CSE 380: Lecture 2
Unix is fun

Instructors:

Dr. Chris Simmons, ICES

csim@ices.utexas.edu

Unix in Practice

- Q: On the June 2010 Top500 list, how many systems ran Unix / Linux?
- A: at least 95% are UNIX-like, 1% windows

Operating system Family	Count	Share %	Rmax Sum (GF)	Rpeak Sum (GF)	Processor Sum
Linux	455	91.00 %	27162011	41989385	3774451
Windows	5	1.00 %	412590	509350	59072
Unix	22	4.40 %	1479895	1891787	118930
BSD Based	1	0.20 %	122400	131072	1280
Mixed	17	3.40 %	3257787	3948902	1177728
Totals	**500**	**100%**	**32434683.70**	**48470495.53**	**5131461**

Unix Background

- Q: How old is Unix?
- A: > 40 Years
  - Unix originally dates back to 1969 with a group at Bell Laboratories
  - The original Unix operating system was written in assembler
  - In 1973 Thompson and Ritchie finally succeeded in rewriting Unix in their new language, C.
  - Most system programming was done in assembler
  - The very concept of a *portable* operating system was unheard of
  - First Unix installations in 1972 had 3 users and a 500KB disk

DEC PDP-11, 1972

What is UNIX?

- UNIX is a multi user, preemptive, multitasking operating system which provides a number of facilities:
  - management of hardware resources
  - directories and file systems
  - loading / execution / suspension of programs
- What does UNIX stand for?
  - Nothing actually - It is a "play on words" of an older multiuser time-sharing OS known as Multics
- There are many flavors of UNIX:
  - Solaris (Sun)
  - AIX (IBM)
  - Tru64 (Compaq)
  - IRIX (SGI)
  - SysV (from AT&T)
  - BSD (from Berkeley)
  - Linux (its not UNIX, but it's close enough from our point of view)

What is Linux?

- Linux is a clone of the Unix operating system written from scratch by Linus Torvalds with assistance from developers around the globe
- Technically speaking, Linux is not UNIX
- Torvalds uploaded the first version of Linux in September 1991
- Only about 2% of the current Linux kernel is written by Torvalds himself
- He remains the ultimate authority on what new code is incorporated into the kernel
- Developed under the GNU General Public License , the source code for Linux is freely available
- Download latest kernels from www.kernel.org
- A large number of Linux-based distributions exist (for free or purchase):
  - RedHat, Fedora, CentOS
  - SUSE
  - Debian
  - Gentoo
  - Slackware
  - Ubuntu
  - Mandrake
  - Mint

Why use UNIX?

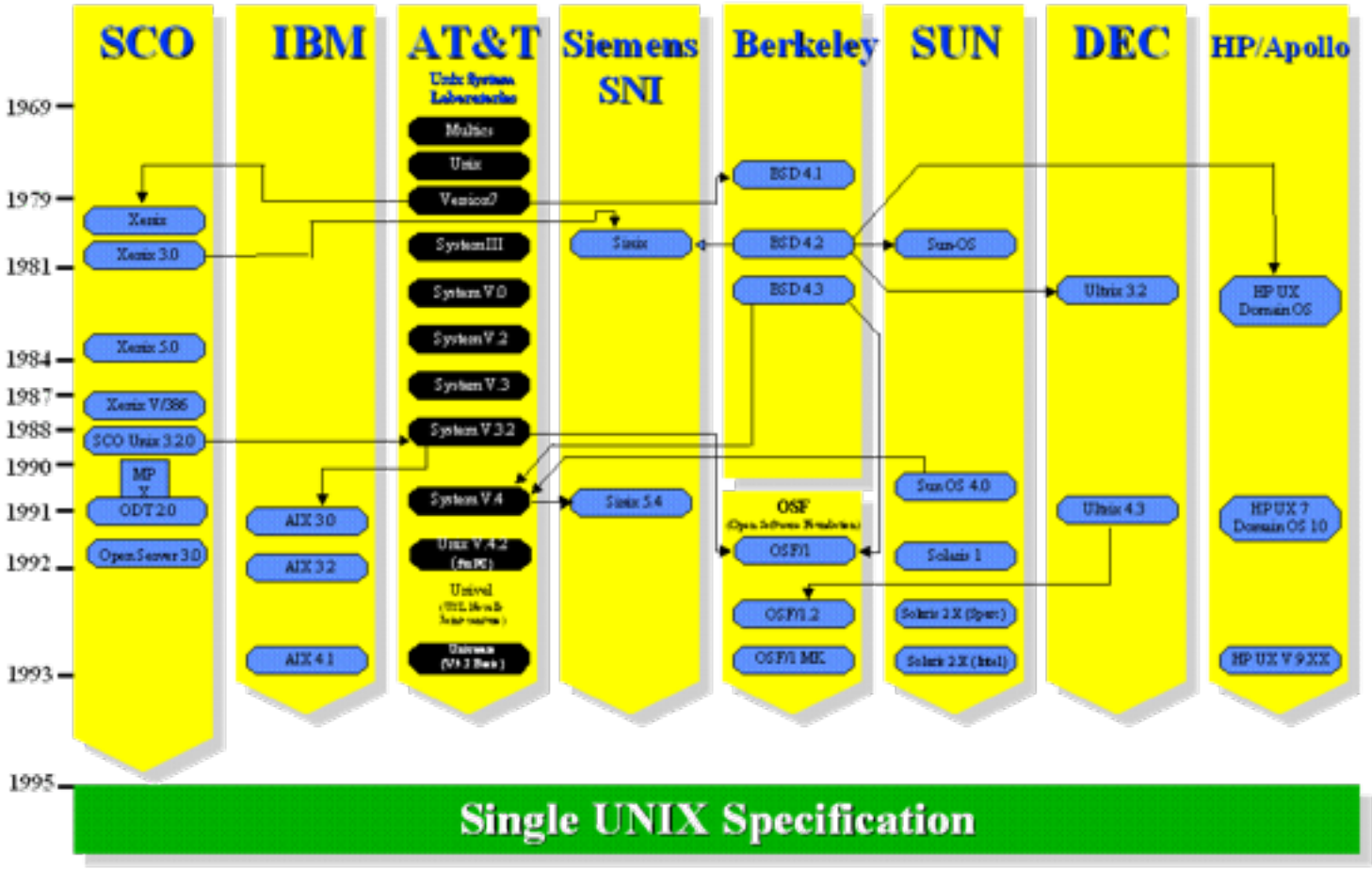
- **Performance:** as we've seen, supercomputers generally run UNIX; rich-multi user environment
- **Functionality:** a number of community driven scientific applications and libraries are developed under UNIX (molecular dynamics, linear algebra, fast-fourier transforms, etc).
- **Flexibility/Portability:** UNIX lets you build your own applications and there is a wide array of support tools (compilers, scientific libraries, debuggers, network monitoring, etc.)

Some Key People

Ken Thompson and Dennis Ritchie
Your new heroes.

