Introduction

What is UNIX?

UNIX Timeline

What is UNIX?

At Bell Labs, in 1969, Ken Thompson created a tiny operating system that came to be known as UNIX.

During the 1970s UNIX gradually grew and evolved, and spread into the computer science community.

In the 1980s and 1990s UNIX became an immensely popular platform for software R&D and later, enterprise computing.

Some hallmarks of UNIX:

- Pre-emptive multi-tasking of processes
- Full support for multiple simultaneous users
- Utilities work well in combination with others
- APIs that combine simplicity, elegance, and power
- The system is stable and resilient
- The keyboard is alive and well
- Sophisticated users are not encumbered
- Casual users are frustrated

UNIX Timeline

1965 Researchers from Bell Labs and other organizations begin work on Multics, a state-of-the-art interactive, multi-user operating system.

1969 Bell Labs researchers, losing hope for the viability of Multics due to performance issues, withdraw from the project.

One of the researchers, Ken Thompson, finds a little-used PDP-7, and in a month implements a simple operating system comprising a kernel, a command interpreter, an editor, and an assembler.

Other Bell researchers, most notably Dennis Ritchie, are attracted to Thompson's system and contribute to it.

1970 Peter Neumann suggests the name "Unics" for Thompson's operating system, a pun on "Multics". A DEC PDP-11 is acquired for further development of UNIX.

1971 In addition to supporting research, the PDP-11 running UNIX hosts a word processing project: the preparation of patent applications. Work begins on the C programming language.

1973 UNIX is rewritten in C.

1975 Ken Thompson takes a sabbatical and teaches at Berkeley. He gets some students, including Bill Joy, interested in UNIX.

1978 Seventh Edition UNIX (V7), incorporating a goal of portability, is released. Some say that V7 is the classic UNIX.

UNIX Timeline, continued

1979 Building on UNIX/32V, UCB produces a version of UNIX that takes advantage of the DEC VAX-11/780 virtual memory support. It is released as 3BSD (Berkeley Software Distribution).

1981 VAXs running 4.1BSD are the system of choice for computer science departments everywhere.

1982 Sun Microsystems is founded; Bill Joy leads their software development.

1984 A federal court decree allows AT&T to get into the computer business; AT&T releases UNIX System V.

1985 Richard Stallman writes the GNU Manifesto and founds the Free Software Foundation. (GNU's Not UNIX.)

1988 IEEE Std 1003.1-1988 is approved. It came to be known as POSIX.1 (Portable Operating System Interface).

1989 System V R4 (SVR4) is released, merging the System V and BSD development lines.

1991 comp.os.minix: "Hello everybody out there using minix - I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. ..."
—Linus Torvalds, a student at the University of Helsinki

1993 AT&T sells UNIX System Laboratories to Novell; Novell conveys "UNIX" trademark to X/Open, a standards organization.

UNIX Timeline, continued

In the years since 1993 there have been a series of licensing deals, business maneuvers, and lawsuits involving Novell, SCO, The SCO Group, Caldera, and Caldera International.

Today:

> The term "UNIX" can be legally applied to any system that passes a certification process established by The Open Group.
>
> The IEEE/ISO POSIX standards facilitate writing software that is portable between a wide range of UNIX and non-UNIX systems.
>
> Linux keeps getting bigger and better.

The Shell—Part 1

Shell and command-line basics

man(1) and built-in help

I/O Redirection

Pipes

The Shell—basics

Users typically interact with UNIX via a "shell".

A reasonable definition for *shell*:

> *A command-line based environment for execution and control of programs.*

There are many different shells but a number of capabilities are common to all popular shells:

- Command execution
- Redirection of input and output
- Piping
- Wildcard expansion
- Process control
- Command recall and editing
- *Turing-complete*

Shells in use on lectura:

Shell	**Users**
Enhanced C Shell (tcsh)	641
Bourne-Again Shell	119
C Shell	60
Korn Shell	8
Bourne Shell	3

Shell basics, continued

The shell we'll focus on is bash, the Bourne-Again Shell.

bash is our shell of choice because it is:

- A typical shell
- Full-featured
- POSIX-compliant
- Widely available

The things you'll learn about bash fall into three categories:

- Things that work with just about every shell
- Things that work with every POSIX-compliant shell
- Things that are bash-specific but have counterparts in other shells

Command line basics

Typing a command name and pressing the <ENTER> key causes the program associated with the command to be executed. The output, if any, is displayed on the screen. When the program terminates, the shell prompts for another command.

Examples: (typed input is in bold)

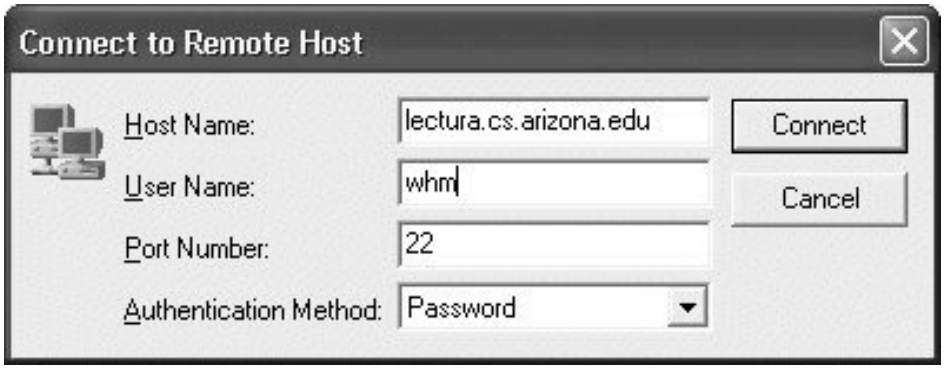
```
$ hostname
lectura.CS.Arizona.EDU
$ whoami
whm
$ true
$ date
Wed Jan 12 22:12:36 MST 2005
$ who
wnj      pts/44   Jan 12 16:42
dmr      pts/73   Jan 11 15:49
drh      pts/50   Jan 10 08:18
ken      pts/50   Jan 11 08:18
rob      pts/47   Jan 11 21:47
fed      pts/44   Jan 12 16:42
$
```

A running instance of a program is called a *process*. A running shell is a process that starts other processes.

Sidebar: Running bash with ssh

The first step to run bash is to login to lectura via a secure shell (ssh) connection.

On departmental Windows machines, you can use Secure Shell Client on the Start menu (or desktop) to establish an ssh connection to lectura. After starting the application, press <ENTER> to display this dialog:



Specify lectura.cs.arizona.edu as the Host Name and your CS Account login id as the User Name. Click Connect and enter your password in the resulting dialog.

You'll then see a window with contents like this:

```
SSH Secure Shell 3.2.9 (Build 283)
[...more...]
Last login:  Wed Jan 12 22:21:18 2005 from amelia.cs...
Sun Microsystems Inc.   SunOS 5.9       Generic May 2002

%
```

Running bash, continued

An alternative ssh client, and the one recommended by the instructor, is PuTTY.

PuTTY can be found here,

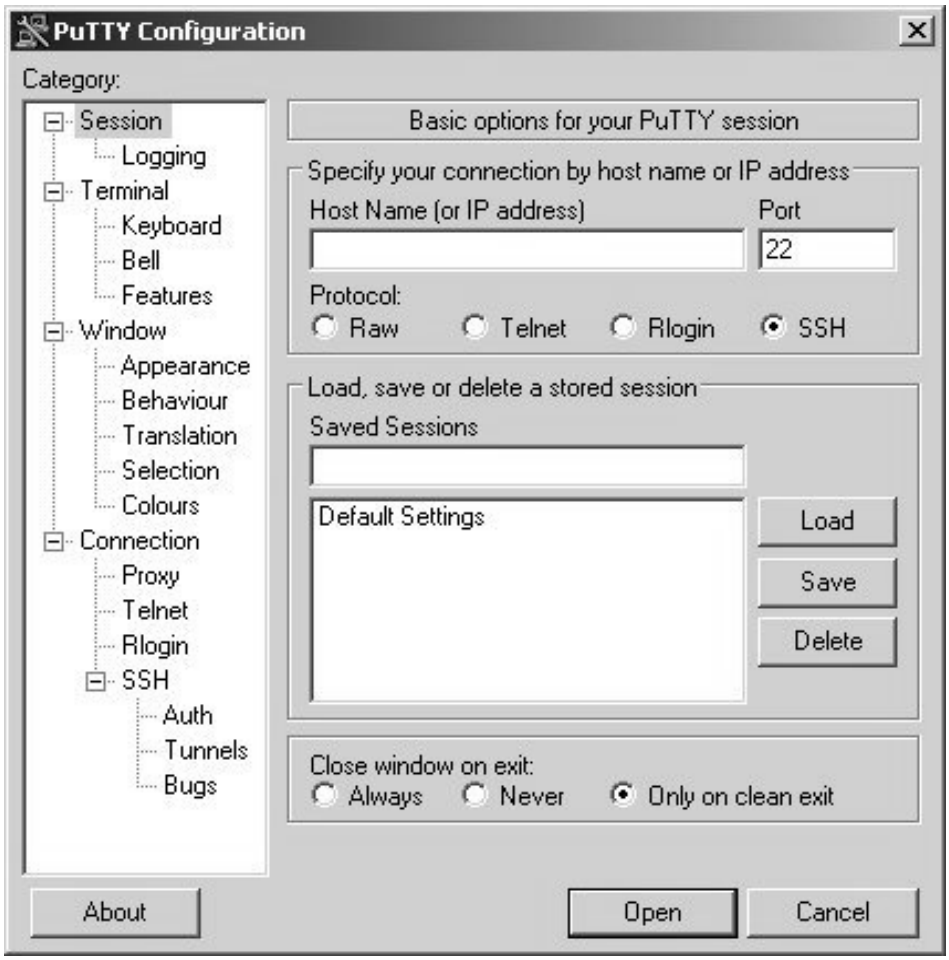
> http://www.chiark.greenend.org.uk/~sgtatham/putty/
> (it's the first hit if you Google for putty)

and on the class website.

There is no installation with PuTTY—there is simply a Windows executable named putty.exe. Put it wherever you want to and run it from there.

Running bash, continued

Running putty.exe produces a configuration screen:



Enter lectura.cs.arizona.edu (just lectura on CS machines) as the host name and click Open. Another window will open that will prompt you for your login name and password. Your shell prompt will then appear.

In PuTTY, a left drag selects text; a right click pastes it.

pscp, found on the PuTTY website, is a ssh-based copy program modeled after the UNIX cp (file copy) command.

Running bash, continued

By default, users on lectura are assigned the enhanced C Shell (tcsh), not bash, as their *login shell*.

To start bash, simply type bash at the tcsh prompt, which typically ends with '%' or '>':

```
% bash
bash-2.04$
```

The default bash prompt is `bash-2.04$`. To avoid clutter on these slides, examples will show the bash prompt as being only a dollar sign.

To terminate bash, type exit. Then type exit again to terminate the login shell (and your ssh session).

```
bash-2.04$ exit
% exit
```

If you wish to avoid the extra step of having to start bash each time you login, change your login shell to be bash. (Do it via http://www.cs.arizona.edu/apply—select CHANGE your UNIX login shell under TYPE of APPLICATION.)

Running bash, continued

ssh clients emulate "dumb terminals"— simple I/O devices that provide little more than a keyboard and a screen that displays a matrix of fixed-size characters.

The stty command displays "terminal" settings:

```
$ stty -a
speed 9600 baud; rows 45; columns 80;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D;
eol = <undef>; eol2 = <undef>; swtch = <undef>; start = ^Q;
stop = ^S; susp = ^Z; dsusp = ^Y; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
[...lots more...]
```

Handy control-characters:

^C (control-C) kills the currently running process.

^U erases the characters typed thus far on the current line.

^W erases the last "word".

^S suspends output; ^Q resumes output.

^O causes output to be discarded until ^O is typed again.

^Z prints "Stopped" and suspends the current process. Execution can be resumed with the fg command; jobs shows active "jobs"; kill kills jobs.

Command line basics, continued

Most commands accept one or more *operands*:

```
$ cal 1 2005
   January 2005
 S  M Tu  W Th  F  S
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
$ banner unix

 #     #  #     #      #     #     #
 #     #  ##    #      #      #   #
 #     #  # #   #      #       ##
 #     #  #  # #      #       ##
 #     #  #    ##     #      #   #
  ####   #     #      #     #     #
```

Command line basics, continued

For many commands the operands are file names.

```
$ cat Hello.java
public class Hello {
    public static void main(String args[]) {
        System.out.println("Hello, world!");
        }
    }
$ javac Hello.java
$ java Hello
Hello, world!
$ rm Hello.class
$ java Hello
Exception in thread "main"
java.lang.NoClassDefFoundError: Hello
```

Note the evidence of the "silence is golden" philosophy, which is common in UNIX programs.

The fgrep command searches for text. Its first argument is a string to search for. The following arguments are the files to search in.

```
$ fgrep Hello Hello.java
public class Hello {
        System.out.println("Hello, world!");
$ fgrep Hello Hello.java Test.java
Hello.java:public class Hello {
Hello.java:  System.out.println("Hello, world!");
$ fgrep Waldo Hello.java Test.java
$
```

Does fgrep embrace "silence is golden"?

Note: The "grep" family will be studied in more depth later.

Command line basics, continued

An operand is one type of command line *argument*. *Options* are another type of command line argument.

Options almost always begin with a '-' (minus sign) . By convention, options appear between the command name and the operands, if any.

Examples:

```
$ date
Thu Jan 13 02:19:20 MST 2005
$ date -u
Thu Jan 13 09:19:22 UTC 2005
$ wc Hello.java
       5      14     127 Hello.java
$ wc -l -w Hello.java
       5      14 Hello.java
```

In some cases, an option has an associated argument:

```
$ javac -verbose -d work Hello.java
[parsing started Hello.java]
[parsing completed 143ms]
[loading
/usr/j2se/jre/lib/rt.jar(java/lang/Object.class)]
...more...
[wrote work/Hello.class]
[total 624ms]
$
```

For most programs the ordering of options is not significant but that is a convention, not a rule.

Command line basics, continued

It is common to allow single character options to be combined into a single multi-character option. For example,

```
wc -l -w Hello.java
```

is equivalent to

```
wc -lw Hello.java
```

Some programs have verbose synonyms for single-character options. Example:

```
wc --words --lines Hello.java
```

As a rule, whitespace is significant in command lines. For example, the commands

```
date-u
```

```
wc -l-w Hello.java
```

are invalid.

There is nothing that prohibits a program from having its own style of argument handling. The dd command, a very old file manipulation utility, uses name/value pairs on the command line:

```
dd if=scores.dat ibs=90 skip=40 count=5 of=x
```

Sidebar: Java and argument handling

When a Java program is run, the shell, the operating system kernel, and the Java run-time system arrange for the command line arguments to appear as an array of strings that is passed to main.

Here is a Java program that displays its arguments:

```
public class args {
   public static void main(String args[]) {
      for (int i = 0; i < args.length; i++)
         System.out.println("|" + args[i] + "|");
      }
   }
```

Interaction:

```
$ java args one two three
|one|
|two|
|three|
$ java args -a -b c +d e=f
|-a|
|-b|
|c|
|+d|
|e=f|
$ java args
$
```

Command line basics, continued

Many non-alphanumeric characters have special meaning to the shell:

```
$ java args :)
bash: syntax error near unexpected token `:)'
```

Characters that have special meaning are often called *metacharacters*. Here are the bash metacharacters:

```
~ ` ! # $ & * ( ) \ | { } [] ; ' " < > ?
```

One way to specify an argument that contains metacharacters or whitespace is to enclose the argument in quotes:

```
$ java args ':)'  "'''"  '"'  ' x '  "y  "  z"'z"
|:)|
|'''|
|"|
| x |
|y  |
|z'z|
```

Note that the enclosing quotes are consumed by the shell. args never sees them.

We'll see later that some metacharacters are still interpreted even when surrounded with double quotes. For the time being, always use single quotes to avoid any surprises.

Problem: Describe a simple way to test whether a character is a shell metacharacter.

Command line basics, continued

An alternative to wrapping with quotes is to use a backslash to "escape" each metacharacter.

If a character is preceded by a backslash, its special meaning, if any, is suppressed.

```
$ java args :\)  \ \'\"\\  x\ y  \x\y\z
|:)|
| '"\|
|x y|
|xyz|
```

Note that it's not an error to escape an ordinary character.

The ASCII NUL (all zero bits) is the only character that can't be passed in an argument.

Command line basics, continued

Multiple commands can be specified on a single command line by separating them with semicolons:

```
$ date -u; cal; wc Hello.java; java args \;
Thu Jan 13 07:42:22 UTC 2005
   January 2005
 S  M Tu  W Th  F  S
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
      5      14     127 Hello.java
|;|
$
```

The shell runs each command in turn, waiting for each to terminate before the next command is started.

