 -rw-rw-r-- 2 hank hank 88 Jul 10 17:27 ulist
400613 -rw-rw-r-- 2 hank hank 88 Jul 10 17:27 whoson
hank@netbook:~$ ln -s whoson users
hank@netbook:~$ ls -li whoson ulist users
400613 -rw-rw-r-- 2 hank hank 88 Jul 10 17:27 ulist
403333 lrwxrwxrwx 1 hank hank  6 Jul 10 17:28 users -> whoson
400613 -rw-rw-r-- 2 hank hank 88 Jul 10 17:27 whoson
hank@netbook:~$
```

Duplicate Inode Numbers and Cross-Device Links

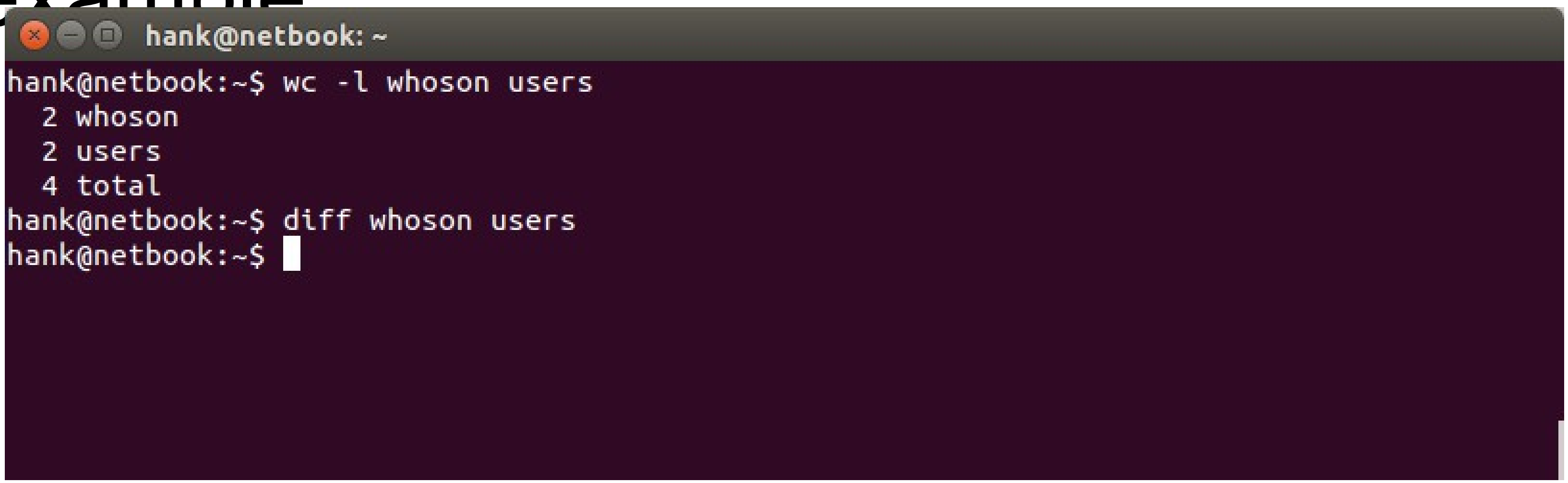
- The files whoson and ulist are links to the same file.
- Both have inode number 400613, and both have same file size, mod time, and number of links.
- The hard link ulist was created with *ln*.

Duplicate Inode Numbers and Cross-Device Links

- *ln -s*, however, makes a symbolic link to the file whoseon and calls that new link users.
- *ls -li* shows users has inode 403333.
- The letter l in the file-type spot says that users is a symbolic link.
- The link count, mod time, and file size differ from the original file.
- This file, users, is not the original file whoseon, but behaves like the original file when programs read or write to it.

Duplicate Inode Numbers and Cross-Device Links

- For example



- *wc* and *diff* read the files, counting lines and comparing content, respectively.
- The kernel uses the name to find the original file.

Duplicate Inode Numbers and Cross-Device Links

- Symbolic links can span file systems because they don't store the inode of the original file.
- Can also point to directories.

Duplicate Inode Numbers and Cross-Device Links

- Some problems with symbolic links though.
- If the file system containing the original file is removed, or the original file gets a new name, or if a different file with that name is installed, the symbolic link will point to nothing, nothing, and something different, respectively.

Duplicate Inode Numbers and Cross-Device Links

- Symbolic links can points to parent directories, creating loops in the directory tree.
- Can turn your file system into spaghetti.
- Kernal knows these are only symbolic links and can check them for lost references and finite loops.

System Calls for Symbolic Links

- The *symlink* system call creates a symbolic link.
- The *readlink* system call obtains the name of the original file.
- *lstat* obtains info about the original file.
- See man pages on *unlink*, *link* to see what they do with symbolic links.

Summary

Main Ideas

- Unix organizes disk storage into file systems.
  --A file system is a collection of files and directories.
  --A directory is a list of names and pointers.
  --Each entry in a directory points to a file or a directory.
  --A directory contains entries that point to tis parent directory and to its subdirectories

Main Ideas

- A Unix file system contains three main parts: a superblock, and inode table, and a data region.
  --File contents are stored in data blocks.
  --File attributes are stored in an inode.
  --The position of the inode in the table is called the inode number of the file.
  --The inode number is the unique file identifier.

Main Ideas

- The same inode number may appear in several directories with various names.
  --Each entry is called a hard link to a file>
  --A symbolic link is a link that refers to a file by name instead of inode number.

Main Ideas

- Several file systems can be connected into one tree.
  --The kernel operation that connects a directory of one file system to the root of another file system is called *mounting*.

Summary

- Unix includes system calls that allow the programmer to create and remove directories, duplicate pointers, move pointers, change the name associated with pointers, and attach and detach other file systems.

Visual Summary

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

Visual Summary

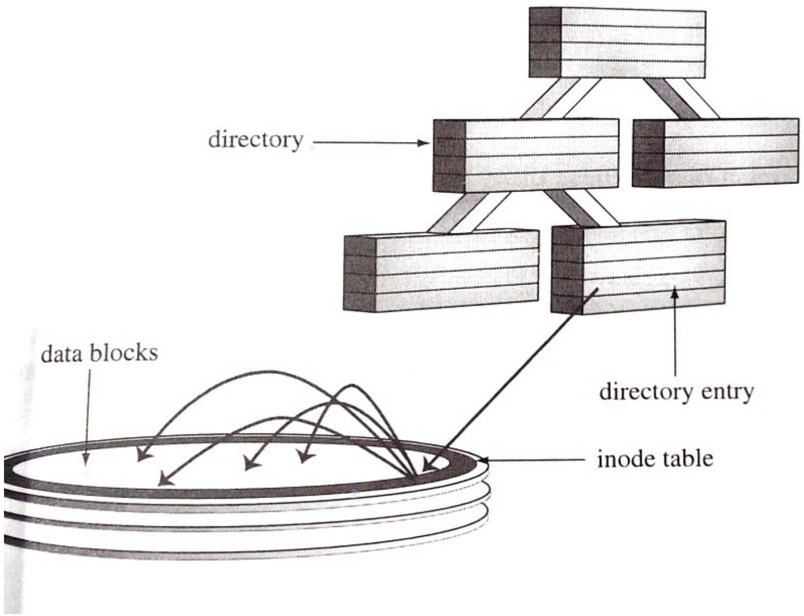


FIGURE 4.15

Inodes, data blocks, directories, pointers.