????
Linus Torvalds

Unix Background: Chronology

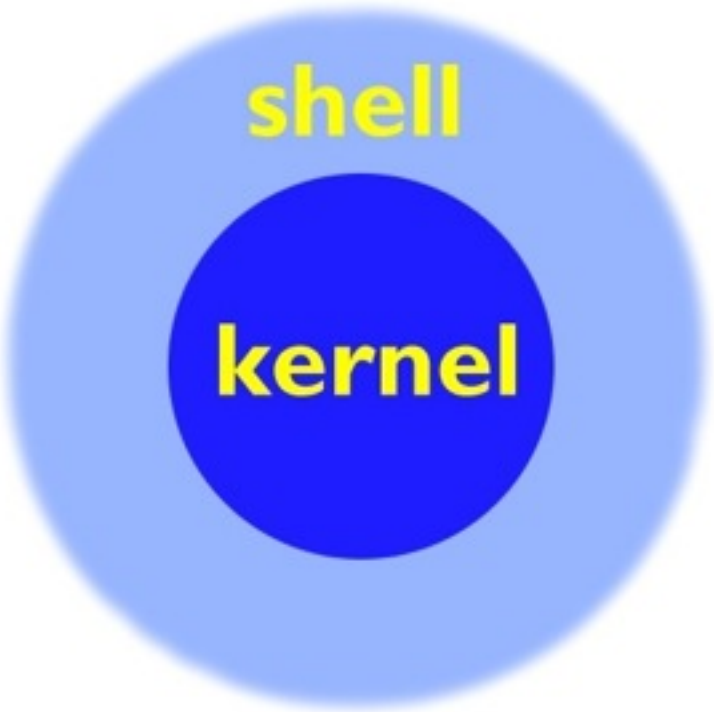


The Single UNIX Specification is the collective name of a family of standards for computer operating systems to qualify for the name "Unix“ (eg. HP-UX, IBM AIX, SGI IRIX, Sun Solaris).

Source: The Open Group, www.unix.org

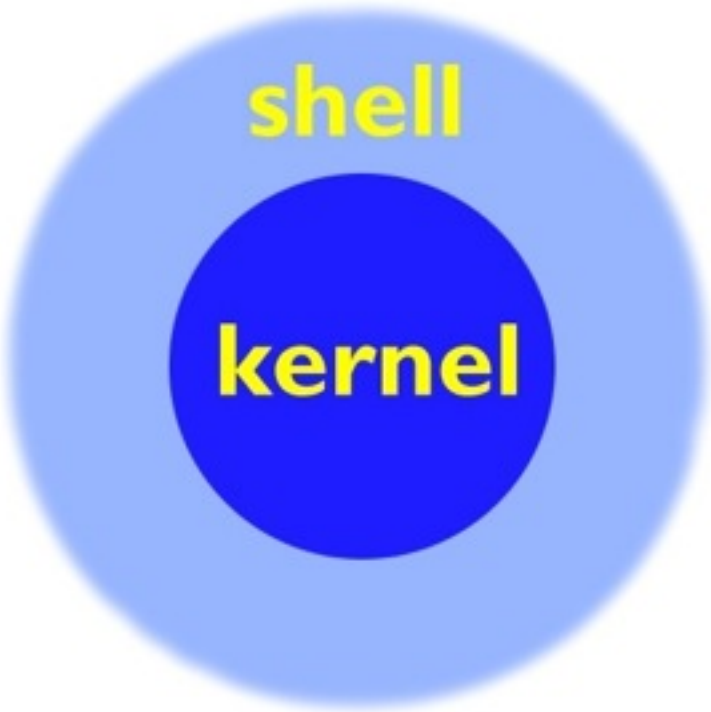
How does UNIX work?

- UNIX has a kernel and one or more shells
- The kernel is the core of the OS
- It receives tasks from the shell and performs them
- Users interact with the shell



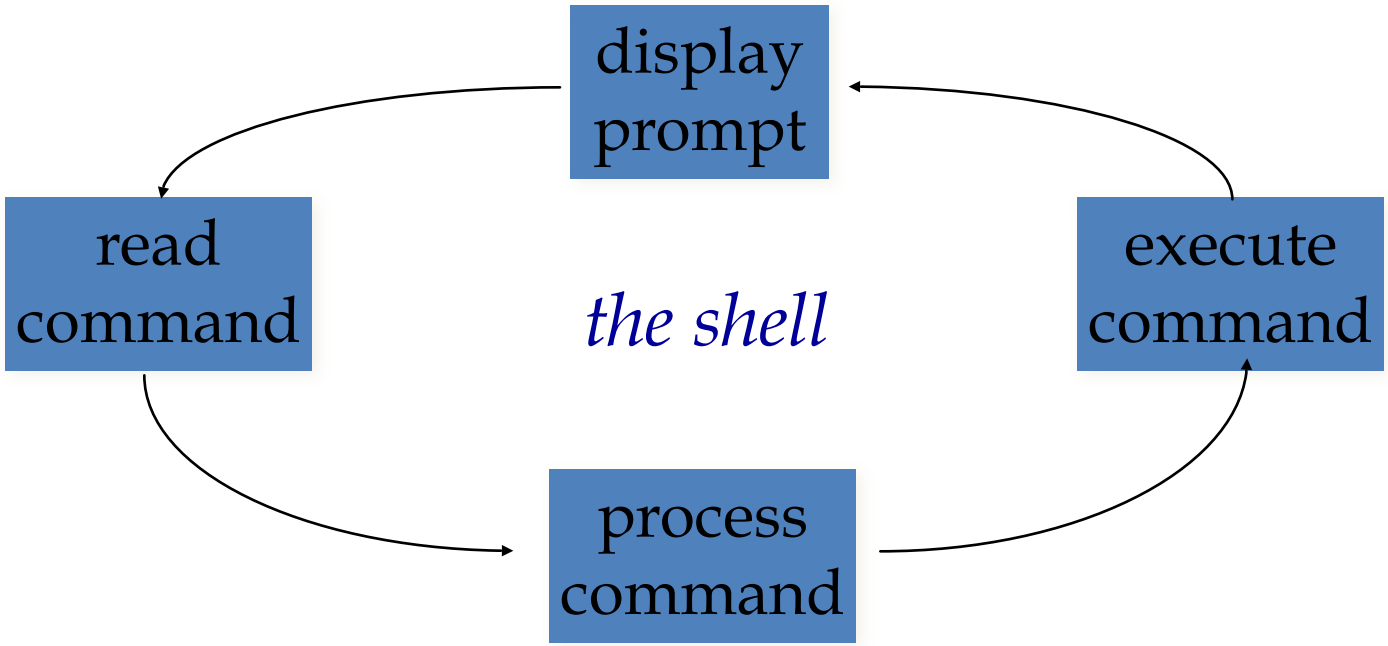
How does UNIX work?