Note that there is no indication of where the output of one command ends and the output of the next begins.

Command-line basics—Summary

As a rule, command invocations have this form:

command-name option1 ... optionN operand1 .. operandN

Options and operands are often collectively referred to as arguments.

Options typically start with a '-' and are often single letters; single letter options can often be combined.

Options sometimes have arguments themselves.

The ordering of options is usually not important.

Many programs allow options to follow operands.

As a rule, whitespace in options and operands is significant.

Interpretation of metacharacters can be suppressed by enclosing the argument in quotes or preceding each metacharacter with a backslash.

All-in-all, there are somewhat firm conventions but no hard rules about options and operands.

The man(1) command

The "man" command displays documentation for commands (and more). Here is a slightly abridged example—the "man page" for cal:

```
$ man cal
NAME
     cal - display a calendar

SYNOPSIS
     cal [ [month] year]

DESCRIPTION
     The cal utility writes a Gregorian calendar to standard out-
     put.  If  the year operand is specified, a calendar for that
     year is written. If no operands are  specified,  a  calendar
     for the current month is written.

OPERANDS
     The following operands are supported:

     month Specify the month to be displayed,  represented  as  a
           decimal integer from 1 (January) to 12 (December). The
           default is the current month.

     year  Specify the year for which the calendar is  displayed,
           represented  as  a decimal integer from 1 to 9999. The
           default is the current year.

SEE ALSO
     calendar(1), attributes(5), environ(5)

NOTES
     An unusual calendar is printed for September 1752.

SunOS 5.9         Last change: 1 Feb 1995
```

Note that the `(1)` in `man(1)` indicates that the page comes from section one of the manual, which contains user commands.

The man(1) command, continued

A very handy man option is -k, which specifies a keyword to search for in (all) the man page "NAME" entries.

Example: ("What was that calendar printing command??")

```
$ calendar 8 2004
/bin/calendar: illegal option -- 8 2004
usage: calendar [ - ]
$
$ man -k calendar
cal         cal (1)         - display a calendar
calendar    calendar (1)    - reminder service
difftime    difftime (3c)   - computes the difference
                              between two times
mktime      mktime (3c)     - converts a tm structure
                              to a calendar time
```

Some man page names appear in more than one section of the manual. For example, printf appears in sections 1, 3c, and 3ucb. The -s flag selects the entry in the specified section.

```
man -s 3ucb printf

man -s 1 printf
```

Most manual sections have an intro page that provides an overview of the section.

```
man -s 4 intro
```

Built-in help for commands

Many commands have a `--help` option:

```
$ wc --help
Usage: wc [OPTION]... [FILE]...
Print line, word, and byte counts for each FILE, and a
total line if more than one FILE is specified.  With no
FILE, or when FILE is -, read standard input.

  -c, --bytes, --chars   print the byte counts
  -l, --lines            print the newline counts
  -L, --max-line-length  print the length of the longest
                         line
  -w, --words            print the word counts
      --help             display this help and exit
      --version          output version information and
                         exit

Report bugs to <bug-textutils@gnu.org>.
```

Some commands don't support `--help`, but...

```
$ cal --help
cal: illegal option -- -
usage: cal [ [month] year ]
$
$ cc --help
cc: Warning: option -- passed to ld
usage: cc [ options] files.  Use 'cc -flags' for
details
```

Note: There are separate "search paths" for programs and manual pages. If the paths get out of sync the page displayed by man might not correspond to the version of the program being run. (More on this later.)

I/O Redirection

There are several possible destinations for the output of a program: the screen, a file, another program, a hardware device, a process on another machine, etc. Similarly, input may come from a variety of sources.

UNIX has a notion of a *standard input* stream and a *standard output* stream. It is common for programs to read from standard input and/or write to standard output.

In Java, System.in is associated with standard input; System.out is associated with standard output.

By default, when the shell starts a program, standard input is associated with the keyboard and standard output is associated with the screen.

I/O Redirection, continued

Here is a Java program that reads lines from standard input and writes the line count to standard output:

```
import java.io.*;

public class lc {
   public static void main(String args[]) throws IOException {
      BufferedReader in = new BufferedReader(
         new InputStreamReader(System.in));

      String line;  int count = 0;
      while ((line = in.readLine()) != null)
         count++;

      System.out.println(count);
      }
   }
```

Interaction with lc.java:

```
$ java lc
just
a
test
^D  (control-D)
3
$
```

Due to the default association of standard input with the keyboard and standard output with the screen, java lc reads lines from the keyboard and displays the line count on the screen.

I/O Redirection, continued

It is possible to *redirect* standard input, so that instead of reading characters from the keyboard, the data comes from a file.

Input redirection is indicated by adding '**<** *file*' to the command line:

```
$ java lc < lc.java
15
$
```

Output redirection is similar:

```
$ java lc > count
one
two
three
^D
$ cat count
3
$
```

If the target file does not exist, it is created. If it exists, it is overwritten.

Both input and output can be redirected:

```
$ java lc <lc.java >out
$ java lc < out
1
$
```

Whitespace before and after **<** and **>** is optional.

I/O Redirection, continued

It is important to understand that the shell completely consumes the redirection operators and the file names that follow the operators.
Example:

```
$ java args a b c < lc.java > out
$ cat out
|a|
|b|
|c|
$
```

To the program, there is no evidence whatsoever of the redirection of standard input from lc.java and standard output to out.

Redirections can appear at any place on the command line:

```
$ java > out args a < lc.java b c
$ cat out
|a|
|b|
|c|
$
```

Explain this error:

```
$ java > args out a < lc.java b c
Exception in thread "main"
java.lang.NoClassDefFoundError: out
```

What arguments did the java command see?

I/O Redirection, continued

Many programs will accept input from either standard input or files named on the command line:

```
$ wc Hello.java
       5      14     127 Hello.java
$ wc < args.java
       6      26     180
```

Consider this:

```
$ wc Hello.java < args.java
       5      14     127 Hello.java
```

Why isn't args.java processed, too?

Problem: Try writing a Java program that behaves as follows:

$ java x a b c	Prints a, b, c
$ java x < y	Prints the contents of the file y
$ java x a b c < y	Prints a, b, c, and then contents of file y
$ java x	Prints lines read on standard input

I/O Redirection, continued

A variant of '**>**' is '**>>**'. It appends standard output to the named file, rather than truncating it. Example:

```
$ cal 1 2005 > out
$ echo ..................... >> out
$ echo >> out
$ cal 2 2005 >> out
$ cat out
   January 2005
 S  M Tu  W Th  F  S
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
.....................

   February 2005
 S  M Tu  W Th  F  S
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28
```

Note that echo simply prints its arguments on standard output.

Could the above example start with **`cal 1 2005 >> out`** instead?

I/O Redirection, continued

For reference:

```
$ cal 1 2005 > out
$ echo ..... >> out
$ echo >> out
$ cal 2 2005 >> out
```

In this case, a similar effect can be achieved like this:

```
$ (cal 1 2005; echo; echo .....; cal 2 2005) > out
```

The parentheses serve to group the commands into a *subshell* that runs each command in turn. The output of the subshell is directed to the file out.

Speculate: What does the following command do?

```
$ cal 1 2005; echo; echo .....; cal 2 2005 > out
```

I/O Redirection, continued

There is a third I/O stream: *standard error*.

By convention, programs send "normal" output to standard output, and "exceptional" output to standard error.

```
$ cal 2005 1 >out
cal: bad month
usage: cal [ [month] year ]
$ cat out
$
```

Standard output and error output can be combined with the '>&' redirection operator:

```
$ cal 2005 1 >& out
$ cat out
cal: bad month
usage: cal [ [month] year ]
```

It is possible to separately redirect standard error with '2>':

```
$ cal 2005 1 >stdout 2>stderr
$ cat stdout
$ cat stderr
cal: bad month
usage: cal [ [month] year ]
$
```

In Java, System.err is associated with standard error.

I/O Redirection, continued

One great benefit of I/O redirection is that a program doesn't need to include any file-handling code. A program can be written in terms of standard input and standard output; opening files (and handling potential failures) is handled by the shell.

Consider an alternative interface for the line counter,

```
java lc -input x.txt -output count
```

and the additional code that would be required. (Problem: Write it!)

Contrast: Once upon a time, users of DEC's VMS operating system did output redirection like this,

```
$ assign/user sys$output out
$ run program
```

which is downright clumsy compared to this:

```
$ program > out
```

Pipes

A *pipe* is an IPC (interprocess communication) mechanism supported by the UNIX kernel.

A pipe connects two processes such that data written into the pipe by the sending process can be read by the receiving process.

The shell supports piping between processes. Example:

```
$ who
hehf     pts/5    Jan 10 16:31
srinivas pts/30   Jan 11 09:20
kobus    pts/10   Jan 12 13:41
christem pts/17   Jan 12 23:16
laurynas pts/18   Jan 12 13:01
whm      pts/9    Jan 13 00:07
...lots more...
$
$ who | wc -l
     98
```

The command **`who | wc -l`** connects the standard output of who with the standard input of wc. who and wc run simultaneously.

Data always flows from left to right in a pipeline.

Problem: How could we find out if pete is logged in?

Pipes, continued

A key element of the UNIX philosophy is to use *pipelines* to combine two or more programs to perform a useful task.

Here is a pipeline that displays users in order by login name:

```
$ who | sort
dmr      pts/73   Aug 15 15:49
drh      pts/50   Aug 13 08:18
drh      pts/73   Aug 14 11:01
ken      pts/50   Aug 13 08:18
rob      pts/47   Aug 12 12:47
rob      pts/47   Aug 16 21:47
wnj      pts/44   Aug 16 16:42
```

Just login names:

```
$ who | sort | cut -f 1 -d " "
dmr
drh
drh
ken
rob
rob
wnj
```

Unique users:

```
$ who | sort | cut -f 1 -d " " | uniq
dmr
drh
ken
rob
wnj
```

Pipes, continued

Programs such as cut, sort, and uniq, which read standard input, transform it, and write the result to standard output are often called *filters*.

Problem: Write pipelines to...

> Count the number of current login sessions for whm. (Note that fgrep reads standard input if no files are specified.)
>
> Count the number of words in /usr/dict/words that contain all the vowels.
>
> Print the login name of the user who has the greatest number of login sessions. (Hint: uniq -c and head)
>
> Confirm that tcsh is the most commonly used shell on lectura.
>
> Print the login names of non-idle users who are using pine. (Hint: use w and cut -c.)

Problem: Devise three pipeline problems.

Pipes, continued

Problem: Draw diagrams for these command lines...

```
date > out

> x java lc < lc.java

cat < lc.java | wc | cat > x

wc >x lc.java | wc
```

Problem: Prove that all processes in a pipeline are running at the same time. Could pipelines be provided on an operating system that can run only one process at a time?

Files and File Management

Filenames

Directories and paths

The ls command

The mkdir command

Copying files with cp

Deleting files with rm

Links

Moving/renaming files with mv

Tilde (~) substitution

Wildcards

Brace expansion

Filename completion

Filenames

All ASCII characters except NUL (all bits zero) and / (slash) can appear in a UNIX filename.

The maximum length of a filename is platform-dependent. On lectura, which runs Solaris 9, the limit is 255 characters.

Here are some valid filenames:

```
Hello.java
a.out
core
.bashrc
a.b.c.d.
!@#$%^&*()_-+=
A collection of assorted notes about UNIX
:)
\_\
            (three blanks and two tabs)
...
```

Filenames are case-sensitive. For example, hello.java and Hello.java name two different files.

If a filename contains shell metacharacters or whitespace characters, the characters must be escaped when the file is specified on the command line:

```
$ wc -c '[1]' Version\ 2 'This|That'
     359 [1]
     688 Version 2
     417 This|That
    1464 total
```

Directories and paths

With UNIX, like most modern operating systems, files are organized using *directories*. Directories are collections of files and directories.

Collectively, the files and directories in a directory are called *directory entries*.

Directory names follow the same rules as file names.

The location of a file or directory is specified with a *path*. A path consists of a series of zero or more directory names and ends with the name of a file or directory. Slashes separate the path components.

Here are some paths:

```
/home/whm/352/Hello.java
/usr/lib
/etc/passwd
/
/cs/www/classes/cs352/spring05
/net/sin/vol/vol0/home/whm
/var/tmp//xdviAAALZaG4A
```

It may not be obvious whether a path specifies a file or directory.

If a path starts with a slash (/), it is an *absolute path*.

The file or directory specified by an absolute path is located by starting at "the root" (/) and traversing each directory in turn.

Every file and directory can be reached from the root.

Traversing a path may carry one across disk and machine boundaries.

Directories and paths, continued

Here are some places of interest on lectura:

/usr/bin
: Standard programs such as cat, man, and wc

/usr/local
: A collection of software that is not supplied by Sun. Examples: /usr/local/bin/mysql-4.0.20 is a directory tree for MySQL; /usr/local/bin contains a collection of programs, including some GNU replacements for programs in /usr/bin.

/usr/lib
: Libraries of code and data. Examples: /usr/lib/libm.a is collection of math routines for C; /usr/lib/spell is a directory of data files used by the spell(1) command.

/tmp
: Temporary files. Anybody can create a file or directory in /tmp.

/etc
: A diverse collection of files that typically related to system administration. Examples: /etc/passwd is a list of all users; /etc/mail/aliases is a list of mail aliases.

/cs/www
: The root of www.cs.arizona.edu.

Directories and paths, continued

Every process has a *current working directory*. The current directory of the shell can be printed with the pwd command:

```
$ pwd
/home/whm
```

The cd command is used to change the current directory of the shell:

```
$ cd /home/cs352/spring05
$ pwd
/home/cs352/spring05
$ cd /etc
$ pwd
/etc
```

When a user logs in, the current directory of the shell is set to the user's *home directory*. With no arguments, cd changes to the home directory:

```
$ pwd
/etc
$ cd
$ pwd
/home/whm
```

On departmental Windows machines, a user's UNIX home directory is mapped to `H:`.

Directories and paths, continued

A path that does not start with a slash is called a *relative path*.

Relative paths are interpreted relative to the current directory of the process.

Processes started by the shell "inherit" the shell's current directory.

Here are some relative paths:

```
notes
352/Hello.java
a1/tests/v1/old
```

If the working directory is /home/whm then

```
wc notes 352/Hello.java
```

operates on the same files as

```
wc /home/whm/notes /home/whm/352/Hello.java
```

Directories and paths, continued

There are two directory entries that are special: `.` (read as "dot") and `..` ("dot dot"). Every directory contains `.` and `..`

The name `.` specifies the current directory. We'll see practical uses for "dot" later but for the moment notice that

```
notes
./notes
./././notes
```

all refer to the file notes in the current directory.

The name `..` refers to the parent of the current directory.

We might use `..` in a relative path to reference files or directories in parent directories or subdirectories of parents:

```
diff Timer.java ../Timer.java
cd ../352/a1
cal > ../../cal.out
```

A path need not be minimal. Example:

```
/home/cs352/../../../home/././//./whm/352/..
```

Problem: Describe an algorithm to interpret a relative path given a current directory.

How many different paths can be used to specify the file /etc/passwd?

The ls command

The ls ("LS") command lists the contents of a directory. By default, it operates on the current directory.

```
$ ls
a  out  test  z.dat
$ ls -l
total 8
-rw-rw-r-- 1 whm  dept     0 Jan 17 21:47 a
-rw-rw-r-- 1 whm  dept    29 Jan 17 21:47 out
drwxrwxr-x 2 whm  dept  4096 Jan 17 22:20 test
-rw-rw-r-- 1 whm  dept   148 Jan 17 22:19 z.dat
```

The "columns" are type and permissions, link count, owner, group, size in bytes, last modification time, and file name. (More on the first four later.)

The leading "d" indicates that test is a directory. "total 8" indicates that a total of eight 1K blocks is required by the entries.

ls has many options. The -t option causes the list to be ordered by modification time, most recent first:

```
$ ls -lt
total 8
drwxrwxr-x 2 whm  dept  4096 Jan 17 22:20 test
-rw-rw-r-- 1 whm  dept   148 Jan 17 22:19 z.dat
-rw-rw-r-- 1 whm  dept     0 Jan 17 21:47 a
-rw-rw-r-- 1 whm  dept    29 Jan 17 21:47 out
```

The ls command, continued

If files or directories are specified as operands for ls, it operates on them instead of the current directory.

```
$ ls -l a out
-rw-rw-r-- 1 whm  dept     0 Jan 17 21:47 a
-rw-rw-r-- 1 whm  dept    29 Jan 17 21:47 out
$ ls test
1  2  C1.java  C2.java
$ ls -l test
total 152
-rw-rw-r-- 1 whm  dept   159 Jan 17 22:20 1
-rw-rw-r-- 1 whm  dept  2430 Jan 17 22:20 2
-r--r--r-- 1 whm  dept 99125 Jan 17 22:20 C1.java
-r--r--r-- 1 whm  dept 39650 Jan 17 22:20 C2.java
```

Files and directories whose names start with a period are called *hidden files*. ls does not display hidden files unless the -a option is specified.

```
$ ls -la
total 16
drwxrwxr-x 3 whm  dept  4096 Jan 17 22:32 .
drwxrwxr-x 3 whm  dept  4096 Jan 17 21:47 ..
-rw-rw-r-- 1 whm  dept    16 Jan 17 22:43 .opts
-rw-rw-r-- 1 whm  dept     0 Jan 17 21:47 a
-rw-rw-r-- 1 whm  dept    29 Jan 17 21:47 out
drwxrwxr-x 2 whm  dept  4096 Jan 17 22:20 test
-rw-rw-r-- 1 whm  dept   148 Jan 17 22:19 z.dat
```

Speculate: What would ls -at display?

The ls command, continued

The -R option causes ls to recursively process subdirectories.

```
$ ls -lR
.:
total 8
-rw-rw-r-- 1 whm  dept     0 Jan 17 21:47 a
-rw-rw-r-- 1 whm  dept    29 Jan 17 21:47 out
drwxrwxr-x 2 whm  dept  4096 Jan 17 22:20 test
-rw-rw-r-- 1 whm  dept   148 Jan 17 22:19 z.dat

test:
total 152
-rw-rw-r-- 1 whm  dept   159 Jan 17 22:20 1
-rw-rw-r-- 1 whm  dept  2430 Jan 17 22:20 2
-r--r--r-- 1 whm  dept 99125 Jan 17 22:20 C1.java
-r--r--r-- 1 whm  dept 39650 Jan 17 22:20 C2.java
```

Learn more: Read the man page for ls and experiment with various options. -F, -C, -1 (one), -r, -s, -S, and -u are among the more commonly used options of ls.

Problem: Explain the following behavior:

```
$ ls
a  out  test  z.dat
$ ls | wc
      4       4      17
```

The mkdir command

The mkdir command creates one or more directories.

```
$ ls
a  out  test  z.dat
$ mkdir test2
$ mkdir test2/a test2/b test2/c
$ ls -l test2
total 12
drwxrwxr-x 2 whm  dept  4096 Jan 17 23:02 a
drwxrwxr-x 2 whm  dept  4096 Jan 17 23:02 b
drwxrwxr-x 2 whm  dept  4096 Jan 17 23:02 c
```

mkdir will not create intermediate directories in a path unless the -p option is specified.

```
$ mkdir test3/x/y
mkdir: cannot make directory `test3/x/y': No such
file or directory
$ mkdir -p test3/x/y
$ touch test3/a test3/x/a
$ ls -lR test3
test3:
-rw-rw-r-- 1 whm  dept     0 Jan 17 23:13 a
drwxrwxr-x 3 whm  dept  4096 Jan 17 23:13 x

test3/x:
-rw-rw-r-- 1 whm  dept     0 Jan 17 23:13 a
drwxrwxr-x 2 whm  dept  4096 Jan 17 23:13 y

test3/x/y:
```

What does the touch command apparently do?

Copying files with cp

Three common file manipulations are copying, deleting, and moving/renaming.

The cp command copies one file to another, or, alternatively, copies one or more files to a directory:

```
cp file1 file2
cp file1 dir1
cp file1 file2 ... fileN dir1
```

Examples:

```
$ echo "some data" > x
$ cp x y
$ ls -l
-rw-r--r--   1 whm  dept       10 Aug 20 00:13 x
-rw-r--r--   1 whm  dept       10 Aug 20 00:13 y
$ cp y z
$ ls -l
-rw-r--r--   1 whm  dept       10 Aug 20 00:13 x
-rw-r--r--   1 whm  dept       10 Aug 20 00:13 y
-rw-r--r--   1 whm  dept       10 Aug 20 00:15 z
$ mkdir d
$ cp x y z d
$ cp x d/a
$ ls d
a  x  y  z
```

What does the following command do?

```
$ cp /etc/passwd /usr/lib/aliases ../x .
```

cp, continued

The -i option provides some safety—it causes cp to prompt if a file is going to be overwritten:

```
$ touch x y
$ ls -l
-rw-r--r--   1 whm  dept        0 Aug 20 00:26 x
-rw-r--r--   1 whm  dept        0 Aug 20 00:26 y
$ cp x y   (No warning -- y was overwritten)
$ cp -i x y
cp: overwrite `y'? y
```

cp's -r option can be used to recursively copy a directory. The -p option preserves modification times (and more).

```
$ ls -l d
-rw-r--r--   1 whm  dept       43 Aug 14 23:18 a
-rw-r--r--   1 whm  dept      293 Aug  4 00:25 b
drwxr-xr-x   2 whm  dept     4096 Aug 20 00:33 c
$ cp -r d d2
$ ls -l d2
-rw-r--r--   1 whm  dept       43 Aug 20 00:42 a
-rw-r--r--   1 whm  dept      293 Aug 20 00:42 b
drwxr-xr-x   2 whm  dept     4096 Aug 20 00:42 c
$ cp -rp d d3
$ ls -l d3
-rw-r--r--   1 whm  dept       43 Aug 14 23:18 a
-rw-r--r--   1 whm  dept      293 Aug  4 00:25 b
drwxr-xr-x   2 whm  dept     4096 Aug 20 00:33 c
```

Note that the contents of d/c are not shown but the contents are copied to d2/c and d3/c.

Deleting files with rm

The rm (remove) command is used to permanently delete one or more files:

```
$ rm tmp.out Hello.java.old a b c
```

To remove a directory, use rmdir.

```
$ rmdir x
```

A directory must be empty before it can be removed with rmdir, but rm's -r option can be used to remove a directory and all its contents.

```
$ mkdir -p y/z
$ rmdir y
rmdir: y: File exists
$ rm -rf y
$
```

The -i option of rm causes a prompt before a file is removed. The -f option forces removal of read-only files.

Recovering files via .snapshot

On lectura, a directory named /home/*user*/.snapshot maintains various virtual snapshots of the user's directory tree:

```
$ ls -F /home/whm/.snapshot
hourly.0/  hourly.3/  hourly.6/   nightly.1/  nightly.4/  weekly.1/
hourly.1/  hourly.4/  hourly.7/   nightly.2/  nightly.5/
hourly.2/  hourly.5/  nightly.0/  nightly.3/  weekly.0/
```

If a file is inadvertently deleted, you may be able to find a copy of it in your .snapshot directory.

```
$ pwd
/home/whm/352
$ ls  -l lc.java
-rw-r--r-- 1 whm  dept   359 Jan 13 00:25 lc.java
$ rm lc.java  (Oops!)
```

Start with hourly.0 and look for the missing file. If found, just copy it back into place.

```
$ ls -l /home/whm/.snapshot/hourly.0/352/lc.java
-rw-r--r-- 1 whm  dept   359 Jan 13 00:25
/home/whm/.snapshot/hourly.0/352/lc.java
$ cp /home/whm/.snapshot/hourly.0/352/lc.java .
```

The virtual snapshots are created periodically. A snapshot might not contain a recently created file or recent modifications to an older file.

Directory trees can be restored with cp -rp.

Note that some file CS servers maintain more snapshots than others.

Links

Internally, a file is represented with an *i-node* (index node), a data structure that contains information such as the starting disk address of the file's data, the owner of the file, modification time, etc.

A directory is simply a file with distinguished status that contains a mapping between i-node numbers and names.

Here is a Java model of a directory:

```
class Directory {
   static class directory_entry {
      int inode_number;
      String name;
      }
   directory_entry entries[];
   int active_entries = 0;
   }
```

The -i option of ls shows i-node numbers:

```
$ date >a
$ mkdir b
$ ls -ia
200793473 .  7372598 ..  48319878 a  8379685 b
```

Links, continued

For reference:

```
$ date >a
$ mkdir b
$ ls -iaF
200793473 ./  7372598 ../  48319878 a  8379685 b/
```

A file can have more than one name in a directory. A file can be in more than one directory.

The ln (link) command can be used to create another directory entry for an existing file:

```
$ ln a c
$ ls -iaF
200793473 ./  7372598 ../  48319878 a  8379685 b/
 48319878 c
$ cat a
Fri Aug 20 17:55:14 MST 2004
$ cat c
Fri Aug 20 17:55:14 MST 2004
$ echo "new contents" > c
$ cat a
new contents
$ ln a b/x
$ ls -iRF
.:
48319878 a  8379685 b/  48319878 c

b:
48319878 x
```

Links, continued

The link created with a command like 'ln f1 f2' is called a *hard link*. Hard links pose problems in some situations. For example, files can not be linked across devices, and directories cannot be linked:

```
$ ln /home/whm/x /tmp/x
ln: cannot create hard link `/tmp/x' to
`/home/whm/x': Cross-device link
$ mkdir d
$ ln d d2
ln: d: hard link not allowed for directory
```

Another problem is that of a *broken link*.

Frequently, a *symbolic link* is a better choice than a hard link.

A symbolic link, or "symlink", is indicated with the -s option of ln:

```
$ date >x
$ ln -s x x2
$ ls -l
total 16
-rw-r--r--   1 whm  dept  29 Aug 20 18:35 x
lrwxrwxrwx   1 whm  dept   1 Aug 20 18:35 x2 -> x
$ cat x2
Fri Aug 20 18:35:12 MST 2004
```

A symbolic link is a file that's treated by the kernel in a distinguished way. Note the link and the target file have different i-node numbers:

```
$ ls -li
 87281 -rw-r--r-- 1 whm dept 29 Aug 20 18:54 x
411545 lrwxrwxrwx 1 whm dept  1 Aug 20 18:53 x2 -> x
```

Links, continued

A symbolic link can be made to a directory:

```
$ ln -s /home/cs352/fall04/assign1/refs a1refs
$ ls -l a1refs
lrwxrwxrwx   1 whm  dept   31 Aug 20 18:42 a1refs
  -> /home/cs352/fall04/assign1/refs
$ cd a1refs
$ pwd
/home/cs352/fall04/assign1/refs
```

A symbolic link can be created (and exist) regardless of whether the target exists.

```
$ ls
$ ln -s a b
$ ls -l
total 8
lrwxrwxrwx   1 whm  dept    1 Aug 20 18:53 b -> a
$ cat b
cat: b: No such file or directory
$ date >a
$ cat b
Fri Aug 20 18:54:07 MST 2004
```

The -L option of ls "follows" symlinks:

```
$ ls -l b
lrwxrwxrwx   1 whm  dept    1 Aug 20 18:53 b -> a
$ ls -lL b
-rw-r--r--   1 whm  dept   29 Aug 20 18:54 b
```

Links, continued

Symlinks can reference other symlinks:

```
$ date >a
$ ln -s a b
$ ln -s b c
$ ls -l
-rw-r--r--   1 whm  dept  29 Aug 20 19:12 a
lrwxrwxrwx   1 whm  dept   1 Aug 20 19:12 b -> a
lrwxrwxrwx   1 whm  dept   1 Aug 20 19:12 c -> b
$ cat c
Fri Aug 20 19:12:29 MST 2004
```

Some programs, such as ls, treat symlinks in special ways. Other programs, such as cat, simply follow the link(s) and open the referenced file, if it exists.

Two more examples: rm removes a symlink, not the referenced file, and does not follow symlinks to directories. cp copies the target file.

```
$ date >x
$ ln -s x x2
$ ls -l
-rw-r--r--   1 whm  dept  29 Aug 21 00:30 x
lrwxrwxrwx   1 whm  dept   1 Aug 21 00:30 x2 -> x
$ cp x2 y
$ rm x2
$ ls -l
-rw-r--r--   1 whm  dept  29 Aug 21 00:30 x
-rw-r--r--   1 whm  dept  29 Aug 21 00:31 y
```

(Bottom line: RTM for the details.)

Moving/renaming files with mv

The mv command can be used to rename a file or directory. The general form is 'mv *old-name new-name*'. Examples:

```
$ mv Node.java Element.java
$ mv tests tests.old
```

The change is accomplished by deleting the directory entry for the old name and making an entry for the new name, with the same i-node number:

```
$ ls -i Element.java
6912838 Element.java
$ mv Element.java Item.java
$ ls -i Item.java
6912838 Item.java
```

mv can also be used to move one or more files (and/or directories) between directories:

```
$ mkdir docs
$ mv Tree.html Scan.html docs
$ mv ../../x /home/whm/notes .
```

mv can handle cross-device moves. In that case the result is achieved via a copy and then a delete. Example:

```
$ ls -i Item.java
6912838 Item.java
$ mv Item.java /tmp/whm
$ ls -i /tmp/whm/Item.java
4677186 /tmp/whm/Item.java
```

The -i option causes mv to prompt if a target file exists.

Tilde (~) substitution

It is often the case that a user wishes to refer to files and/or directories in their home directory.

One option is to specify a full path, such as /home/whm/notes, but that can be cumbersome if the path to a home directory is long, such as /r/serv1/vol2/users/w/whm.

The shell treats the tilde character (~) as a shorthand for the user's home directory, replacing occurrences of ~ with the home directory path:

```
$ java args ~
|/home/whm|
$ echo ~/x ~/352/args.java
/home/whm/x /home/whm/352/args.java
$ cp ~/x.java .
$ cd ~/352
```

Also, the shell expands ~*user* to the home directory of the specified user:

```
$ echo ~whm
/home/whm
```

Important: Note that echo (and args) allow us to see the end result of the command-line transformations made by the shell:

```
$ echo cp ~/x.java .
cp /home/whm/x.java .
```

To "see" what a command is "seeing", just prepend echo!

Tilde (~) substitution, continued

Note that tilde substitution is done by the shell, not the operating system. A program can be coded to support shell-like tilde expansion—some programs do, but many programs don't.

The java.io.File class does not handle tilde:

```
$ ls -l ~/x
-rw-r--r--   1 whm      dept            0 Aug 25 02:34 /home/whm/x
$ cat tilde.java
import java.io.*;
public class tilde {
   public static void main(String args[]) {
      System.out.println(new File("/home/whm/x").exists());
      System.out.println(new File("~/x").exists());
      }
   }
$ javac tilde.java
$ java tilde
true
false
```

Wildcards

Wildcards allow the user to specify files and directories using textual patterns.

The simplest wildcard metacharacter is ?, a question mark. A question mark matches any one character. Examples:

- ? Matches one-character names. Examples: a, B, :, -
- x? Matches names that are two characters long and begin with an 'x'. Examples: xx, x+, x.

What would be matched by the following?

???

a??b

???.?

Wildcards, continued

If a command line argument contains one or more wildcard specifiers the shell replaces that argument with a list of file names in the current directory that match the specified pattern.

Examples:

```
$ ls
62  X  a  a.b  ab  axe  b oxo

$ echo ?
X a b

$ java args ?? a??
|62|
|ab|
|a.b|
|axe|

$ ls -l ?b ??b ??
-rw-r--r--   1 whm      0 Sep  8 00:12 62
-rw-r--r--   1 whm      0 Sep  8 00:11 a.b
-rw-r--r--   1 whm      0 Sep  8 00:11 ab
-rw-r--r--   1 whm      0 Sep  8 00:11 ab
```

The process of replacing a wildcard-bearing argument with file names is called *wildcard expansion*. An old term, "globbing", is occasionally used.

It is important to understand that **the shell handles wildcard expansion**. It is not done by individual programs.

Wildcards, continued

If a wildcarded argument matches no files, it is passed unchanged to the program:

```
$ ls
62  X  a  a.b  ab  axe  b oxo

$ java args a?? ??a
|a.b|
|axe|
|??a|
```

Hidden files (which start with a dot) are not matched by a wildcard unless the pattern starts with a dot:

```
$ ls

$ touch .xyz

$ echo ????
????

$ echo .???
.xyz
```

Wildcards, continued

A more powerful wildcard is * (asterisk). It matches any sequence of characters, including an empty sequence.

*	Matches every (non-hidden) name
*.java	Matches every name that ends with .java
x	Matches every name that contains an x. Examples: x, ax, axe, xxe, xoxox
.*	Matches every hidden name, including . and ..

What would be matched by the following?

```
*old.*
```

```
*x*y
```

```
*.*
```

Handouts for this class are prepared using WordPerfect, which in turn generates PDF and PostScript files. Here's a command that the instructor uses from time to time:

```
ls -lt  *.wpd  *.pdf  *.ps | more
```

What's he doing?

Wildcards, continued

Wildcards are often combined. Examples:

`??*` Matches names that are two or more characters long

`*.?` Matches names whose next to last character is a dot

What would be matched by the following?