- Everything in UNIX is either a *file* or a *process*
- A process …
  - is an executing program identified by a unique PID (process identifier).
  - may be short in duration or run indefinitely
- A file is …
  - a collection of data.
  - created by users using text editors, running compilers, etc
- The UNIX kernel is responsible for organizing processes and interacting with files



What does the Shell Do?

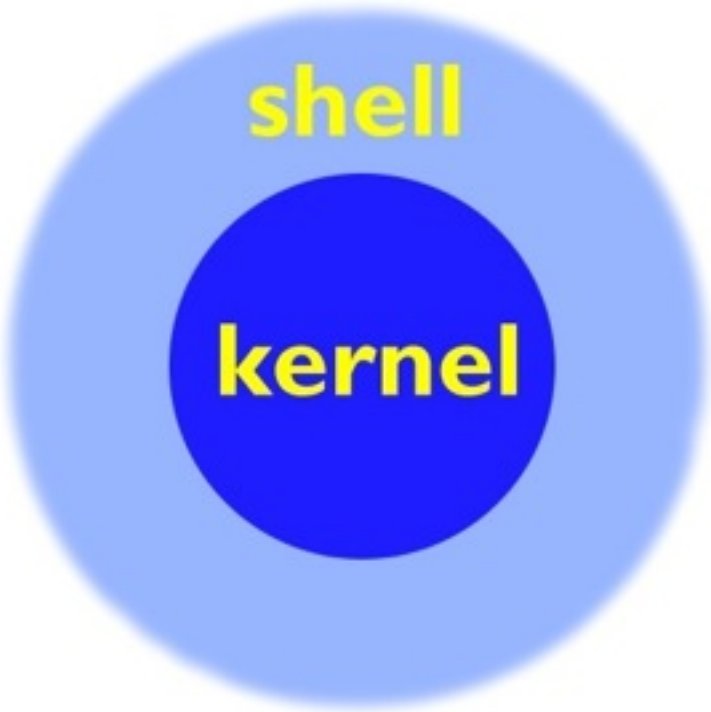
- The UNIX user interface is called the *shell*
- The shell tends to do 4 jobs repeatedly:



An Example

Example: A user wants to remove a file:

- User has a command-line prompt (the shell is waiting for instructions)
- User types the command (rm myfile) in the shell
- The shell searches the filesystem for the file containing the program (rm)
- A new process is forked from the shell to run the command with an instruction to remove myfile
- The process requests that the kernel, through system calls, delete the reference to myfile in the filesystem
- When the rm process is complete, the shell then returns to the UNIX prompt indicating that it is waiting for further commands
- The process ID (PID) originally assigned to the rm command is no longer active

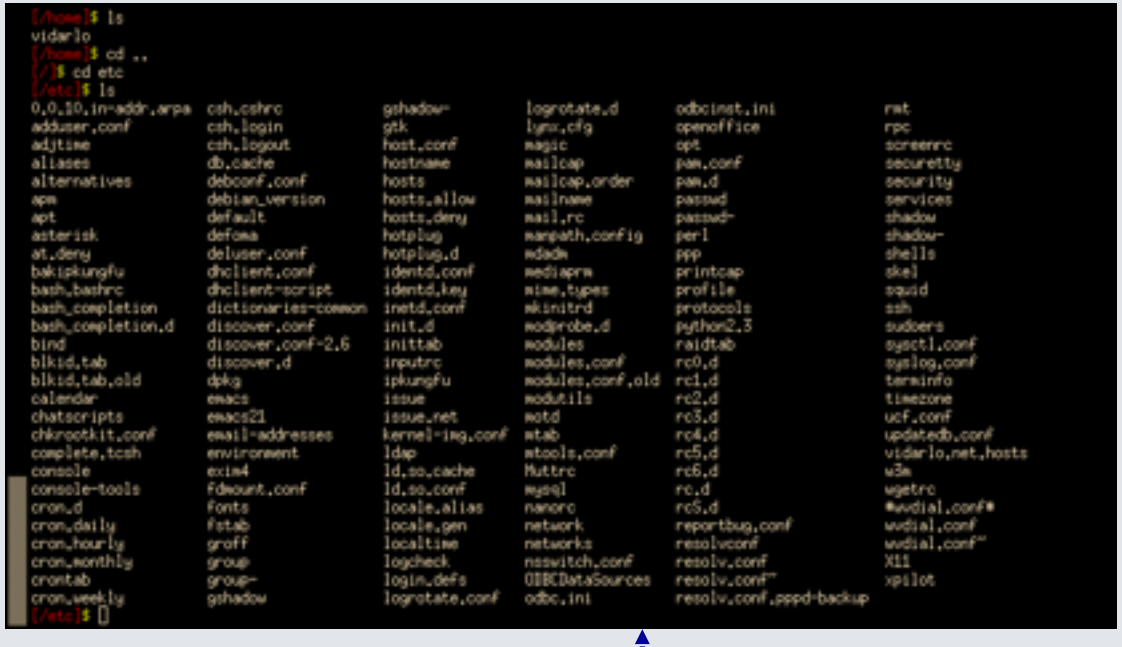


Unix Interaction

- The user interacts with UNIX via a shell
- The shell can be graphical (X-Windows) or text-based (command-line) shells like tcsh and bash
- To remotely access a shell session, use ssh (secure shell)
- ssh is a secure replacement for telnet

X-Windows and Unix

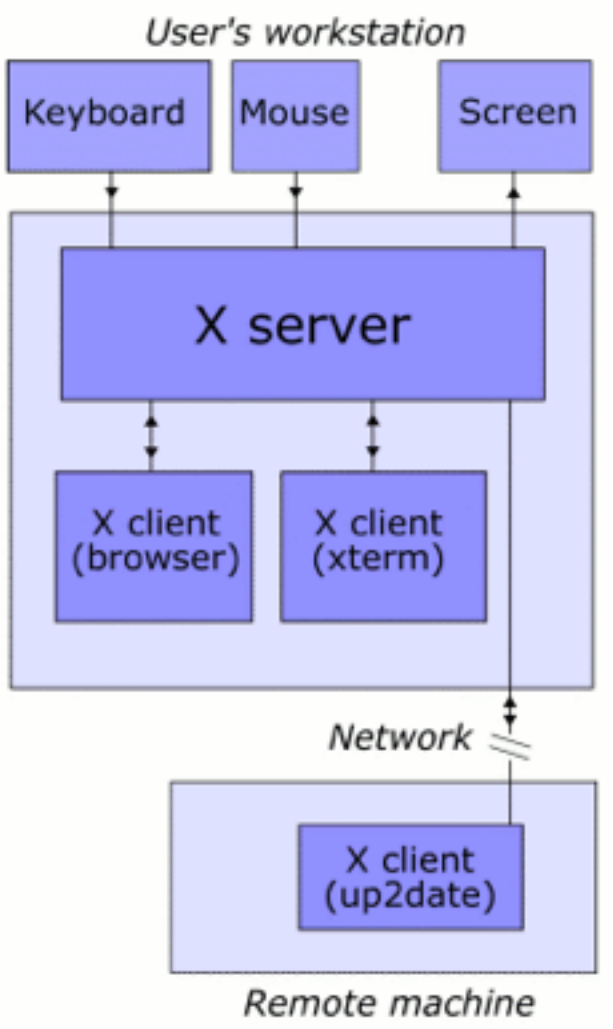
- X-Windows is the standard graphical layer for UNIX systems
- Most graphical interfaces for UNIX are actually built on top of X-Windows
- Fundamental command-line application in X-windows is an *xterm*