`?x?*`

`*-?-*`

`*~`

Wildcards, continued

It is possible to require that a single character be one of several:

[a-z] Matches names that consist of a single lowercase letter.

*.[hcy] Matches names whose suffix is .h, .c, or .y

[A-Z]*.[0-9]
Matches names that start with a capital letter and end with a dot and a digit.

*.[!0-9] Matches names that end with a dot and a non-digit character. Equivalent: *.[^0-9]

[Tt]ext Matches Text and text.

Problems:

At most, how many files could be matched with this pattern: [ab][cde][fghi]

A directory has a series of a files that are numbers starting at 1 and increasing. See if any files in the range 90-135 are missing

Move all directories whose names start with a lower case vowel into a directory named V. Move all directories starting with a lower case consonant into a directory named C.

Wildcards, continued

Slashes can be included in a pattern to match files elsewhere than the current directory. For example,

```
wc ~/*.java
```

runs wc on every Java source file in the user's home directory.

Problem: Describe the high-level operation performed by each of the following commands.

```
ls */TreeWalker.java

cat *.java */*.java */*/*.java >jsrc

ls -lt ~/.snapshot/hourly.?/352/*.notes

ln -s /home/cs352/fall04/a1/* .

ls -ld /????/???

echo /*/
```

Unfortunately, bash has no way to indicate a recursive match but find(1) provides a reasonable alternative.

There are other wildcard specifiers but ?, *, and [...], along with brace expansion are most commonly used.

Wildcards, continued

Problem:

This Windows command,

```
rename *.cc *.cpp
```

changes the extension of every .cc file to .cpp. (block.cc would become block.cpp, for example.)

Consider a well-intentioned analog with mv:

```
mv *.cc *.cpp
```

What would be the result? What conditions might influence the result?

Brace expansion

Common strings in a series of arguments can be concisely specified using *brace expansion*. The construct {*string1*,*string2*,...,*stringN*} expands to N arguments containing each string in turn.

Example:

```
$ ls
$ echo {one,two,three}
one two three

$ echo x-{one,two,three}.txt
x-one.txt  x-two.txt  x-three.txt

$ touch x-{one,two,three}.txt
$ ls
x-one.txt  x-three.txt  x-two.txt
```

Note that brace expansion does not simply plug existing file names into the command line, like wildcards do. Instead, it is synthesizing command-line text.

Brace expansion can be combined with wildcards.

```
ls *.{java,class}
```

is equivalent to:

```
ls *.java *.class
```

Brace expansion, continued

Brace expansion is done first, then wildcard matching is performed.

```
$ ls
x.java  y.java

$ echo *.{java,class}   (equivalent to echo *.java *.class)
x.java y.java *.class
```

Any number of brace expansions can appear in an argument:

```
$ echo {red,blue,green}_{sky,water,land}
red_sky red_water red_land blue_sky blue_water blue_land
green_sky green_water green_land
```

Brace expansion is sensitive to whitespace:

```
$ echo {a,b,  c}
{a,b,  c}
```

Explain this:

```
$ echo{a,b,c}
bash: echoa: command not found
```

What does the following expand to?

```
~{dmr,ken,rob}/bin/am{.sh,}
```

Filename completion

bash provides *filename completion*—it will "type" unique portions of filenames and show alternatives when appropriate.

Consider this directory:

```
$ ls
ListTest.d1  ListTest.java  TreeWalker.java
```

If you type,

```
$ javac T<TAB>
```

bash will "complete" the filename, producing this,

```
$ javac TreeWalker.java
```

and you can press <ENTER> to run javac.

If you type

```
$ javac L<TAB>
```

bash will produce this,

```
$ javac ListTest.
```

and beep. A second <TAB> will show the two alternatives. You might then type j<TAB>, which bash would respond to with

```
$ javac ListTest.java
```

Filename completion, continued

In some cases the functionality of wildcards overlaps with filename completion. Given these files,

```
$ ls
ListTest.d1  ListTest.java  TreeWalker.java
```

one might do the same two compilations like this:

```
$ javac T*
$ javac L*a
```

Shell Scripts

Basics

Scripts and I/O streams

Variables

Parameters

The for statement

The if statement

The while statement

Debugging shell scripts

Lots more with scripts...

Shell script basics

A *shell script* is simply a file that contains a series of shell commands.

Here is a script that prints the number of current login sessions:

```
$ cat ucount
echo -n Current logins:
who | wc -l | tr -s " "
```

One way to run a script is to run bash with the script as an argument:

```
$ bash ucount
Current logins: 44
```

It would be nice to only type ucount, but that produces an error:

```
$ ucount
bash: ./ucount: Permission denied
```

The problem is that the permissions of ucount do not indicate that it is executable. The chmod command adjusts permissions.

```
$ ls -l ucount
-rw-r--r-- 1 whm  dept    48 Jan 24 09:53 ucount
$ chmod +x ucount
$ ls -l ucount
-rwxr-xr-x 1 whm  dept    48 Jan 24 09:53 ucount
```

The script can now be run by simply naming it, like any other command:

```
$ ucount
Current logins: 44
```

Scripts and I/O streams

Redirecting a script's standard output produces a catenation of standard output of all the commands in the script.

For reference:

```
$ cat ucount
echo -n Current logins:
who | wc -l | tr -s " "
```

The output of ucount can be redirected:

```
$ ucount >out
$ cat out
Current logins: 44
```

The file out ends up with the output of each command in turn.

Redirecting the output of a script does not affect output redirections inside the script. Here is an oddly divided version of ucount that illustrates the point:

```
$ cat ucount2
echo -n Current logins:
who | wc -l >tmp
tr -s " " < tmp
$ ucount2 > out
$ cat out
Current logins: 44
```

Scripts and I/O streams, continued

Programs run inside a script "inherit" the standard input stream of the script. Example:

```
$ cat countbytes
wc -c

$ date | countbytes
      29
```

Here is a trivial script that avoids the nuisance of having to type "java" when running the lc.java utility:

```
$ cat lc
java lc

$ lc < args.java
6
```

Scripts and I/O streams, continued

The following script illustrates the technique of "capturing" standard input so that the data can be used as standard input for more than one program.

```
$ cat countboth
cat >.countboth.tempfile
echo -n "wc says: "
wc -l < .countboth.tempfile
echo -n "lc says: "
java lc < .countboth.tempfile
rm .countboth.tempfile
```

cat is used to create a temporary file that contains standard input. The file is then fed to wc and java lc as standard input for each of them.

Usage:

```
$ cal | countboth
wc says:       8
lc says: 8
```

What are three hazards posed by this simple-minded use of a temporary file?

Variables

bash provides for variables in a shell script. Here is a version of countboth that uses a variable to avoid repetitious (and possibly erroneous) specification of the temporary file name:

```
$ cat countboth2
tmp=.countboth.tempfile
cat > $tmp
echo -n "wc says: "
wc -l < $tmp
echo -n "lc says: "
java lc < $tmp
rm $tmp
```

There is no declaration of tmp. Variables come into existence when they are first assigned a value.

A dollar sign such as in $tmp indicates the following identifier should be treated as a variable and its value *interpolated*.

There can be no spaces on either side of the assignment operator.

If a value contains whitespace, use escapes or quotes: x="a b"

Unless special steps are taken, the value of a variable is considered to be a string.

Variables, continued

Concatenation of strings is accomplished by juxtaposing variables and, possibly, text.

An artificial example:

```
$ cat var1
who=Bob
a=apple
b=orange
sen="$who compared ${a}s and ${b}s"
echo "$sen all day long."
```

Execution:

```
$ var1
Bob compared apples and oranges all day long.
```

Speculate: What would

```
sen="$who compared $as and $bs"
```

produce?

Note that variable interpolation is done inside double quotes, but not single quotes:

```
$ x="abc"
$ echo "x is $x"
x is abc
$ echo 'x is $x'
x is $x
```

Script parameters

It is often useful to pass arguments to scripts. An argument can be fetched using $N, where N is the 1-based position of the parameter. Example:

```
$ cat printargs
echo The first argument is $1
echo Arg 2: "'"$2"'"
echo Third arg: $3

$ printargs just testing
The first argument is just
Arg 2: 'testing'
Third arg:

$ printargs Dots "  ....  " and more dots
The first argument is Dots
Arg 2: ' .... '
Third arg: and
```

If there is no Nth argument, $N expands to nothing.

Script parameters, continued

Here is a script that compiles a Java source file and then executes the resulting class file.

```
$ cat rj
javac $1.java
java $1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ${11}
```

Usage:

```
$ rj lc < lc.java
15

$ rj args a test here
|a|
|test|
|here|
```

What are two shortfalls in rj?

Script parameters, continued

The "special variable" $* expands to all command line arguments. We can use it to write a version of rj that is not limited to a fixed number of parameters:

```
$ cat rj2
javac $1.java
java $*
```

The command

```
$ rj2 args a b c d e f g h i j k l m
```

causes these commands to be executed.

```
javac args.java
java args a b c d e f g h i j k l m
```

Another special variable that is sometimes handy is $#. It expands to the command line argument count. Example:

```
$ cat numargs
echo There are $# command line arguments

$ numargs a b "c d" e
There are 4 command line arguments
```

Script parameters, continued

Problem: Write a script named cpxt that copies a file, adding an extension to the name.

The command

```
$ cpxt Search.java nullbug
```

would be equivalent to

```
$ cp Search.java Search.java.nullbug
```

The for statement

bash, like most shells, can be thought of as an interpreter for a programming language, albeit highly specialized for line-by-line interaction. It has several control structures, including if, while, for, and case.

A common use of the for loop is to iterate over the command line arguments. Here is an argument-printing script:

```
echo $# arguments:
for i in $*
do
    echo $i
done
```

Usage:

```
$ args just testing
2 arguments:
just
testing
```

$* expands to two "words", just and testing. The variable i is assigned each word in turn.

What does the following script do?

```
for i in $*
do
    echo -n "$i: "
    java lc < $i
done
```

for, continued

Here is the general form of for:

```
for i in words
do
    cmd1
    ...
    cmdN
done
```

The shell is very line-oriented. Here is a faulty version of args:

```
echo $# arguments:
for i in $* do echo $i done
```

Execution:

```
$ args just testing
2 arguments:
args: line 3: unexpected end of file
```

Speculate: What was the shell expecting to see next?

for, continued

At hand: "unexpected end of file" with this script:

```
echo $# arguments:
for i in $* do echo $i done
```

The problem is that the shell considers do, echo, $i, and done to be additional words to assign to i!

The loop can be made a one-liner by appropriately inserting semicolons:

```
for i in $*; do echo $i; done
```

A loop can be typed "on the fly", at the shell prompt:

```
$ mkdir some_long_directory_name
$ for i in src tests lib docs
> do
> mkdir some_long_directory_name/$i
> done
$
$ ls some_long_directory_name
docs  lib  src  tests
```

The shell is printing "**>** ", which is the *secondary prompt*. The secondary prompt is printed when the shell detects an incomplete expression.

The loop can be recalled for editing (with up-arrow). You'll see this:

```
for i in src tests lib docs; do mkdir some_long_d
irectory_name/$i; done
```

for, continued

Problem:

Write a script that echos its arguments in reverse order.

Example:

```
$ revargs they jumped up
up jumped they
```

Contrast: Here is a C shell version of args:

```
#!/bin/csh
echo $#argv arguments:
foreach i ($*)
    echo $i
end
```

The first line, `#!/bin/csh`, indicates that /bin/csh should be used to run the script.

The if statement

Here is the general form of the if statement in bash:

```
if command-line
then
    cmd1
    ...
    cmdN
fi
```

First, the command-line is executed. If the resulting exit code is zero, which indicates success, then the enclosed commands are executed.

Here is a script that compiles a Java program and, if the compilation is successful, runs the program:

```
if javac $1.java
then
    java $*
fi
```

Note that the special variable $? holds the exit code of the last process.

```
$ echo $?
0
$ grep lsdjfsdjfsl /etc/passwd
$ echo $?
1
$ true
$ echo $?
0
$ false
$ echo $?
1
```

if, continued

The test(1) command can be used to check for a variety of conditions such as whether a relation, like equality, holds between two values. It can also be used to test for file attributes such as existence, executability, etc.

test indicates success or failure by setting the exit code.

For example, test -f *file* tests to see if *file* exists and is a "regular file":

```
$ test -f args.java
$ echo $?
0
$ test -f xyz.java
$ echo $?
1
```

Here is a further refinement of the compile-and-run script:

```
if test $# -eq 0
then
    echo "Usage: $0 file (w/o .java!)"
    exit 1
fi
if test -f $1.java
then
    if javac $1.java
    then
        java $*
    fi
else
    echo $1.java: Not found
fi
```

Sidebar: The [command

Many shells consider [(left square bracket) to be an alias for the test command. A common practice is to use [, not test in scripts.

Another version of compile-and-run:

```
if [ $# -eq 0 ]
then
    echo "Usage: $0 file (w/o .java!)"
    exit 1
fi
if [ -f $1.java ]
then
    if javac $1.java
    then
        java $*
    fi
else
    echo $1.java: Not found
fi
```

The while statement

Here is the general form of the while statement in bash:

```
while command-line
do
    cmd1
    ...
    cmdN
done
```

Here is a cheap real-time clock via an on-the-fly while loop:

```
$ while true
> do        # REMEMBER: > is the secondary prompt!
> date
> sleep 1
> done
Mon Jan 24 17:08:12 MST 2005
Mon Jan 24 17:08:13 MST 2005
Mon Jan 24 17:08:14 MST 2005
^C
```

Here is a script that prints a banner when a user logs off. It checks every ten seconds.

```
$ cat waitfor
while who | grep -q $1
do
    sleep 10
done
banner $1 is off!
```

You might run it as a background process:

```
$ waitfor whm &
```

Debugging shell scripts

The primary tool for debugging scripts is the echo command, but the -v and -x options of bash are sometimes useful, too.

The -v option causes each command to be echoed before it is executed:

```
$ cat buggy
x=$2.$1
echo $x

$ bash -v buggy aa bb
x=$2.$1
echo $x
bb.aa
```

The -x option causes each command to be echoed after expansion:

```
$ bash -x buggy aa bb
+ x=bb.aa
+ echo bb.aa
bb.aa
```

With both:

```
$ bash -v -x buggy aa bb
x=$2.$1
+ x=bb.aa
echo $x
+ echo bb.aa
bb.aa
```

Lots more with scripts...

These slides address only very basic elements of shell scripts. *UNIX Power Tools* devotes about 100 pages to scripts. Entire books have been written about programming with shell scripts. However, **the material presented in these slides should be adequate to solve the shell-programming problems posed in the assignments.**

Shell scripts can solve many problems quickly but a growing script can slowly turn into a Frankenstein.

Here are some of the issues encountered when writing scripts that perform non-trivial computations:

Limited selection of data types
Syntactic irregularities
No debugging tools
Tedious quoting of values with metacharacters.
Slow execution

It is important to have a sense of when to move from a shell script to a program in an agile language[1] such as Python, Ruby, Perl or, the instructor's favorite, Icon.

One example of a large script is /home/cs352/fall04/gnu/diffutils-2.8.1/configure. It does platform-specific configuration of the source code for the GNU "diffutils". It is 15,482 lines long.

[1] Languages such as Icon, Python, Ruby, and Perl are often called "scripting languages" but the instructor thinks the term "agile language" is more accurate. The term "script" usually refers to an automated sequence of commands that could be performed by hand, but Icon, Python, Ruby, and Perl (and others) are designed for general purpose computation, not just orchestrating execution of commands.

The Shell—Part 2

More about variables

$PATH—The command search path

bash initialization files

Aliases

Functions

Command substitution

Named pipes

The history facility

The directory stack

More about shell variables

Variables can be created and used interactively. For example, to easily reference files associated with assignment 1, you might use a variable named a1:

```
$ a1=~cs352/spring05/a1
$ echo $a1
/home/cs352/spring05/a1
$ ls $a1
...output...
$ cp $a1/mgrep.class .
$ date | $a1/rev
5002 TSM 45:55:41 13 naJ noM
```

The shell has a collection of variables. The full set of variables and values can be displayed with the set command. Here is some heavily edited output:

```
$ set
COLUMNS=80
HOME=/home/whm
HOSTNAME=lectura.CS.Arizona.EDU
LINES=45
LOGNAME=whm
OLDPWD=/home/whm/icon/src
PS1='$ '
PWD=/home/whm/352
SHELL=/usr/local/bin/bash
TZ=US/Arizona
$ set | wc -l
     68
```

How could I copy a file from the directory I was previously in to the directory I'm currently in?

Variables, continued

It is often handy to set a shell variable and then use it inside a script or program but by default, variables are not transmitted to an executable. Example:

```
$ a1=~cs352/spring05/a1
$ echo $a1
/home/cs352/spring05/a1
$ cat vtest
echo a1 is $a1
$ vtest
a1 is
$
```

Note that a1 has a value on the command line, but not inside the script!

To direct the shell that a variable should be transmitted to an executable, the variable must be put on the *export list* by using the export command:

```
$ export a1
$ vtest
a1 is /home/cs352/spring05/a1
$ a1="something different"
$ vtest
a1 is something different
```

Note that an export need only be done once in a shell session.

With no arguments, export shows the variables that are currently on the export list. export -n *var* removes var from the export list.

Variables, continued

The set of variables (and associated values) that is exported is known as "the environment". The env command prints the environment:

```
$ env
PWD=/home/whm/352
TZ=US/Arizona
HOSTNAME=lectura.CS.Arizona.EDU
MSTKSIZE=100000
a1=something different
a2=/home/cs352/spring05/a2
[...lots more...]
```

A variable that is exported is commonly called an "environment variable".

Environment variables often influence program behavior. For example, date consults the environment variable TZ:

```
$ date
Mon Jan 31 16:09:04 MST 2005
$ TZ=US/Eastern
$ date
Mon Jan 31 18:09:16 EST 2005
```

If a variable assignment appears before a command, the assignment only exists for the execution of that command:

```
$ TZ=Asia/Baghdad date
Tue Feb  1 02:18:22 AST 2005
$ date
Mon Jan 31 16:18:29 MST 2005
```

Transmission of environment variables is "one way": changing the value of a variable in a script does not change the value in the caller.

Variables, continued

man consults MANPATH to see what man-page trees the user wishes to have searched, and in what order. A temporary setting might be used to see a different version of a man page:

```
$ echo $MANPATH
/usr/local/man:/usr/man:/usr/X11/man:/opt/unsuppo
rted/man
$ man ls

FSF                                                LS(1)

NAME
     ls - list directory contents
[...more...]

$ MANPATH=/usr/man man ls

User Commands                                      ls(1)

NAME
     ls - list contents of directory

[...more...]
```

Variables, continued

Certain variables control the behavior of bash. For example, PS1 specifies the shell's (primary) prompt string.

```
$ echo $PS1
$
$ PS1="Command? "
Command? date
Mon Jan 31 16:25:01 MST 2005
Command? who | wc
     73     365    2263
Command?
```

If the prompt contains certain escaped characters, the shell replaces those escapes with appropriate values.

Two of many:

\t is replaced with the current time
\w is replaced with the current directory

Example:

```
Command? PS1="\t \w $ "
01:29:14 ~/352 $ cd /
01:29:24 / $ cd /home/cs352
01:29:31 /home/cs352 $ cd
01:29:37 ~ $
```

The full set of escapes can be found on the bash man page—search twice for "PROMPTING". (Type /PROMPTING, then n at the more prompt.)

Variables, continued

The variable PS2 is the secondary prompt string. It is issued when an apparently fragmentary command line is present:

```
$ echo $PS2
>
$ echo 'testing
> this'
testing
this
$ ls |
> wc |
> wc
       1       3      24
```

$PATH–The command search path

An important variable is PATH. It lists the series of directories that bash is to search to find a command. Here is the instructor-recommended setting for lectura:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/uc
b:/opt/unsupported/bin:.
```

When the user enters a command such as ls, each directory in turn is searched for an executable named ls. The first one found, if any, is executed.

For better or worse, there may be many versions of an executable with a given name. For example, there are several versions of ls:

```
$ ls -l /usr/bin/ls /usr/local/bin/ls /usr/ucb/ls
-r-xr-xr-x root  19084 Apr  6  2002 /usr/bin/ls
-rwxr-xr-x root 297276 Mar 28  2000 /usr/local/bin/ls
-rwxr-xr-x root  13844 Apr  6  2002 /usr/ucb/ls
```

On lectura, /usr/local/bin contains both non-standard executables such as emacs, and improved ("improved"?) versions of standard executables (such as ls).

The search path controls both inclusion and preference of executables:

- Because /usr/local/bin in the path, the command emacs runs /usr/local/bin/emacs.
- Because /usr/local/bin precedes /usr/bin and /usr/ucb, the command ls runs /usr/local/bin/ls.

The search path, continued

For reference:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/uc
b:/opt/unsupported/bin:.
```

Here is a quick list of the directories that follow /usr/local/bin:

/usr/bin is where most standard executables reside.

On lectura, /bin is just a symlink to /usr/bin but on the Fedora Linux machines it is separate directory. By including it, the same PATH value suffices for both systems.

/usr/bin/X11 contains executables related to the X Window System.

/usr/ucb contains a number of executables that orginated at U. C. Berkeley.

/opt/unsupported/bin contains executables that have been installed and are maintained by volunteers, not the lab staff.

Dot (.) is the current directory.

The questions of presence, and position, of the current directory in the search path, have no simple answers. The document *To Dot or Not to Dot*, on the class website discusses the issues involved. The instructor believes that putting the current directory at the end of the path provides a reasonable balance between convenience and security.

The search path, continued

It is common for users to have their own "bin" directory that contains executables for personal use. Some users put their bin first in their path to be able to override standard versions of executables:

```
$ echo $PATH
/home/whm/bin:/usr/local/bin:/usr/bin:/bin:/usr/b
in/X11:/usr/ucb:/opt/unsupported/bin:.
```

A personal bin might include links (or copies) of standard programs, but with different names:

```
$ ln -s /usr/ucb/ps ~/bin/ups
$ ln -s /usr/bin/ls ~/bin/binls
```

A user might segregate binaries and scripts using two bins, ~/bin and ~/sbin, for example.

It is important to keep PATH and MANPATH "in sync":

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/uc
b:/opt/unsupported/bin:.
$ echo $MANPATH
/usr/local/man:/usr/man:/usr/X11/man:/opt/unsuppo
rted/man
```

Problem: Describe the possible results of this combination:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/ucb:.
$ echo $MANPATH
/usr/man:/usr/local/man:/usr/X11/man
```

The search path, continued

A common question is which directory contains a particular command. The type command answers that question:

```
$ type ls
ls is /usr/local/bin/ls
```

To speed up command search, bash will cache the location of a command the first time the command is executed:

```
$ type date
date is /usr/local/bin/date
$ date
Tue Aug 24 23:41:42 MST 2004
$ type date
date is hashed (/usr/local/bin/date)
```

The -a option of type shows all the occurrences:

```
$ type -a ls
ls is /usr/local/bin/ls
ls is /usr/ucb/ls
ls is /usr/bin/ls

$ type -a echo
echo is a shell builtin
echo is /usr/local/bin/echo
echo is /usr/ucb/echo
echo is /usr/bin/echo

$ type -a type
type is a shell builtin
type is /usr/bin/type
```

bash start-up files

The behavior of bash can be customized by putting commands in the initialization files that bash reads.

When bash is started as a login shell (the shell specified in a user's /etc/passwd entry), it looks for three files in turn: ~/.bash_profile, ~/.bash_login, and ~/.profile. Upon finding one, it executes the commands in that file (and doesn't look any further).

Here is a .bash_profile that sets the prompt, PATH, cs352, and a1. a1 and cs352 are then exported.

```
$ cat ~/.bash_profile
PS1="Command? "
PATH=/usr/local/bin/:/usr/ucb:/usr/bin
cs352=~cs352/spring05
a1=$cs352/a1
export cs352 a1
```

The file can be tested by "sourcing" it with the source command:

```
$ source ~/.bash_profile
Command? env | fgrep a1
a1=/home/cs352/spring05/a1
Command?
```

It is important to use the source command because source processes commands as if they had been typed at the prompt. It is usually a mistake to run a start-up file as a script (bash .bash_profile, for example) because then things such as assignments and exports survive for only the lifetime of the script.

bash start-up files, continued

When bash is started as a non-login shell, such as when run from tcsh, it looks for a file called ~/.bashrc but does not look for any of the files used when it is a login shell.

To get the same settings regardless of whether bash is a login shell, one could copy ~/.bash_profile to ~/.bashrc when changes are made, or link the files together with ln or ln -s.

Another option is to put all initializations in .bashrc and have .bash_profile source .bashrc:

```
$ cat ~/.bash_profile
source ~/.bashrc
```

If bash is a login shell, then .bash_profile is processed, and in turn it simply processes ~/.bashrc.

bash initialization files can be quite elaborate. It is common to see things such as conditional logic that sets PATH and other variables based on the type of system one has logged into.

Aliases

It is possible to create an *alias* for a command. Example:

```
$ alias jc=javac
$ jc Hello.java
```

If the expansion is more than one word, use quotes (or escapes):

```
$ alias lf="ls -F"
$ alias rm='rm -i'    (no problem with self-reference)
$ alias tf=TRACE=0
$ alias tn=TRACE=-1
```

With no arguments, alias shows the current set of aliases:

```
$ alias
alias jc='javac'
alias lf='ls -F'
alias rm='rm -i'
alias tf='TRACE=0'
alias tn='TRACE=-1'
```

The type command is aware of aliases:

```
$ type rm
rm is aliased to `rm -i'
```

It is common to define aliases in ~/.bashrc.

Functions

In many ways, bash scripts are like functions: they encapsulate a sequence of operations. But, bash also provides functions.

Here is a simple function, which might be found in a .bashrc:

```
mcd()
{
    mkdir $1
    cd $1
}
```

The function encapsulates a common operation: Making a new directory and going into it.

It is run like a script. Note that the prompt reflects the change of directory:

```
~/352 $ mcd tests
~/352/tests $
```

This example shows an important difference between scripts and functions: A cd in a script changes the directory for the script but not for the caller. A cd in a function does change the caller's directory. Similarly, an assignment in a script can change a variable in the caller.

There is a lot more to shell functions than is shown here.

Command substitution

The *command substitution* facility provides a way to turn the output of a command into command-line arguments. Example:

```
$ cat srcfiles
lc.java
mkall.icn
getpid.c
$ echo $(cat srcfiles)
lc.java mkall.icn getpid.c
```

On a command line, the form $(*command-line*) indicates to run the enclosed *command-line*, and substitute the whitespace-separated words it produces for the $(...) construct. The resulting command line is then executed.

Any number of command substitutions may appear on a command line and the enclosed commands may be arbitrarily complex.

More examples:

```
$ ls -l $(cat srcfiles)
-rw-r--r--   1 whm      74 Sep  1 14:23 getpid.c
-rw-r--r--   1 whm     360 Aug 14 18:54 lc.java
-rw-r--r--   1 whm     115 Aug 17 00:57 mkall.icn

$ wc $(cat srcfiles datafiles)
      15      36     360 lc.java
       6      16     115 mkall.icn
       6      13      74 getpid.c
       7      24     452 lc.1
      14      36     259 lc.2
      48     125    1260 total
```

Command substitution, continued

Another example:

```
$ wc $(cat srcfiles datafiles | sort -k 2 -t.)
      7      24     452 lc.1
     14      36     259 lc.2
      6      13      74 getpid.c
      6      16     115 mkall.icn
     15      36     360 lc.java
     48     125    1260 total
```

Note that the echo command and command substitution are inverses:

> *echo turns arguments into output; command substitution turns output into arguments.*

Consider this:

```
$ echo a b c
a b c
$ echo $(echo a b c)
a b c
```

What does the following script do?

```
for i in $*
do
    mv -i $i $(echo $i | tr A-Z a-z)
done
```

Command substitution, continued

What does the following command do?

```
yes "x" | head -n $(wc -l < lc.java) > lc.x
```

Problem: Run more on the files in the current directory that contain the word "if". Recall that fgrep -l (-L) prints only the names of files that contain a match.

Problem: Copy files in the current directory that contain the words "if" and "while" to a subdirectory named both.

Problem: Write a script mkprefix *pfx N* with this behavior:

```
$ ls
$ mkprefix x 5
$ ls
x.1  x.2  x.3  x.4  x.5
```

Hint: seq N prints the integers from 1 through N, one per line.

Command substitution, continued

The result of command substitution can be assigned to a variable.

```
$ f=$(fgrep -l Reader *.java)
$ wc $f
      15      36     359 lc.java
      11      37     278 lc2.java
      26      73     637 total
$ more $f
[...]
```

Here is a script that produces the sum of its arguments.

```
$ cat sum
sum=0
for i in $*
do
    sum=$(java eval $sum + $i)
done
echo $sum
```

Problem: Use sum to compute the sum of the integers from 1 through 100.

Command substitution is performed inside double-quoted literals. Here are a couple of lines from a script:

```
echo "Hostname: $(hostname | cut -f1 -d.)"
echo "Users:    $(echo $(who | wc -l))"
```

What is the purpose of the echo in the second line?

Command substitution, continued

Problem: Write an equally concise version of the following sentence without taking advantage of command substitution:

> *"To see what your login shell is, type this:* `finger $(whoami)`*"*

An older, but very commonly used form of command substitution is `...` (back-quotes):

```
finger `whoami`
wc `cat srcfiles datafiles | sort +1 -t.`
```

The older form is a little easier to type, but doesn't nest:

```
$ echo $(echo $(echo x))
x
$ echo `echo `echo x``
echo x
```

Process substitution

The *process substitution* facility of bash is best introduced via a task that is simplified by using the facility.

The cmp command can be used to see if two files differ:

```
$ cat a
Just testing...
$ cat b
just TESTING...
$ cmp a b
a b differ: char 1, line 1
$
```

To compare the files in a case-insensitive way, we could do this:[1]

```
$ tr A-Z a-z < a > a.s
$ tr A-Z a-z < b > b.s
$ cmp a.s b.s
$
```

Process substitution allows the comparison above to be expressed in one command. It looks like this:

```
cmp <(tr A-Z a-z < a) <(tr A-Z a-z < b)
```

Here's one way to think about it: The syntax <(*command*) causes the shell to generate an entity that can be read from as if it were a file. The data produced by this entity is the standard output stream of *command*.

[1] We'll see later that the -i option of diff is the Right Way to perform a case-insensitive comparison of two files.

Process substitution, continued

At hand:

```
cmp <(tr A-Z a-z < a) <(tr A-Z a-z < b)
```

Exploration:

```
$ echo cmp <(tr A-Z a-z < a) <(tr A-Z a-z < b)
cmp /dev/fd/63 /dev/fd/62

$ ls -l <(tr A-Z a-z < a) <(tr A-Z a-z < b)
crw-rw-rw- root  246,  62 Feb  2 10:17 /dev/fd/62
crw-rw-rw- root  246,  63 Feb  2 10:17 /dev/fd/63
```

Problem: See if files v1.out and v2.out contain the same lines but in differing orders, ignoring lines that contain the word "test".

History

The shell maintains a "history" of command lines.

The history command displays the accumulated list:

```
$ history
   41  pwd
   42  ls
   43  ls -l
   44  javac args.java
   46  java args a test here
   47  history
```

The HISTSIZE variable controls the size of the history list. The default is 500. (The above list is abbreviated!)

The last command can be recalled and executed by typing !! ("bang-bang"):

```
$ date
Mon Sep  6 22:07:36 MST 2004
$ !!                              (typed by user)
date                              (printed by shell)
Mon Sep  6 22:07:37 MST 2004
```

Note that the command is echoed, then executed.

A previous command can be re-executed with !*N*, where *N* is the number of the command in the history list:

```
$ !44
javac args.java
```

History, continued

Another way to recall a command is with !*prefix*. The history is searched in reverse order for a command that starts with *prefix*. If a command starting with *prefix* is found, it is re-executed:

```
$ !javac
javac args.java
```

Sometimes a prefix search produces an unexpected result. A user might try to re-execute java args like this,

```
$ !java
javac args.java
```

and have a javac command re-executed instead. (Why?)

Adding ":p" to a history selection causes the selected command to only be printed, but the command is added to the history list. If the command is the one desired, it can then be run with !!:

```
$ !fg:p
fgrep x words | fgrep y  (not run, only printed)
$ !!
fgrep x words | fgrep y  (printed, then executed)
```

Be especially careful with one letter searches, like !r, !c, !m — you might be surprised with an rm, cp, or mv!

If a search fails, an error is produced:

```
$ !xyz
bash: !xyz: event not found
```

History, continued

The construct !$ is replaced with the last argument of the last command:

```
$ ls -l deepest.1
-rw-r--r--   1 whm   73 Jan 25 21:22 deepest.1

$ cp !$ ~cs352/spring05/a1
cp deepest.1 ~cs352/spring05/a1

$ cd !$
cd ~cs352/spring05/a1
```

Another example:

```
$ more SomeLongName.java
[...]
$ cp !$ !$.bak
cp SomeLongName.java SomeLongName.java.bak
```

The construct !* specifies all arguments of the last command line.

```
$ more *.out x* *.bak
[...files examined -- nothing worth keeping...]
$ rm !*
rm *.out x* *.bak
```

Appending :p to either !$ or !* causes the command to be printed and put in the history list, but not executed.