- A user can have many different invocations of xterm running at once on the same display, each of which provides independent input/output for the process running in it (normally the process is a Unix shell)

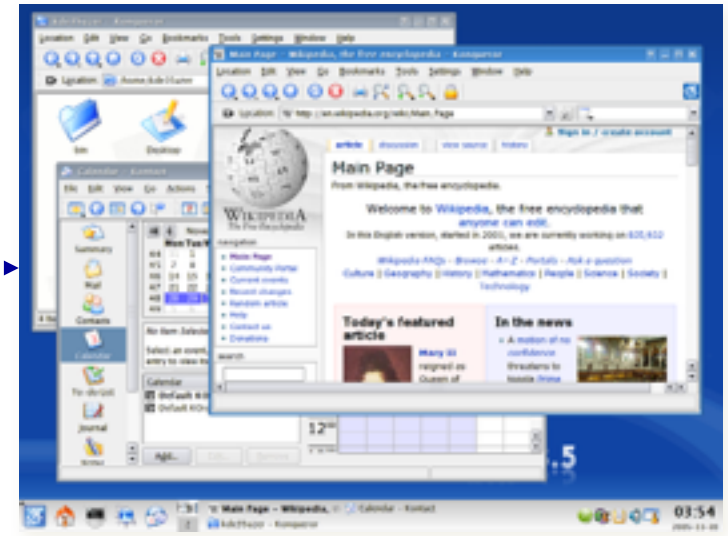
X-Windows

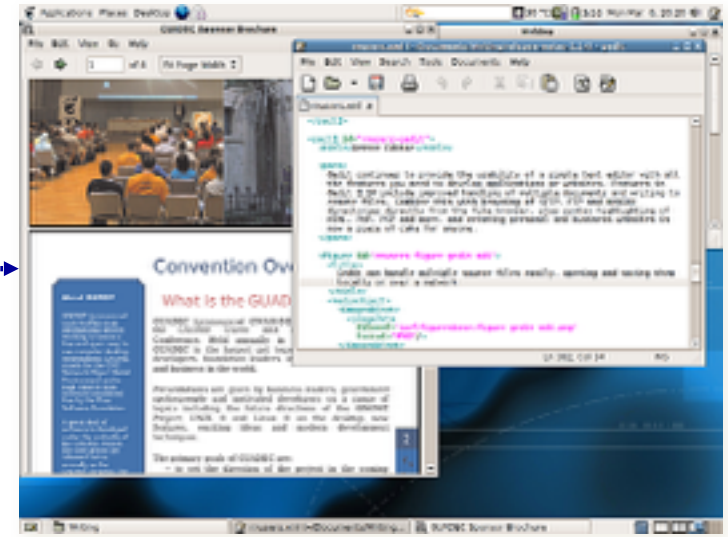
- The original idea of X emerged at MIT in 1984
- It provides a standard toolkit and protocol to build graphical user interfaces (GUI) on Unix, or Unix-like operating systems
- X supports remote connectivity
- The computer where application programs (the *client* applications) run can differ from the user's local machine (the display *server*).
- X's usage of the terms "client" and "server" reverses what people often expect, in that "server" refers to the user's local display ("display server") rather than to a remote machine.



X-Windows and Unix

- Several nice desktop environments exist for Linux
  - KDE
  - Gnome
- Cygwin for Windows also includes an Xserver and xterm client
- X.Org is a freely redistributable open-source implementation of the X Window System (http://www.x.org/)





Unix Accounts

- To access a Unix system you need to have an *account*
- Unix account includes:
  - username and password
  - userid and groupid
  - home directory
    - a place to keep all your snazzy files
    - may have a quota (system imposed limit on how much data you can have)
  - a default shell preference

Unix Accounts

- A username is a sequence of alphanumeric characters
  - *eg. csim or* karl
- The username is the primary identifying attribute of your account
- the name of your home directory is usually related to your username:
  - *eg, /home1/00416/csim*

Unix Accounts

- A password is a secret string that only the user knows
- Not even the system knows a user's password
- When you enter your password the system encrypts it and compares to a stored string
- It's a good idea to include numbers and/or special characters (don't use an english word, as this is easy to crack)

Unix Accounts

- A *userid* is a number (an integer) that identifies a Unix account.
- Each userid must be unique
- In Unix-speak, userids are known as *UIDs*
- Why does Unix implement UIDs? It's easier (and more efficient) for the system to use a number than a string like the username
- You don't necessarily need to know your userid

Unix Accounts

- Unix includes the notion of a "group" of users
- A Unix group can share files and active processes
- Each account is assigned a "primary" group
- The *groupid* is a number that corresponds to this primary group
- In Unix-speak, groupids are knows as *GIDs*
- A single account can belong to many groups (but has only one primary group)

Files and File Names

- A file is a basic unit of storage (usually on a disk)
- Every file has a name
- Unix file names can contain any characters
- Some characters make it hard to access the file
- Unix file names can be long!
  - how long depends on your specific flavor of Unix

File Contents

- Each file can hold some raw data
- Unix does not impose any structure on files
  - files can hold any sequence of bytes
  - it is up to the application or user to interpret the files correctly
- Many programs interpret the contents of a file as having some special structure
  - text file, sequence of integers, database records, etc.
  - in scientific computing, we often use binary files for efficiency in storage and data access
    - Fortran unformatted files
    - Scientific data formats like NetCDF or HDF have specific formats and provide APIs for reading and writing
    - Portability is an issue with some formats (*little endian vs. big endian)*

Directories

- A directory is a special kind of file
- Unix uses a directory to hold information about other files
- We often think of a directory as a container that holds other files (or directories)
- Mac and Windows users can relate a *directory* to the same idea as a *folder*