History, continued

A common practice is to search the history list and then select a command by number:

```
$ history | fgrep java
  390  java mgrep x x.java
  392  java mgrep x mgrep.java x.java
  397  java mgrep x x
  398  java mgrep x x x22
  402  java mgrep
  415  java -cp /home/cs352/spring05/a1 mgrep 1 x
  578  javap -c java.lang.String
  595  java mgrep 1 x
$ !415
java -cp /home/cs352/spring05/a1 mgrep 1 x
```

Handy: alias hg='history | fgrep'. (Usage: hg java)

A series of commands might be pieced together and repeatedly executed as a unit:

```
$ history | tail
...
  671  javac mgrep.java
  672  java mgrep csh /etc/passwd >mine
  673  diff -c mine ref
$ !671; !j; !!
javac mgrep.java; java mgrep csh /etc/passwd
>mine; diff -c mine ref
[...edit...]
$ !j
javac mgrep.java; java mgrep csh /etc/passwd
>mine; diff -c mine ref
```

History—odds and ends

Sometimes it's useful to change every occurrence of a string in the last command. Example:

```
$ cp ../372/elisp.sli.wpd emacs.sli.wpd
$ !!:gs/sli/nts/
cp ../372/elisp.nts.wpd emacs.nts.wpd
```

Alterations to the last command are usually done by recalling it with up-arrow (or C-p) but ^*old*^*new* can be used:

```
$ fgrep
$ cut -f 2 -d : /etc/passwd
[...]
$ ^2^3
cut -f 3 -d : /etc/passwd | head
```

There is an obvious overlap in functionality between the history mechanism and command line editing via Emacs-style bindings (or vi-style bindings via set -o vi). Mix and match as so inclined!

When a bash login shell exits, the history is saved to ~/.bash_history. The number of commands saved is controlled by HISTFILESIZE.

Adding \# to the variable PS1 causes the number of the current command to be displayed in the prompt:

```
$ PS1="\# $ "
658 $ date
Thu Feb  3 02:12:18 MST 2005
659 $
```

And as usual...There is a lot more to the history mechanism than is shown in these slides.

The directory stack

The shell maintains a *directory stack*. The cd and pwd commands change and query, respectively, the top element of the directory stack.

The dirs command displays the contents of the directory stack:

```
$ pwd
/home/whm
$ dirs
~
```

The pushd command pushes a directory on the stack and changes to it; the popd command removes the top entry from the stack and changes to the then-top entry:

```
$ pushd ~/352
~/352 ~
$ pushd ~cs352/spring05
/home/cs352/spring05 ~/352 ~
$ popd
~/352 ~
$ pushd    (with no argument, swaps the top two)
~ ~/352
```

Some alias suggestions:

```
alias pwd=dirs
alias to=pushd
alias bk=popd
```

Experiment: Push several directories on the stack and try pushd +1.

Assorted Utilities

diff (and patch)

find

tar

Regular expressions and the grep family

sed

diff

diff is a sophisticated tool for comparing one file to another.

diff is run by naming two files. The default output format is cryptic, but the -c (context) format is easily understood. Example:

```
$ diff -c users.1 users.2
*** users.1     Tue Sep  7 11:59:09 2004
--- users.2     Wed Sep  8 23:28:21 2004
***************
*** 1,6 ****
  ralph
! jhh
  pete
  phil
- debray
  greg
--- 1,6 ----
  ralph
! jcropper
  pete
  phil
  greg
+ gmt
```

users.1	users.2
ralph	ralph
jhh	jcropper
pete	pete
phil	phil
debray	greg
greg	gmt

The first two lines of output show the involved files. The next line, all asterisks, precedes each section of differences (only one, in this case).

The next line, `*** 1,6 ****`, indicates that lines 1-6 of the first file immediately follow. The three asterisks correspond to the asterisks in `*** users.1`

Further down, `--- 1,6 ----`, indicates that lines 1-6 of the second file come next. The three dashes correspond to the dashes in `--- users.2`.

diff, continued

For reference:

```
$ diff -c users.1 users.2
*** users.1     Tue Sep  7 11:59:09 2004
--- users.2     Wed Sep  8 23:28:21 2004
***************
*** 1,6 ****
  ralph
! jhh
  pete
  phil
- debray
  greg
--- 1,6 ----
  ralph
! jcropper
  pete
  phil
  greg
+ gmt
```

users.1	users.2
ralph	ralph
jhh	jcropper
pete	pete
phil	phil
debray	greg
greg	gmt

Three differences are shown between the two files:

The line jhh in one file is jcropper in the other. "!" indicates a line that differs between the two files.

The line debray in users.1 is not in users.2. "-" indicates this line is not present in the other file.

The line gmt in users.2 is not in users.1. "+" indicates this line is not present in the other file.

In most cases, the differences are reported as concisely as possible.

diff, continued

For reference:

```
$ diff -c users.1 users.2
*** users.1     Tue Sep  7 11:59:09 2004
--- users.2     Wed Sep  8 23:28:21 2004
***************
*** 1,6 ****
  ralph
! jhh
  pete
  phil
- debray
  greg
--- 1,6 ----
  ralph
! jcropper
  pete
  phil
  greg
+ gmt
```

users.1	users.2
ralph	ralph
jhh	jcropper
pete	pete
phil	phil
debray	greg
greg	gmt

Another way to think about the report is in terms of modifications to the first file that would produce the second file:

Change jhh to jcropper.

Delete debray.

Add gmt.

diff, continued

Here is a diff showing several sections of differences. Note that if the change in a section is solely insertion or deletion, only one "side" is shown. The -c1 option restricts the context to one line.

```
$ diff -c1 100 100.mod
*** 100     Thu Sep  9 00:04:46 2004
--- 100.mod Thu Sep  9 00:08:18 2004
***************
*** 12,15 ****
  12
- 13
  14
  15
--- 12,15 ----
  12
  14
+ a
  15
***************
*** 24,26 ****
  24
- 25
  26
--- 24,25 ----
***************
*** 66,67 ****
--- 65,67 ----
  66
+ b
  67
***************
*** 89,91 ****
  89
! 90
  91
--- 89,92 ----
  89
! ninety
! 90.1
  91
```

diff, continued

By default, diff is very strict—two files are considered identical only if character by character their contents are the same. In such a case, diff produces no output:

```
$ cp users.1 users.3
$ diff users.1 users.3
$
```

diff has many options to relax the comparison in various ways. The -b option ignores differences in runs of whitespace:

```
$ cat text.1
to be
or not to
be
$ cat text.2
to   be
or    not   to
b  e
$ diff -c1 -b text.[12]
*** text.1      Thu Sep  9 01:04:19 2004
--- text.2      Thu Sep  9 01:05:25 2004
***************
*** 2,3 ****
  or not to
! be
--- 2,3 ----
  or    not   to
! b  e
```

With -w, all whitespace is ignored:

```
$ diff -w text.[12]
$
```

diff, continued

The -r option directs diff to recursively compare directories.

```
$ ls -R
.:
a  b

./a:
w  x  y

./b:
w  x  z
$ cat a/x
line 1
line 2
line 3
$ cat b/x
line 1
line two
line 3
$ diff -r -c1 a b
diff -r -c1 a/x b/x
*** a/x Thu Sep  9 01:50:24 2004
--- b/x Thu Sep  9 01:50:48 2004
***************
*** 1,3 ****
  line 1
! line 2
  line 3
--- 1,3 ----
  line 1
! line two
  line 3
Only in a: y
Only in b: z
```

diff, continued

If one diff operand specifies a file and the other specifies a directory, the file name is assumed to be the same in the directory. For example, if v1 is a directory,

```
diff Parser.java v1
```

is equivalent to

```
diff Parser.java v1/Parser.java
```

Process substitution is very handy with diff. What is the following command doing?

```
diff <($a5/fill 10 < f1 ) <(fill 10 < f1 )
```

diff has many output formats. You might experiment with the -e, -u, and -y options.

Some related programs:

cmp compares files but simply reports the first difference, and exits:

```
$ cmp users.1 users.2
users.1 users.2 differ: char 9, line 2
```

diff3 does three-way differencing.

xdiff and meld provide graphical interfaces for viewing differences.

patch

patch makes changes to a file based on a "diff", which is the output of a diff run. Here is a simple scenario:

> Version 1 of Probe.cc is developed and distributed in source form.
>
> Version 2 of Probe.cc is created. A diff of the two versions is produced:
>
> > diff -c v[12]/Probe.cc >v1-v2.diff
>
> The diff is made available along with version 2 of Probe.cc.
>
> A user can upgrade to version 2 in two ways: (1) Fetch the new version of Probe.cc; or, (2) Fetch the diff and use patch to transform version 1 into version 2. Here is a slightly edited version of a patch run:

```
$ patch < v1-v2.diff
Hmm...  Looks like a new-style context diff to me...
The text leading up to this was:
--------------------------
|*** v1/Probe.cc
|--- v2/Probe.cc
--------------------------
Patching file v1/Probe.cc using Plan A...
Hunk #1 succeeded at 1034.
Hunk #2 succeeded at 2223.
Hunk #3 succeeded at 3455.
done
```

patch can typically accommodate changes that the user has made to version 1. Also, patch handles diffs made with -r.

The find command

The find command recursively searches a directory tree (or trees) for files matching specified criteria.

The simplest use is to specify a directory name. find then prints the name of every directory entry in the tree:

```
$ /bin/pwd
/home/cs352/fall04

$ find .
.
./admin
./admin/test.java
./admin/test.class
./bin
./bin/showmail
./bin/jikesx
./bin/jikes
./bin/jikes~
./mail
...lots more...

$ find . | wc -l
   4269

$ find etc
etc
etc/bash_profile
etc/.bash_profile
etc/.bashrc
etc/bashrc
```

Note that hidden files are shown but . and .. are special-cased.

If no directory is specified, the default is the current directory.

find, continued

A number of "tests" are available to constrain the results.

A commonly-used test is -name, which requires that files match a wildcard specification:

```
$ find -name "*.java"
./rev.java
./args.java
./Hello.java
./lc.java
./errout.java
./dir.java
./a1/rev.java
./a1/sum.java
./a1/iota.java
./a1/echo.java
...

$ ls -l $(find -name "[A-Z]*.java")
-rw-r--r-- 1 whm 127 Aug 13 14:32 ./Hello.java
-rw-r--r-- 1 whm 598 Sep  8 21:35 ./tmp/Scanner.java

$ more $(find -name notes)
...
```

Problem: A friend says that sometimes find works and sometimes it doesn't. Here is the command being used:

```
$ find -name *.java
```

What's happening?

find, continued

find has a number of tests related to file access and modification times. Here is a search for files that were modified in the last two "days":

```
$ find -mtime -2
.
./work
./work/outline.notes.0105
./work/outline.notes.0138
./rev.java
./tmp
./tmp/TreeWalker.java
./tmp/Scanner.java
...
```

Note that both ordinary files and directories are produced.

The specification -mtime -2 means modified in the last 48 hours; -mtime +3 means last modified more than 72 hours ago; -mtime 2 means last modified between 48 and 72 hours ago. Adding -daystart causes a shift to calendar days: find -daystart -mtime 0 produces files modified today.

If more than one test is specified an AND'ing of tests results. Here's a search for *.java files modified in the last three days:

```
$ find -name "*.java" -mtime -3
```

find, continued

More examples of searches:

Look for files over 100K in size that are over 200 days old:

```
$ find -size +100k -mtime +200
```

Find directories named test:

```
$ find -name test -type d
```

A test can be negated by preceding it with ! (or -not), one of find's "operators":

```
$ find ! -name \*.java
```

Problem: The above command produces names like x.java~. Exclude them, too.

Speculate: What does the following search look for?

```
$ find -newer x ! -empty ! -user whm
```

Another operator is -o, for "or". Grouping can be done with parentheses, but they must be escaped, and whitespace is significant.

```
find -size +10k -o \( -name \*.java -mtime +10 \)
```

find, continued

In addition to tests, find has "actions". Once upon a time, the -print action was required to print the names:

```
find . -print
```

Most current versions assume -print if no other actions are specified.

The -ls action causes 'ls -lids' to be performed on each file found:

```
$ find ~cs352/fall04/etc -ls
3409699    4 drwxr-sr-x   2 whm 4096 Sep 12 14:06 etc
3409701    0 lrwxrwxrwx   1 whm   13 Sep 10 18:03
etc/bash_profile -> .bash_profile
3409705    0 -rw-r--r--   1 whm   12 Sep 10 17:29
etc/.bash_profile
3409706    4 -rw-r--r--   1 whm  362 Sep 11 17:46 etc/.bashrc
3409703    0 lrwxrwxrwx   1 whm    7 Sep 10 18:04 etc/bashrc
-> .bashrc
```

find, continued

A very general action is -exec. It runs a specified command.

```
$ find -name "[A-Z]*.java" -exec wc -l {} \;
      5 ./Hello.java
     15 ./tmp/TreeWalker.java
```

The command starts with the argument following -exec and continues to the next semicolon. The semicolon must be escaped so that it is passed to find as an argument, and not intercepted by the shell. For each match, the path to the entry is substituted for **{}**.

Here's how the instructor removes his Emacs backup files from a tree:

```
$ find -name ".ZBK.*.ZBK" -exec rm {} \;
```

Note that shell aliases do not come into play with -exec. Even if rm is aliased to rm -i, no prompting is done.

Problem: What's a quick way to see which files find will remove and then, with a second command, remove them?

Problem: Find .h and .c files and make copy of each, adding a .bak suffix. For example, x.h would be copied to x.h.bak.

Problem:

> Imagine a GUI-based version of find, where you build up the search by clicking options. Would you rather use it or the command-line version? (For the rest of your career...)

Sidebar: find vs. .snapshot

Ugly problem: When in your home directory on lectura, find descends into your .snapshot directory:

```
$ cd
$ find
.
./!
./372
./372/a4
./372/a4/i1.icn
./372/a4/n
./372/a4/i1
...lots of output...
./.snapshot
./.snapshot/hourly.7
./.snapshot/hourly.7/!
./.snapshot/hourly.7/372
./.snapshot/hourly.7/372/a4
./.snapshot/hourly.7/372/a4/i1.icn
```

End result: find's in your home directory are painful. The instructor has only about 10,000 files in his tree but look at this:

```
$ find ~ | wc -l
 167913
```

find vs. .snapshot, continued

Solution: Find which NFS server your home directory is on and make a symlink, perhaps "net", to your directory on that server:

```
$ cd
$ df .
Filesystem            1k-blocks      Used
Available Use% Mounted on
sinagua:/vol/vol0/home/whm
                      228789416 188377520
40411896  82% /home/whm

$ ln -s /net/sin/vol/vol0/home/whm net
```

You can then go to ~/net and do a find there, or do this:

```
find ~/net/.
```

The trailing /. is required because by default, find does not follow symbolic links.

The tar command

tar is the "tape archiver" but it is most widely used to create an "archive" that contains all files in a tree or inversely, create a tree of files from a tar archive.

Here's a small tree to work with:

```
$ ls -R p1
p1:
args.java  lc.java  x  y

p1/x:
a  b

p1/y:
```

To make a "tar file" (or "tar") of p1, do this:

```
$ tar cvzf p1.tz p1
p1/
p1/x/
p1/x/a
p1/x/b
p1/args.java
p1/y/
p1/lc.java
$ ls -l p1.tz
-rw-r--r--   1 whm    638 Sep 12 21:07 p1.tz
```

The first argument is an option string that indicates to create (c) a tar file, be verbose (v), compress the output (z), and write the result to a file (f), not a tape. The second argument, p1.tz is the file to create. The third argument is the directory to "tar up".

tar, continued

The table of contents of a tar file can be displayed with the t option:

```
$ tar tvzf p1.tz
drwxr-xr-x whm/dept    0 2004-09-12 21:05 p1/
drwxr-xr-x whm/dept    0 2004-09-12 21:04 p1/x/
-rw-r--r-- whm/dept   29 2004-09-12 21:04 p1/x/a
-rw-r--r-- whm/dept  139 2004-09-12 21:04 p1/x/b
-rw-r--r-- whm/dept  180 2004-09-12 20:53 p1/args.java
drwxr-xr-x whm/dept    0 2004-09-12 21:03 p1/y/
-rw-r--r-- whm/dept  360 2004-09-12 20:53 p1/lc.java
```

Without the v (verbose) option, only paths are shown with t:

```
$ tar tzf p1.tz
p1/
p1/x/
p1/x/a
...
```

tar, continued

The x option extracts the contents of an archive, reproducing the original tree:

```
$ mkdir v2
$ cd v2
$ tar xvzf ../p1.tz
p1/
p1/x/
p1/x/a
p1/x/b
p1/args.java
p1/y/
p1/lc.java
$ ls
p1

$ ls -R p1
p1:
args.java  lc.java  x  y

p1/x:
a  b

p1/y:
```

In this case the tar was unloaded into an empty directory. If a tree p1 had already existed, tar would have added and/or overwritten files and directories in that tree.

tar, continued

Individual files and/or directories can be extracted:

```
$ tar xvf ../p1.tar p1/y p1/x/b
p1/x/b
p1/y/
$ ls -R p1
p1:
x  y

p1/x:
b

p1/y:
```

An issue that requires a bit of thought when creating a tar is how it is "rooted". p1.tar is rooted at p1:

```
$ tar tf p1.tar
p1/
p1/x/
p1/x/a
p1/x/b
p1/args.java
p1/y/
p1/lc.java
```

Because p1.tar is rooted at p1, extracting files causes the directory p1 to be created.

tar, continued

An alternative is to root a tar at the current directory:

```
$ tar cvf ../p1d.tar .
./
x/
x/a
x/b
args.java
y/
lc.java

$ mkdir v3
$ cd v3
$ tar xvf ../p1d.tar
./
x/
x/a
x/b
args.java
y/
lc.java
$ ls
args.java  lc.java  x  y
```

With this approach, no directory named p1 is created.

Such an archive can be said to be "rooted at dot".

tar, continued

tar is great for making a snapshot of work in progress. Here is one example:

```
$ cd ~/352
$ tar cvzf ../352.091204.tz .
```

Here's another way to snapshot:

```
$ cd
$ tar cvzf 352.091204.tz 352
```

Don't do this:

```
$ tar cvzf t.tz .
...output...
tar: t.tz: file changed as we read it
```

Because the output file, t.tz, is in the current directory it is included in the archive, and tar notices that the file grew as the archive was being created.

Another mistake:

```
$ tar cvzf ~/352.tz ~/352
```

~/352 expands to an absolute path such as /home/whm/cs352. When files are extracted, they'll overwrite the files currently in /home/cs352/home—probably not what is desired. It is almost always a mistake to create an archive that contains absolute path names.

tar, continued

The T option directs tar to read a file that in turn lists the files to include in the tar:

```
$ find . -name "*.java" >jfiles
$ tar cvzTf jfiles jfiles.tz
```

tar can read from standard input and write to standard output, indicated by '-' for the input or output file:

```
$ tar czf - p1 | supercrypt > p1.cpt
$ supercrypt -d p1.cpt | tar xzf -
```

Here is a pipeline that uses a subshell to copy a directory tree.

```
$ /bin/pwd
/home/whm/352
$ tar cf - . | (mkdir ../352.2; cd ../352.2; tar xf -)
```

tar has many options but the subset presented here is adequate for a wide variety of needs.

A program popular with some users that's similar to tar is cpio.

Sad but true:

> On lectura, GNU tar (described herein) is in /usr/local/bin/tar. Unfortunately, there is no man page for it, but --help does work.
>
> "man tar" produces the man page for /usr/bin/tar, which differs from GNU tar. (Two examples: Option 'z' is not supported and instead of 'T' there is '-I' (I).)

The fgrep command

The fgrep command prints lines that match a pattern. A simple example:

```
$ fgrep print *.java
Hello.java:        System.out.println("Hello, world!");
args.java:           System.out.println("|" + args[i] + "|");
dir.java:    void print()
dir.java:            System.out.println(entries[i].inode_number + ": "
+
dir.java:        d.print();
dir.java:        d2.print();
lc.java:        System.out.println(count);
```

Each .java file is processed line by line and if a line contains 'println', it is printed, preceded by the name of the file that contains it.

fgrep can read standard input:

```
$ cat *.java | fgrep print
        System.out.println("Hello, world!");
           System.out.println("|" + args[i] + "|");
    void print()
           System.out.println(entries[i].inode_number + ": " +
        d.print();
        d2.print();
        System.out.println(count);
```

fgrep, continued

The -l (L) flag causes fgrep to simply print the names of files that have an occurrence of the pattern:

```
$ fgrep -l print *.java
args.java
lc.java
```

The -v causes inversion—non matching lines are printed:

```
$ fgrep -v print args.java
public class args {
    public static void main(String args[]) {
        for (int i = 0; i < args.length; i++)
        }
    }
```

Problems:

Find lines initializing an Object array with "new" of some sort.

Find println calls that don't contain a double-quote but only consider Java files that contain the string "Test".

Other handy options (among many):

-w searches for whole "words".
-c prints a count of matching lines.
-n prints line numbers.
-C prints surrounding lines (five-line "window" by default).
-e is used like this: 'fgrep -e -x ...', to search for -x.
-f *file* reads patterns from a file.

Regular expressions

In computer science theory, a language is a set of strings. The set may be infinite.

The Chomsky hierarchy of languages looks like this:

Unrestricted languages	("Type 0")
Context-sensitive languages	(Type 1)
Context-free languages	(Type 2)
Regular languages	(Type 3)

Roughly speaking, natural languages are *unrestricted languages* that can only specified by *unrestricted grammars*.

Programming languages are usually *context-free languages*—they can be specified with a *context-free grammar*, which has very restrictive rules. Every Java program is a string in the context-free language that is specified by the Java grammar.

A *regular language* is a very limited kind of context free language that can be described by a *regular grammar*. A regular language can also be described by a *regular expression*.

A regular expression is simply a string that may contain metacharacters. Here is an example of a regular expression:

```
^\([A-Z].*[ab]\?\)\|[a-fA-F]\{3,5\}
```

Regular expressions have some similarities to filename wildcards but the two facilities are used in different contexts and behave very differently in most cases.

Regular expressions, continued

grep behaves like fgrep but instead of searching for literal text, it assumes its first argument is a regular expression.

In a regular expression, all alphanumeric characters (and some special characters) match themselves. The commands

```
$ fgrep abc123 x
```

and

```
$ grep abc123 x
```

produce the same results.

Note that abc123 is a regular expression that describes a regular language that contains one string: abc123

The simplest RE (regular expression) metacharacter is the period (or "dot"). In a RE, a dot matches any character. Example:

```
$ grep "a.b.c" /usr/dict/words
albacore
barbecue    (note that match is not at beginning)
canvasback
drawback
iambic
[...more...]
```

What is matched by the following?

```
$ grep "....." /usr/dict/words
```

As a matter of habit, enclose regular expressions in quotes.

Regular expressions, continued

A caret (^) at the beginning of a regular expression requires that the following RE appear at the start of a line.

```
$ grep "^spot" /usr/dict/words
spot
spotlight
spotty
```

A dollar sign requires that the line end with the preceding RE.

```
$ grep "spot$" /usr/dict/words
despot
spot
sunspot
tenspot
```

Problems:

Print words that are exactly four characters long.

How many words have an "a" as their third character?

Does the file lc.java contain any empty lines?

What is the longest word in the dictionary?

Speculate: Are the quotes really needed around "^spot" and "spot$"?

Regular expressions, continued

Another rule:

> *R** matches zero or more occurrences of the regular expression *R*.

Example:

> ab*c Matches lines that contain an 'a' that is followed by zero or more 'b's that are followed by a 'c'. Examples: ac, abc, abbbbbbc, back, cache, Babcock.

Problems:

> Find words that start with 'a' and end with 'b'.
>
> Describe lines matched by ^a*b.*c..$
>
> Describe lines matched by if.*f(.*)
>
> Find words that have all vowels in order.
>
> Speculate: What does a*.\. match?

Tip: To experiment with grep, run it with no input. Then, type lines of test input. The lines that match are echoed.

Example: What does the RE *.c match? Find out:

```
$ grep "*.c"
```

Regular expressions, continued

Regular expressions have precedence rules. The * operator has higher precedence than juxtaposition (one character next to another).

Example: a.*b is interpreted as a(.*)b rather than (a.*)b .

The operator * is "greedy"—it tries to match the longest string possible, and cuts back only to make the full expression succeed. Example:

Given a.*b and the input 'abbb', the first attempt is:

a	matches a
.*	matches bbb
b	*fails*—no characters left!

The matching algorithm then *backtracks* and does this:

a	matches a
.*	matches bb
b	matches b

Regular expressions, continued

Another rule:

> The notation [*characters*] is a RE that matches any one of the specified *characters*. *[^characters]* is an RE that matches any character not in the set. (It matches the *complement* of the set.) A dash between two characters in a set specifies a range.

For example, ^[AEIOU].*[^0-9]$ matches lines that start with a capital vowel and don't end with a digit.

Problems:

> Find words that contain a capital letter in a position other than the first.
>
> Strings that match [A-Za-z_][A-Za-z_0-9]* commonly occur in programs. What are they?
>
> Find lines that have an if statement and one of these comparisons: !=, ==, <=, >=
>
> Are there any non-alphabetic characters in /usr/dict/words?

Regular expressions, continued

If *R* is a regular expression, *R*\+ matches one or more instances of *R.*

If *R* is a regular expression, *R*\? matches zero or one instances of *R.*

Example:

> [a-z]\+[0-9]\? matches one or more lower case letters and, possibly, a digit.

Like *, \+ and \? have higher precedence than juxtaposition.

REs can be grouped with (escaped) parentheses to override precedence rules.

Example:

> \(ab\)\+ matches strings like ab, abab, ababab, etc.

Describe the input lines that would be printed by this grep invocation:

```
$ grep "^\(\(ab\)\+c\?\(xyz\)*\)\?$"
```

The sequence \| provides an "or" capability. Example:

```
\(one\|two\|three\) \(apple\|biscuit\)s\?
```

Speculate: What does the following RE match?

```
one\|two\|three apple\|biscuits\?
```

Regular expressions, continued

Many programs have some support for regular expressions. However, the set of metacharacters can vary from program to program. Example:

> grep considers +, ?, | and parentheses to have special meaning only when escaped. egrep considers those same characters to be metacharacters unless escaped.
>
> The grep RE \(a\|bc\|de\)\+ has this egrep equivalent: (a|bc|de)+
>
> Problem: Specify the egrep equivalent for the grep RE "(+)".

The set of RE operations varies from program to program. For example, some programs provide one or more of the following:

> \<print\> matches the string print when it appears as a "word".
>
> [0-9]\{1,6\} matches strings of 1-6 digits.
>
> Lines that have nothing but whitespace: ^[[:space:]]*$

The java.util.regex package supports many more constructs.

The pinnacle of support for regular expressions (or deepest pit, depending on your view) is Perl.

As a rule, however, you can do a lot with just these:

. * + ? [...] [^...] ^ $

The sed command

sed is the "stream editor". It is designed to perform one or more transformations on a stream of input text.

The s/*from*/*to*/ command of sed performs simple textual substitution:

```
$ date | sed 's/0/zero/'
Mon Feb  7 17:zero5:03 MST 2005

$ date | sed 's/0/zero/g' (g is "global")
Mon Feb  7 17:zero5:zero6 MST 2zerozero5

$ echo $a1 | sed 's./. .g'  (or 's/\// /g')
 home cs352 spring05 a1
```

sed recognizes regular expressions:

```
$ cal | sed 's/[0-9]\+/<num>/g'
   February <num>
 S  M Tu  W Th  F  S
       <num>  <num>  <num>  <num>  <num>
 <num>  <num>  <num>  <num> <num> <num> <num>
<num> <num> <num> <num> <num> <num> <num>
...

$ sed 's/(.*)/()/' < args.java
public class args {
    public static void main() {
        for ()
            System.out.println();
...
```

Problem: Use sed to strip // comments from a Java source file. Take a naive approach. For example, don't worry about a literal like "=//=".

sed, continued

In the "to" field of sed's s command, an ampersand specifies the text matched by the "from" field. Example:

```
$ date | sed 's/[0-9]\+/(&)/g'
Mon Feb  (7) (17):(36):(53) MST (2005)
```

Elements of the "from" field that are grouped with parentheses can be plugged into the replacement with \N.

Example:

```
$ echo /a/b/x.c | sed 's;\(.*\)/\(.*\);\2 in \1;'
x.c in /a/b
```

Several sed commands may be specified on the command line. The following command deletes whole-line comments and strips intra line comments.

```
$ sed -e '/^\/\//d' -e 's;//.*;;' < x.java
...
```

A sequence of sed commands can be put into a file:

```
$ cat stripcmt
/^\/\//d
s;//.*;;
$ sed -f stripcmt < args.java
...
```

sed has many more capabilities than are discussed here. One example: `sed -n -e 1p -e '$p'` prints the first and last line of standard input.

Why call it "grep"?

The ed editor was one of the first applications to use regular expressions. ed's g (global) command applies a series of commands to each line that matches a regular expression.

Example:

```
$ ed lc.java
360
g/=/p
        BufferedReader in =
        int count = 0;
        while ((line = in.readLine()) != null)
q
$
```

This operation, printing lines in a file that match a regular expression was being done frequently enough that it became encapsulated in a program. That program's name summarized the operation: g/re/p

The grep family

There are three classic members in the grep family: grep, fgrep, and egrep.

The need for three distinct programs was driven by performance and memory limitations at the time they were created.

Here's a quote from the 4BSD manual, c. 1980:

> *"grep patterns are limited regular expressions in the style of ex(1) [an improved ed]; it uses a compact nondeterministic algorithm. egrep patterns are full regular expressions; it uses a fast deterministic algorithm that sometimes needs exponential space. fgrep patterns are fixed strings; it is fast and compact.*

Today:

> grep is a good first choice and is the most frequently used member of the family.
>
> egrep makes complex regular expressions less tedious due to the enlarged set of metacharacters.
>
> fgrep treats no characters as special—use it when the pattern contains lots of special characters that you don't want treated specially. Example: fgrep '*.*' *.sh

Other grep variants exist. (Try man -k grep.) One of them is agrep—"approximate" grep. Try agrep -2 Willim /etc/passwd to find lines that are within two characters of matching "Willim".

Emacs' grep command

Emacs has a grep command that runs the grep program and lets you work through "hits" one by one, just like compilation errors.

M-x grep produces a partial command line in the minibuffer:

Run grep (like this): grep -n -e

Finish the command line:

Run grep (like this): grep -n -e **println *.java**

A buffer named *grep* will appear that shows the matching lines.

Typing C-x ` (back quote!) runs next-error, which for each hit, opens the file and positions you on the matching line.

Emacs and regular expressions

Emacs supports regular expressions. All the details can be found via info: Choose the Emacs menu item and then the Regexps menu item.

M-C-s starts an incremental regular expression search.

Recall this example: Consider the steps to change lines from this:

```
/home/whm/x.java
/x/y.z
/a/bb/ccc/dddd/eeee
```

to this:

```
x.java   /home/whm
y.z      /x
eeee     /a/bb/ccc/dddd
```

C-M-% runs query-replace-regexp. The replacement pattern can use \1, \2, etc. to "plug in" groups from the matched string.

Here's an RE to search with:

```
^\(.*\)/\(.*\)$
```

and a replacement

```
\2 TAB \1
```

Files and File Management—Part 2

User and groups

File permissions

Directory permissions

root—the "superuser"

Symbolic links and permissions

umask

The set-uid mechanism

"Everything is a file"

Users and groups

Associated with every login name, like whm, there is a numeric user id (*uid*).

A user's uid is specified in the third field of the user's /etc/passwd entry:

```
$ egrep "whm|ranjini" /etc/passwd
whm:x:3186:46:William H. Mitchell:...
ranjini:x:2349:7449:Ranjini Swaminathan:...
```

A user can display his or her user id with the id command:

```
$ id -u
3186
```

Internally, UNIX uses the uid (as an integer), not the login name, to identify users.

Users and groups, continued

UNIX has a notion of *group membership*. A user may be in many groups at once. /etc/group specifies members of groups. Here are a few selected lines of a representative /etc/group file: (not lectura's)

```
staff:*:15:root,steve,joff,alt,mca,tas
other:*:17:
recruit:*:22:recruit,peter,mp
faculty:*:26:alt,robert,ed,lionel,alan,mca
office:*:24:beth,maggie,jmc
turnin:*:37:cs227,cs340,cs352,cs372,alt,robert,\
            ed,lionel,alan,mca,ak,andrew
```

Every user is in at least one group, specified in the fourth /etc/passwd field as a numeric group id (gid).

The groups command shows group memberships. id shows full details:

```
$ groups
dept turnin cs352 cs451 whm

$ id
uid=3186(whm) gid=46(dept) groups=46(dept),
37(turnin), 119(cs352),451(cs451),8086(whm)
```

When a user runs a process, the uid and gid of that user become the uid and gid of that process.

File permissions

When a user attempts to perform an operation on a file, the UNIX kernel determines if the user has permission for that operation on that file.