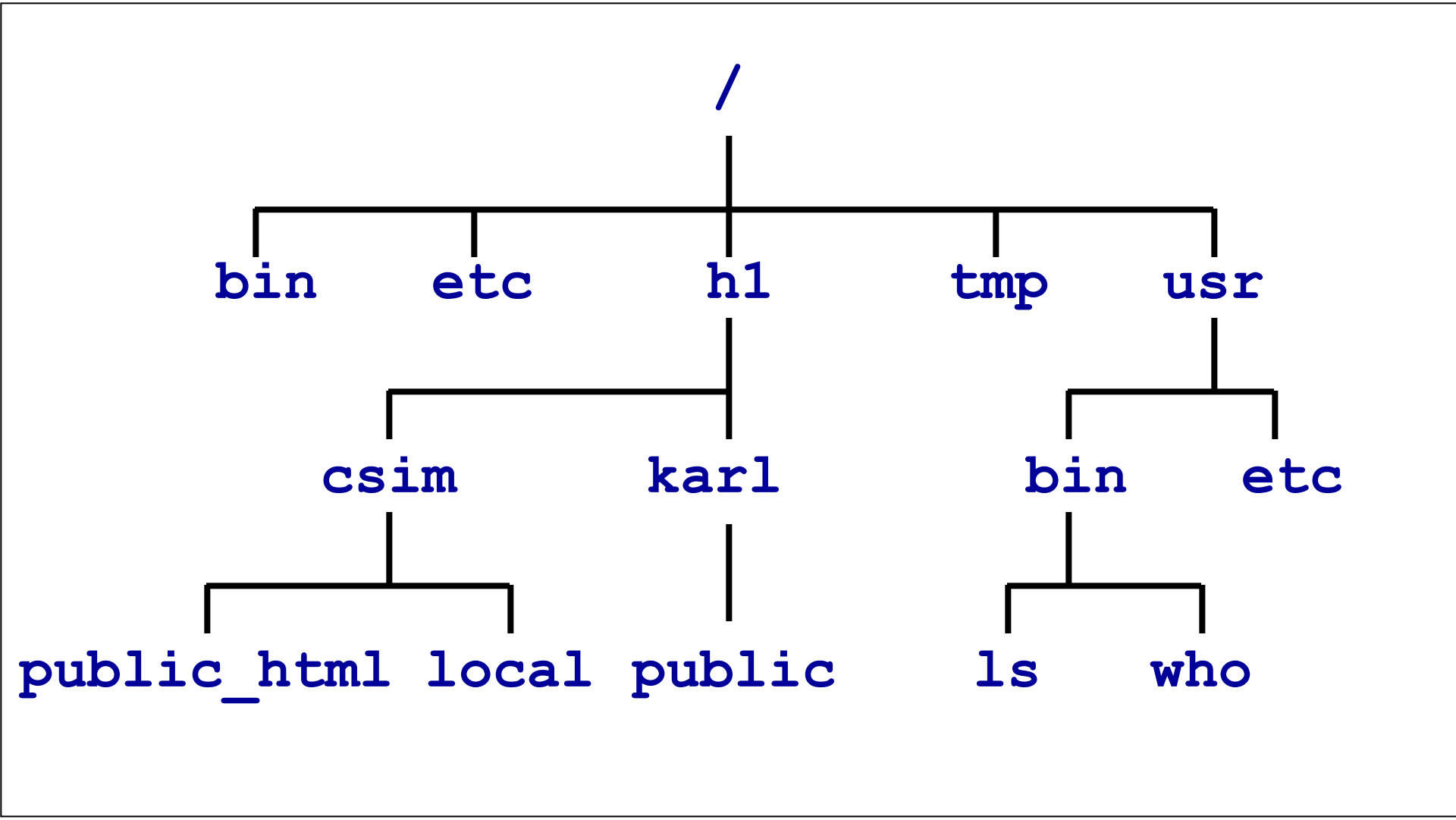
More about File Names

- Every file must have a name
- Each file in the same directory must have a unique name
- Files that are in different directories can have the same name
- Note: Unix is *case-sensitive*
  - So, "texas-fight" is different from "Texas-Fight"
  - caveat: the default mac file-system is dodgy

Unix Filesystem

- The filesystem is a hierarchical system of organizing files and directories
- The top level in the hierarchy is called the "root" and holds all files and directories.
- The name of the root directory is / (the “slash” directory)
- Typical system directories below the root directory include:
  /bin contains many of the programs which will be executed by users
  /etc files used by system administrators
  /dev hardware peripheral devices
  /proc a pseudo file system which tracks running processes and system state
  /lib system libraries
  /usr normally contains applications software
  /home home directories for different systems

Unix Filesystem (an upside-down tree)



Pathnames

- The full *pathname* of a file includes the file name and the name of the directory that holds the file, and the name of the directory that holds the directory that holds the file, and the name of the …
  ….*all the way up up to the root directory*

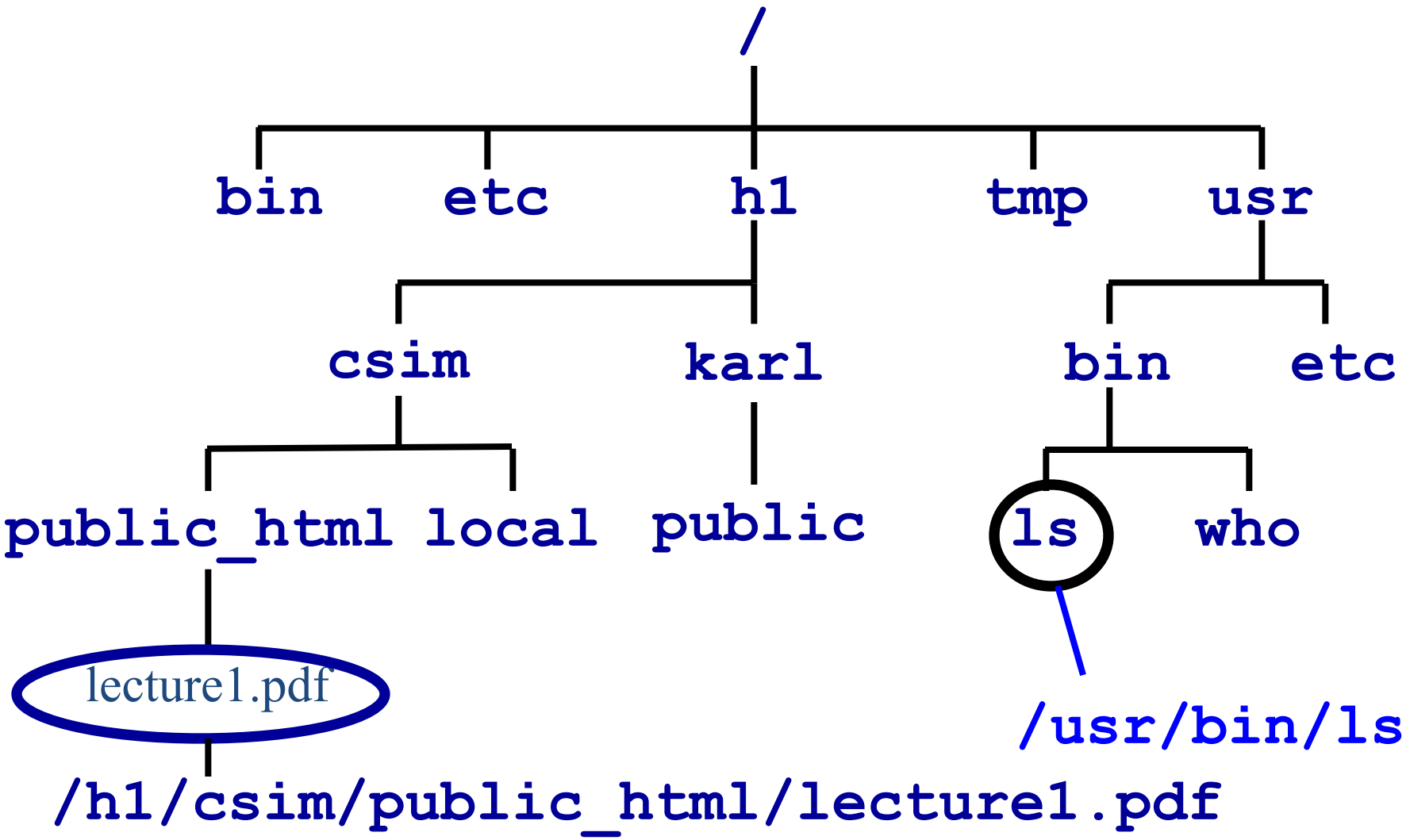
- The full pathname of every file in a Unix *filesystem* is unique (falls from the requirement that every file in the same directory must be a unique name)