Every file is owned by a user and a group. The -l (L) option of ls shows the ownership. The -n flag suppresses look-up of the uid and gid; the numeric values are displayed instead.

```
$ cd ~cs352/spring05/etc
$ ls -l .bashrc
-rw-r--r-- 1 whm   cs352    395 Jan 24 21:27 .bashrc

$ ls -ln .bashrc
-rw-r--r-- 1 3186  119      395 Jan 24 21:27 .bashrc
```

Every file has a collection of nine "mode bits" (or, "the mode") that specify which operations can be performed by whom. The characters in the leftmost column of ls -l output show the permissions.

File permissions, continued

At hand:

```
$ ls -l .bashrc
-rw-rw-r-- 1 whm   cs352    395 Jan 24 21:27 .bashrc
```

The first character is the type of the file and is not related to permissions. (A dash is an ordinary file, `d` is directory, `l` is symbolic link; there are others that are less common.)

The next three characters (`rw-`) indicate the operations that can be performed by the owner of the file (whm): he can read data from it and write data to it.

If the file were executable, the third character would be x, not -.

The next three characters specify the operations permitted by group members. The file can be read and written, but not executed by members of the cs352 group.

The final group of three characters specifies what "others" can do. If a user is not whm, and not in the cs352 group, the only operation that can be performed is reading the file.

File permissions, continued

Running the command 'cat .bashrc' eventually reaches a line of code in cat.c that calls a kernel routine that in turn opens the file for reading:

```
/* Code from GNU textutils-2.0/src/cat.c */

int input_desc;
int file_open_mode = O_RDONLY;
...
input_desc = open (infile, file_open_mode);
```

When a program attempts to open a file, the kernel determines the relationship of the user to the file. If whm is running the program then the kernel consults the owner permission bits.

Because the owner permission bit to allow read access is "on", the file is opened successfully.

Here is the access validation algorithm in broad terms:

> Based on the uid and gid of the process, classify the user as owner of the file, group member, or "other".
>
> Classify operation as read, write, or execute.
>
> Check the mode bit associated with the user and operation classification.

File permissions, continued

A file might be writable but not readable:

```
$ ls -l log
--w--w--w-   1 whm  dept   55978 Sep 15 00:52 log

$ cat log
cat: log: Permission denied

$ date >>log

$ ls -l log
--w--w--w-   1 whm  dept   56007 Sep 15 00:53 log
```

A more practical example:

```
$ ls -l log
-rw--w--w-   1 whm  dept   56007 Sep 15 00:53 log
```

How might a mode 611 file be useful?

File permissions, continued

The third character in each mode triple specifies whether the file is executable by each class of user.

```
$ ls -l hello*
-rw-r--r--   1 whm   dept  21 Sep 15 00:48 hello1
-rwxr-xr-x   1 whm   dept  21 Sep 15 00:49 hello2

$ head hello*
==> hello1 <==
echo "Hello, world!"

==> hello2 <==
echo "Hello, world!"

$ ./hello1
bash: ./hello1: Permission denied

$ ./hello2
Hello, world!
```

The only difference between hello1 and hello2 is that hello2 has the mode bits for execute "on". hello2 is said to be *executable*.

A file can be unreadable but still executable:

```
$ ls -l date
---x--x--x   1 whm  dept  80608 Sep 15 00:39 date
$ wc date
wc: date: Permission denied
$ ./date
Wed Sep 15 00:50:49 MST 2004
```

What are the pros and cons of having an "executable" bit?

File permissions, continued

The chmod command is used to change the mode of a file.

One alternative is to specify the mode in octal (base 8)—each digit corresponds to a character triple:

```
$ chmod 742 x
$ ls -l x
-rwxr---w-  1 whm    dept       0 Sep 15 01:00 x
```

Interpretation:

```
7 = 111     rwx
4 = 100     r--
2 = 010     -w-
```

Remember: The leading dash is the file type.

Problems:

What octal digits correspond to `r-x--xrw-` ?

What character triples correspond to the octal digits 123?

File permissions, continued

The mode can also be specified symbolically:

```
$ chmod 0 A      (Note that the name of the file is "A")
$ ls -l A
----------   1 whm   dept        0 Sep 15 01:00 A

$ chmod u+r A   ('u' is the (u)ser who owns the file)
$ ls -l A
-r--------   1 whm   dept        0 Sep 15 01:00 A

$ chmod o+w A   ('o' is "others"—not the user or the group)
$ ls -l A
-r------w-   1 whm   dept        0 Sep 15 01:00 A

$ chmod g+rw A
$ ls -l A
-r--rw--w-   1 whm   dept        0 Sep 15 01:00 A

$ chmod ug-w A   ('-' turns off bits)
$ ls -l A
-r--r---w-   1 whm   dept        0 Sep 15 01:00 A

$ chmod 644 A
$ ls -l A
-rw-r--r--   1 whm   dept        0 Sep 15 01:00 A

$ chmod +x A    (if no class(es) specified, apply to all)
$ ls -l A
-rwxr-xr-x   1 whm   dept        0 Sep 15 01:00 A

$ chmod u=rw,g=wx,o=r A     (OUCH!  Note: no spaces!)
$ ls -l A
-rw--wxr--   1 whm   dept        0 Sep 15 01:00 A
```

File permissions, continued

Many files may be specified at once:

```
chmod +x *.sh

chmod 444 $(find -name "*.java")
```

The -R option recursively applies a change to every file (and directory) in a tree:

```
chmod -R -w src
```

Different programs handle read-only files in different ways. For example, rm prompts if a read-only file is specified:

```
$ touch a b
$ chmod 444 a
$ ls -l
-r--r--r--   1 whm    dept    0 Sep 16 02:42 a
-rw-r--r--   1 whm    dept    0 Sep 16 02:42 b
$ rm *
rm: remove write-protected file `a'? y
$ ls
$
```

cp doesn't allow a read-only file to be overwritten, unless -f is specified:

```
$ touch x y; chmod 444 y
$ cp x y
cp: cannot create regular file `y': Permission
denied
$ cp -f x y
$ ls -l y
-rw-r--r--   1 whm     dept    0 Sep 16 02:48 y
```

File permissions, continued

Problem: A naive system administrator has "disabled" a too-popular game:

```
$ /usr/games/zork
bash: /usr/games/zork: Permission denied

$ ls -l /usr/games/zork
-rwxr--r--   1 root  dept    703908 Sep 15 02:26
/usr/games/zork
```

How can be Zorking be continued?

Directory permissions

Permissions apply to directories, too, but with a different interpretation:

> Read permission allows the directory entries to be listed.
>
> Write permission allows entries to be added or deleted.
>
> Execute permission allows a directory to be searched and/or cd'd into.

The instructor's home directory is mode 700:

```
$ ls -ld ~whm
drwx------ 88 whm empty 32768 Feb  8 09:38 /home/whm
```

If a directory is unwritable, files can't be removed or created, but existing files can be changed:

```
$ touch x y
$ ls -l
-rw-r--r--   1 whm   dept        0 Sep 15 01:25 x
-rw-r--r--   1 whm   dept        0 Sep 15 01:25 y
$ chmod u-w .
$ ls -ld .
dr-xr-xr-x   2 whm   dept     4096 Sep 15 01:25 .
$ rm x
rm: cannot unlink `x': Permission denied
$ touch z
touch: z: Permission denied
$ date >y
$ ls -l
-rw-r--r--   1 whm   dept        0 Sep 15 01:25 x
-rw-r--r--   1 whm   dept       29 Sep 15 01:25 y
```

Directory permissions, continued

The contents of an unreadable directory cannot be listed:

```
$ mkdir d1
$ date >d1/x
$ touch d1/y
$ ls -l
drwxr-xr-x   2 whm   dept    4096 Sep 15 01:47 d1
$ ls -l d1
-rw-r--r--   1 whm   dept      29 Sep 15 01:47 x
-rw-r--r--   1 whm   dept       0 Sep 15 01:47 y
$ chmod 0 d1
$ ls -l
d---------   2 whm   dept    4096 Sep 15 01:47 d1
$ ls d1
ls: d1: Permission denied
```

If a directory has only search permission (`--x`) files in the directory can be accessed by name, but the directory contents can't be listed:

```
$ chmod 111 d1
$ ls -ld d1
d--x--x--x   2 whm   dept    4096 Sep 15 01:47 d1
$ ls d1
ls: d1: Permission denied
$ ls -l d1
ls: d1: Permission denied
$ cat d1/x
Wed Sep 15 01:47:33 MST 2004
$ date >d1/y
$ touch d1/z
touch: d1/z: Permission denied
```

Directory permissions, continued

For reference: d1 has two files:

```
-rw-r--r--   1 whm   dept       29 Sep 15 01:47 x
-rw-r--r--   1 whm   dept       29 Sep 15 01:47 y
```

You can cd to a search-only directory and modify existing files (assuming appropriate permission on the files) but you can't look around or change directory entries.

```
$ ls -ld d1
d--x--x--x   2 whm   dept    4096 Sep 15 01:47 d1
$ cd d1
$ ls
ls: .: Permission denied
$ cat x
Wed Sep 15 01:49:20 MST 2004
$ cp x y
$ touch z
touch: z: Permission denied
```

Search-only directories are typically used to provide users access to files, but only if the user has been informed of the name(s). Analogy: A web page that's not referenced anywhere but that can be displayed if the URL is known.

Is there a way to determine the names of files in a search-only directory?

Speculate: What are the characteristics of a mode 444 directory?

root—the "superuser"

Uid 0 is the user id of root, the "superuser". System administration work is typically done as root. When running as root, read and write permissions are simply ignored—root has full access to every file in every directory.

Example: (Note that by convention, root's prompt contains a #.)

```
# id -u
0

# ls -l x
----------   1 whm   dept       29 Sep 15 03:11 x

# cat x
Wed Sep 15 03:11:28 MST 2004

# cal >x

$ ls -l x
----------   1 whm   dept      139 Sep 15 03:11 x
```

Execute permissions do apply to root:

```
# ./x
bash: ./x: Permission denied
```

Symbolic links and permissions

Symbolic links are shown as mode 777, but the mode is inconsequential because read and write operations apply to the underlying file, not the link.

```
$ ls -l
total 0
-r--r--r--  1 whm  dept   29 Sep 15 03:17 x
lrwxrwxrwx  1 whm  dept    1 Sep 15 03:17 x1 -> x

$ date >x
bash: x: Permission denied

$ date >x1
bash: x1: Permission denied

$ chmod 666 x1
$ ls -l
-rw-rw-rw-  1 whm  dept   29 Sep 15 03:17 x
lrwxrwxrwx  1 whm  dept    1 Sep 15 03:18 x1 -> x

$ cal > x1
$ ls -l
-rw-rw-rw-  1 whm  dept  139 Sep 15 03:21 x
lrwxrwxrwx  1 whm  dept    1 Sep 15 03:18 x1 -> x
```

The ability to delete or create a symbolic link is determined by the permissions of the directory containing (or to contain) the link, just as if deleting or creating an ordinary file.

umask

Every process has a *file mode creation mask*. The umask command displays the current value of the mask, which is inherited by child processes:

```
$ umask
022
```

touch and bash's > operator, to name two examples, specify an initial mode of 666 (`rw-rw-rw-`) for files that are created. The GNU C compiler specifies an initial mode of 777 for executable files it creates.

The initial mode of a file is calculated by starting with the mode specified by the creating program and turning off the umask'd bits.

```
$ umask
022
$ touch x
$ gcc -o trivial trivial.c
$ ls -l
-rwxr-xr-x  1 whm  dept  6376 Sep 15 23:09 trivial
-rw-r--r--  1 whm  dept     9 Sep 15 23:09 trivial.c
-rw-r--r--  1 whm  dept     0 Sep 15 23:09 x
```

The same sequence, but with a more restrictive umask:

```
$ umask 077
$ rm x trivial
$ touch x
$ gcc -o trivial trivial.c
$ ls -l
-rwx------  1 whm  dept  6376 Sep 15 23:10 trivial
-rw-r--r--  1 whm  dept     9 Sep 15 23:09 trivial.c
-rw-------  1 whm  dept     0 Sep 15 23:10 x
```

The "set-uid" mechanism

In some cases it is useful to allow ordinary users to read and/or write files they do not have permission to read or write.

Example: Any user can change his or her password using passwd(1). However, that requires modifications to /etc/shadow:

```
$ ls -l /etc/shadow
-r--------  1 root 89554 Sep 15 21:20 /etc/shadow
```

The set-uid (SUID) mechanism provides a solution:

> If an executable has mode bit 4000 set, when that executable is run the effective user id (euid) of the process becomes the uid of the owner of the file.

Here is what a set-uid executable looks like: (the mode is 4555)

```
$ ls -l /usr/bin/passwd
-r-sr-xr-x   1 root  sys   21964 Apr  6  2002
/usr/bin/passwd
```

Note the `s` instead of an `x` in the owner triple. When passwd is run, that process has the file access privileges of root, and can make changes to /etc/shadow.

Similarly, mode bit 2000, the set-gid (SGID) bit causes the effective group id of a process to be set based on the executable's group.

It is notoriously difficult to write a perfectly secure non-trivial set-uid application.

The set-uid mechanism was awarded a patent.

"Everything is a file"

It is sometimes said that with UNIX, "everything is a file": UNIX creates a facade that allows peripherals and "pseudo-devices" to be treated as files.

Once upon a time, "dumb terminals" were connected to UNIX machines via serial lines. The terminals appeared as files in the /dev directory. At that time, you might have seen this:

```
$ ls -l /dev/tty?
crw--w--w- 1 root      1,  0  Oct  1 10:10 ttya
crw--w--w- 1 jte       1,  1  Oct  2 09:23 ttyb
crw--w--w- 1 mp        1,  2  Oct  2 10:14 ttyc
crw--w--w- 1 tas       1,  3  Oct  2 10:09 ttyd
```

You could do something like this,

```
$ echo "All files deleted." > /dev/ttyb
```

and jte would see those words on his screen.

You could then do this,

```
$ stty erase " " >/dev/ttyb
```

which would cause the spacebar to act like the backspace key, and further test your friendship with jte.

"Everything is a file"

Hardwired terminals are less common these days but there are "files" in /dev that correspond to "pseudo-terminals" that are used for network logins. The tty (teletype) command shows the name of the terminal that one is using:

```
$ tty
/dev/pts/149
$ ls -lL /dev/pts/149
crw--w----   1 whm   tty   24, 149 Sep 15 23:44
/dev/pts/149
```

If output is redirected to the device, it appears on the screen:

```
$ date > /dev/pts/149
Wed Sep 15 23:42:30 MST 2004
```

The device can be read:

```
$ wc /dev/pts/149
a
test
^D
      2       2       7 /dev/pts/149
```

"Everything is a file", continued

There are devices corresponding to peripherals. Here are some examples from Red Hat Linux:

/dev/sda is the entirety of the first SCSI disk.

/dev/sdb2 is the second partition of the second SCSI disk.

/dev/fd0 is the first floppy disk drive.

/dev/lp1 is the second parallel-port printer.

Devices on Solaris, the flavor of UNIX that lectura runs, aren't named as simply. For example, the root filesystem is on /dev/dsk/c1t0d0s0, which is a symbolic link to

/devices/pci@8,600000/SUNW,qlc@4/fp@0,0/ssd@w21000004cf967393,0:a

"Everything is a file", continued

Here are some devices (pseudo-devices, actually) that are handy for the ordinary user:

/dev/null	The null device. Data written to it is discarded. If read, it produces end of file immediately.
/dev/random	A source of random data. (man -s 7d random)
/dev/tty	The terminal you're currently on.
/dev/zero	An infinite source of zero-valued bytes.

To discard some output, send it to /dev/null. Imagine you want to see only standard error output from a command that writes thousands of lines to standard output:

```
$ ls -lR /home/icon >/dev/null
ls: /home/icon/bugs/tc: Permission denied
ls: /home/icon/bugs/tc2: Permission denied
$
```

Imagine a program insists on error log file being written. If you don't care about the log, you might do this:

```
$ app2 -log /dev/null ...
```

A file of zero-valued bytes can be made by reading a portion of /dev/zero:

```
$ head -100000c /dev/zero > zeroes
```

"Everything is a file", continued

One use of /dev/tty is to read from the keyboard regardless of whether standard input is redirected. Example:

```
import java.io.*;
public class oklines {
   public static void main(String args[]) throws IOException {
     BufferedReader in =
         new BufferedReader(new
                                  InputStreamReader(System.in));
     BufferedReader tty =
         new BufferedReader(new FileReader("/dev/tty"));

      String line;
      while ((line = in.readLine()) != null) {
         System.err.print(line + "? ");  // Note: System.err
         if (tty.readLine().charAt(0) == 'y')
            System.out.println(line);
         }
      }
  }
```

Usage:

```
$ mkdir new
$ cp $(ls *.java | java oklines) new
Test.java? n
env1.java? y
ruler.java? y
x.java? n
$ ls new
env1.java  ruler.java
```