Pathnames (cont.)

- To create a pathname you start at the root (so you start with "/"), then follow the path down the hierarchy (including each directory name) terminating with the filename

- In between every directory name you put a "/"

Pathname Examples

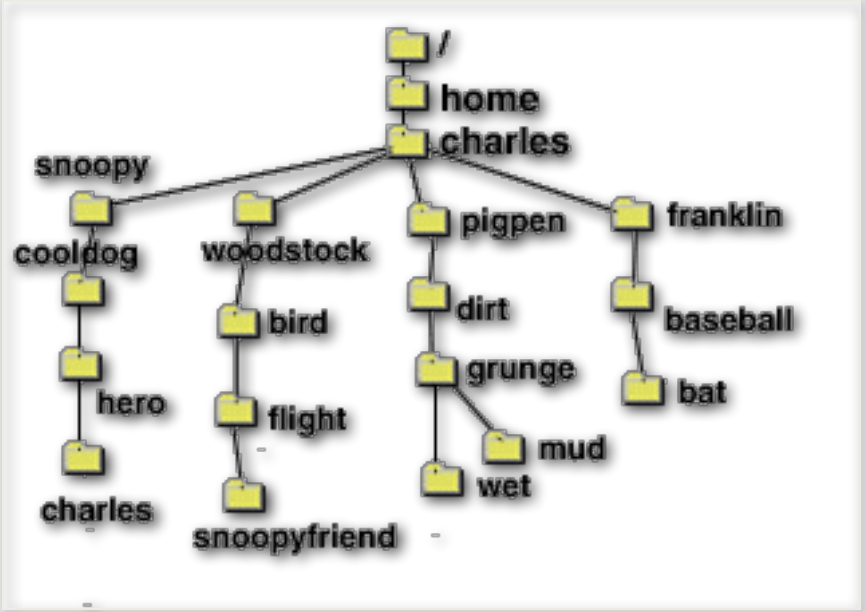


Absolute Pathnames

- The pathnames described in the previous slides start at the *root*
- These pathnames are *absolute pathnames*
- We can also talk about the pathname of a file *relative* to a directory

Relative Pathnames

- A *relative pathname* specifies a file in relation to the current working directory (CWD)
- If CWD=*/home, then the relative pathname to charles is: charles*
- If CWD=*/home, then the relative pathname to pigpen is: charles/pigpen*
- *If CWD=/home, then the relative pathname to baseball is: charles/franklin/baseball*



- Most Unix commands deal with pathnames
- We often use relative pathnames when specifying files (for convenience)

Special Directory Names

- There is a special relative pathname for the current working directory (CWD):

  . *(yes, that's a dot)*

  Example**: ./foo** (refers to "foo" in the current directory)

- There is also a special relative pathname for the parent directory:

  .. *(affectionately known as a dot-dot)*

  Example**: ../foo** (refers to "foo" in the parent directory)

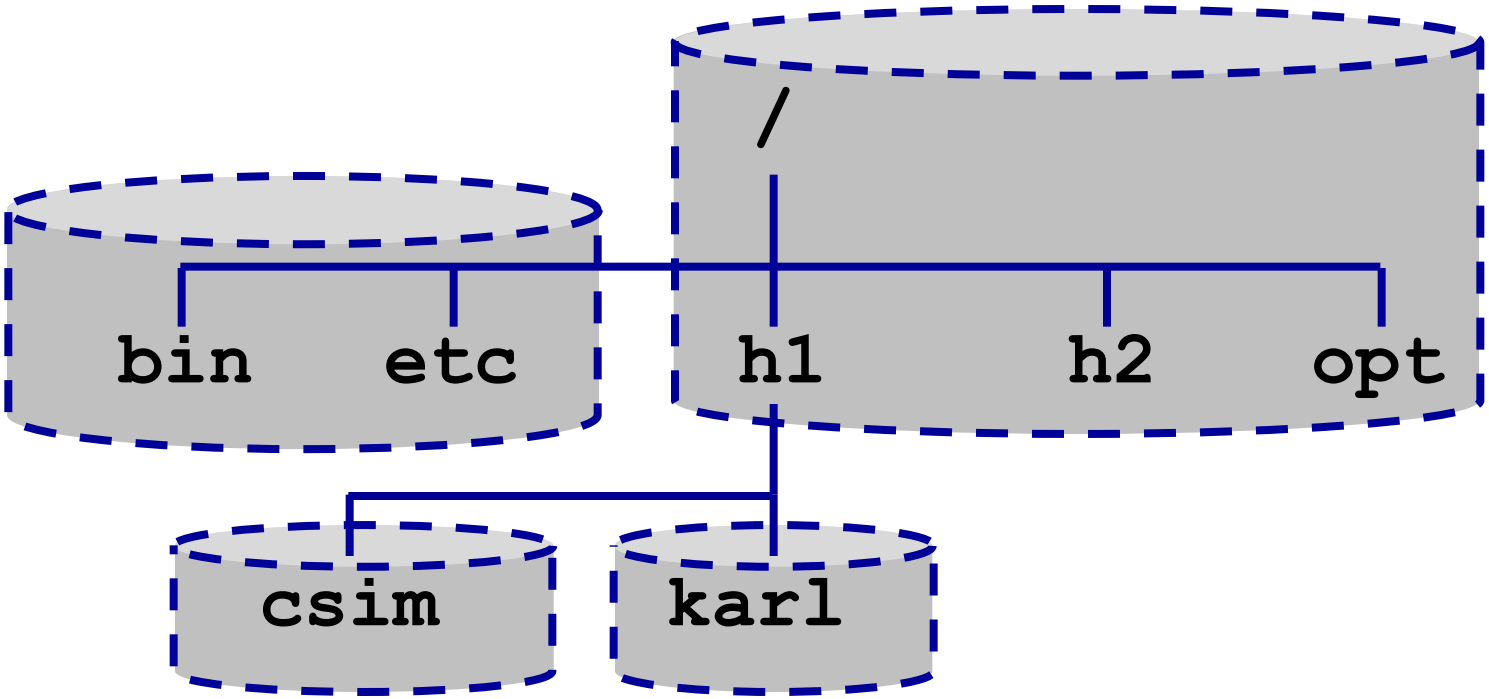
- There is a special symbol for the location of your home directory:

  ~ (that's a tilde)

  Example**: ~csim** (refers to the home directory for user "csim")

Disk vs. Filesystem

- Note that the file system hierarchy can actually be served by one or more physical disk drives
- In addition, some directories may be provided from other computers (e.g. NFS)



Basic Commands

- Some basic commands for interacting with the Unix file system are:
  - ls - pwd - touch
  - cd - cp - mkdir
  - df - awk - rmdir
  - cat - rm - find
  - more - chmod - grep
  - head - tail - chown/chgrp
- We will focus on ls first

The `ls` command

- The ls command displays the names of files
- If you give it the name of a directory as a *command line parameter* it will list all the files in the named directory

Example `ls` Commands

`ls`	list files in current directory
`ls /`	list files in the root directory
`ls .`	list files in the current directory
`ls ..`	list files in the parent directory
`ls /usr`	list files in the directory **`/usr`**

Command Line Options

- We can modify the output format of the ls program with a *command line option*.
- The ls command supports a bunch of options:
  - l *long* format (include file times, owner and permissions)
  - a *all* (shows hidden files as well as regular files)
  - F include special char to indicate file types

In Unix, hidden files have names that start with "."

`ls` Command Line Options

- To use a command line option precede the option letter with a minus:

  `ls -a` or `ls -l`

- You can use two or more options at the same time like this:

  `ls -al`

General `ls` command line

- The general form for the ls command is:

```
ls [options] [names]
```

- The options must come first!
- You can mix any options with any names.
- An example:

```
ls -al /usr/bin
```

Command Line Syntax

- `ls [options] [names]`
  - The brackets around options and names in the general form of the ls command means that something is optional
  - This type of description is common in the documentation for Unix commands
  - Some commands have required parameters

Variable Argument Lists

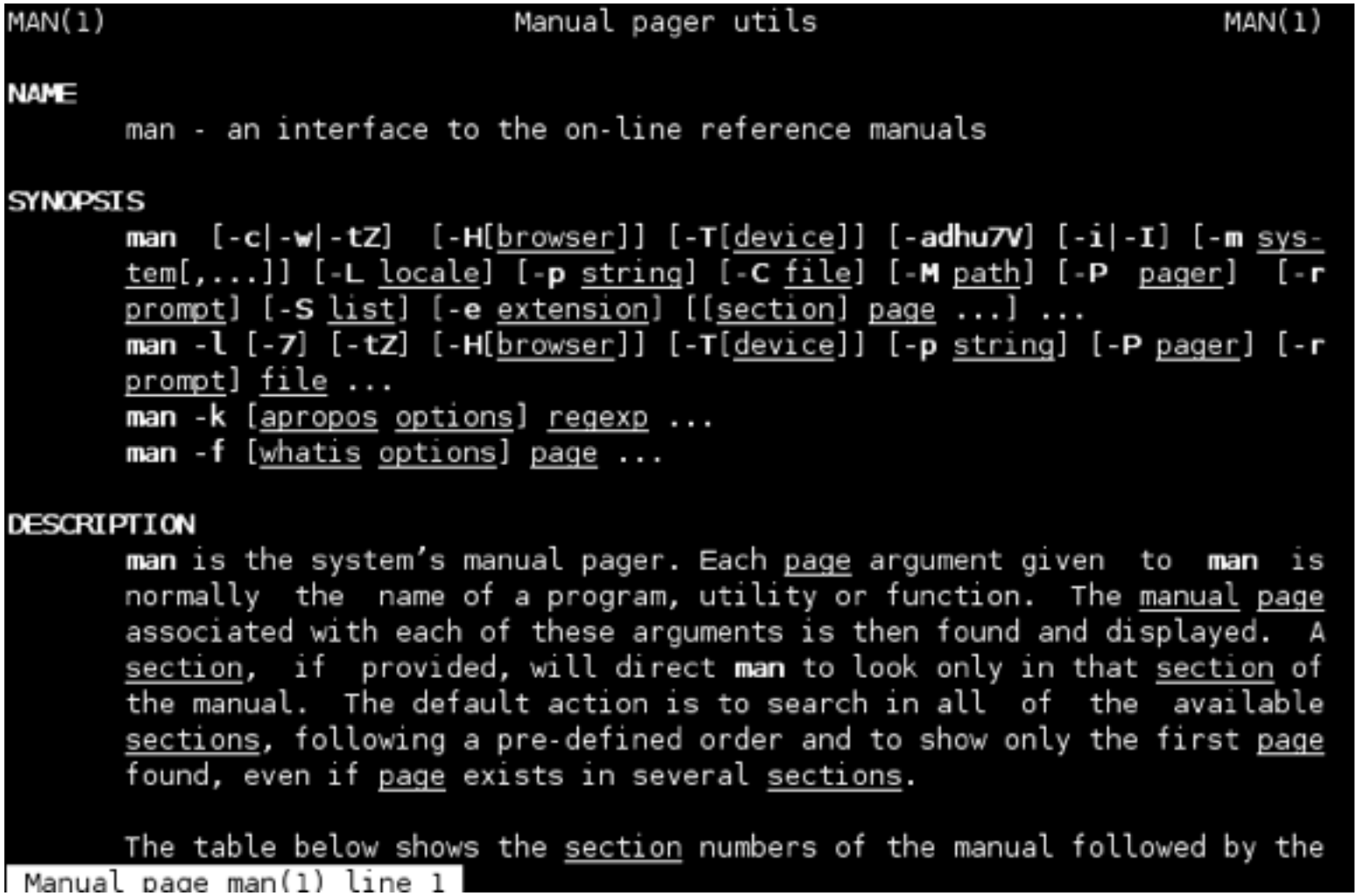
- You can give the ls command many files or directory names to display:

```
ls /usr /etc
ls -l /usr/bin /tmp /etc
```

Where to Get More Information?

- Almost all UNIX systems have extensive *on-line* documentation known as **man pages** (short for "manual pages").
- The Unix command used to display them is man. Each page is a self-contained document.
- So, to learn more about the ls command, refer to its man page:
  - **man ls**
- Man pages are generally split into 8 numbered sections (on BSD Unix and Linux):
  - 1 General commands
  - 2 System calls
  - 3 C library functions
  - 4 Special files (usually devices, those found in /dev)
  - 5 File formats and conventions
  - 6 Games
  - 7 Miscellaneous
  - 8 System administration commands and daemons
- You can request pages from specific sections:
  - **man 3 printf** (shows manpage for C library function)

Example Man Page



```
MAN(1)                          Manual pager utils                          MAN(1)

NAME
       man - an interface to the on-line reference manuals

SYNOPSIS
       man  [-c|-w|-tZ]  [-H[browser]] [-T[device]] [-adhu7V] [-i|-I] [-m sys-
       tem[,...]] [-L locale] [-p string] [-C file] [-M path] [-P  pager]  [-r
       prompt] [-S list] [-e extension] [[section] page ...] ...
       man -l [-7] [-tZ] [-H[browser]] [-T[device]] [-p string] [-P pager] [-r
       prompt] file ...
       man -k [apropos options] regexp ...
       man -f [whatis options] page ...

DESCRIPTION
       man is the system's manual pager. Each page argument given  to  man  is
       normally  the  name of a program, utility or function.  The manual page
       associated with each of these arguments is then found and displayed.  A
       section,  if  provided, will direct man to look only in that section of
       the manual.  The default action is to search in all  of  the  available
       sections, following a pre-defined order and to show only the first page
       found, even if page exists in several sections.

       The table below shows the section numbers of the manual followed by the
 Manual page man(1) line 1
```

Unix: A Culture in Itself

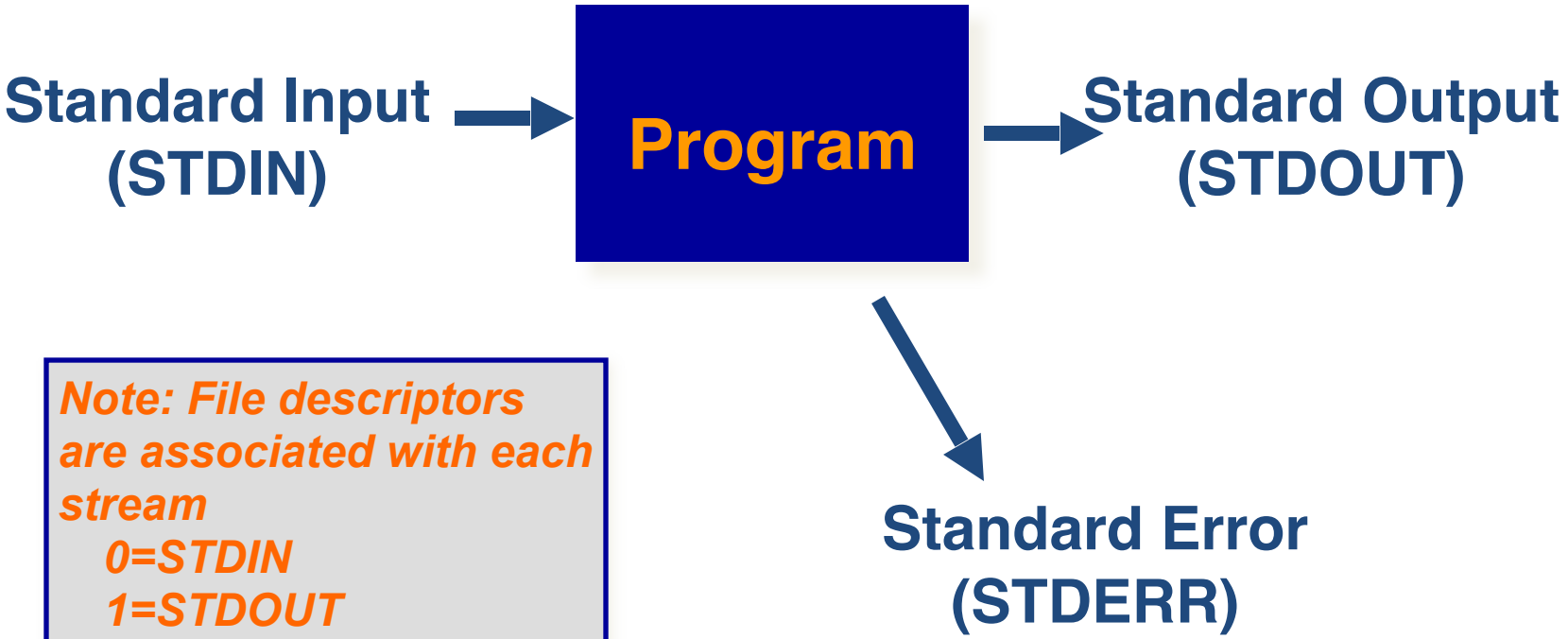
"Two of the most famous products of Berkeley are LSD and Unix. I don't think that this is a coincidence."
(Anonymous quote from The UNIX-HATERS Handbook.)

Interacting with the Shell

Running a Unix Program

- Typically, you type in the name of a program and some command line options
- The shell reads this line, finds the program and runs it, feeding it the options you specified
- The shell establishes 3 separate I/O streams:
  - Standard Input
  - Standard Output
  - Standard Error

Programs and Standard I/O

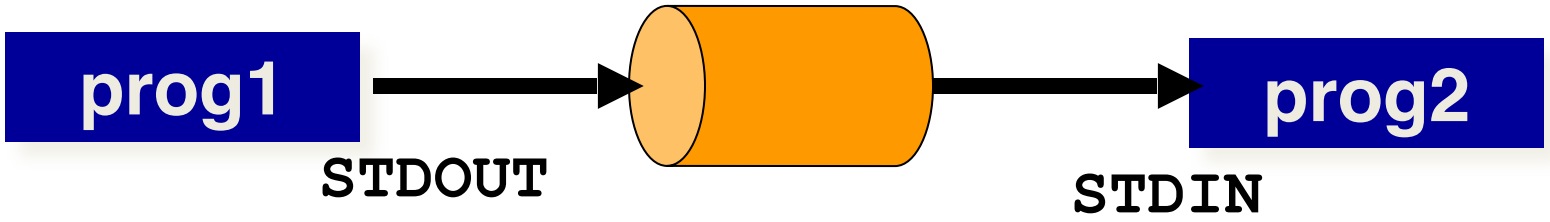


Note: File descriptors are associated with each stream

0=STDIN
1=STDOUT
2=STDERR

Unix Pipes

- A pipe is a holder for a stream of data
- A Unix **pipeline** is a set of processes chained by their standard streams, so that the output of each process (*stdout*) feeds directly as input (*stdin*) of the next one
- This is handy for using multiple unix commands together to perform a task



Building Commands

- More complicated commands can be built up by using one or more pipes
- Use the "**|**" character to *pipe* two commands together
- The shell takes care of all the hard work for you
- Example:

```
> cat apple.txt
core
worm seed
jewel

> cat apple.txt | wc
3  4  21
```

Note: the wc command prints the number of newlines, words, and bytes in a file

File Attributes

- Every file has a specific list of attributes:
  - Access Times:
    - when the file was created
    - when the file was last changed
    - when the file was last read
  - Size
  - Owners
    - user (*remember UID*)
    - group (*remember GID*)
  - Permissions

File Time Attributes

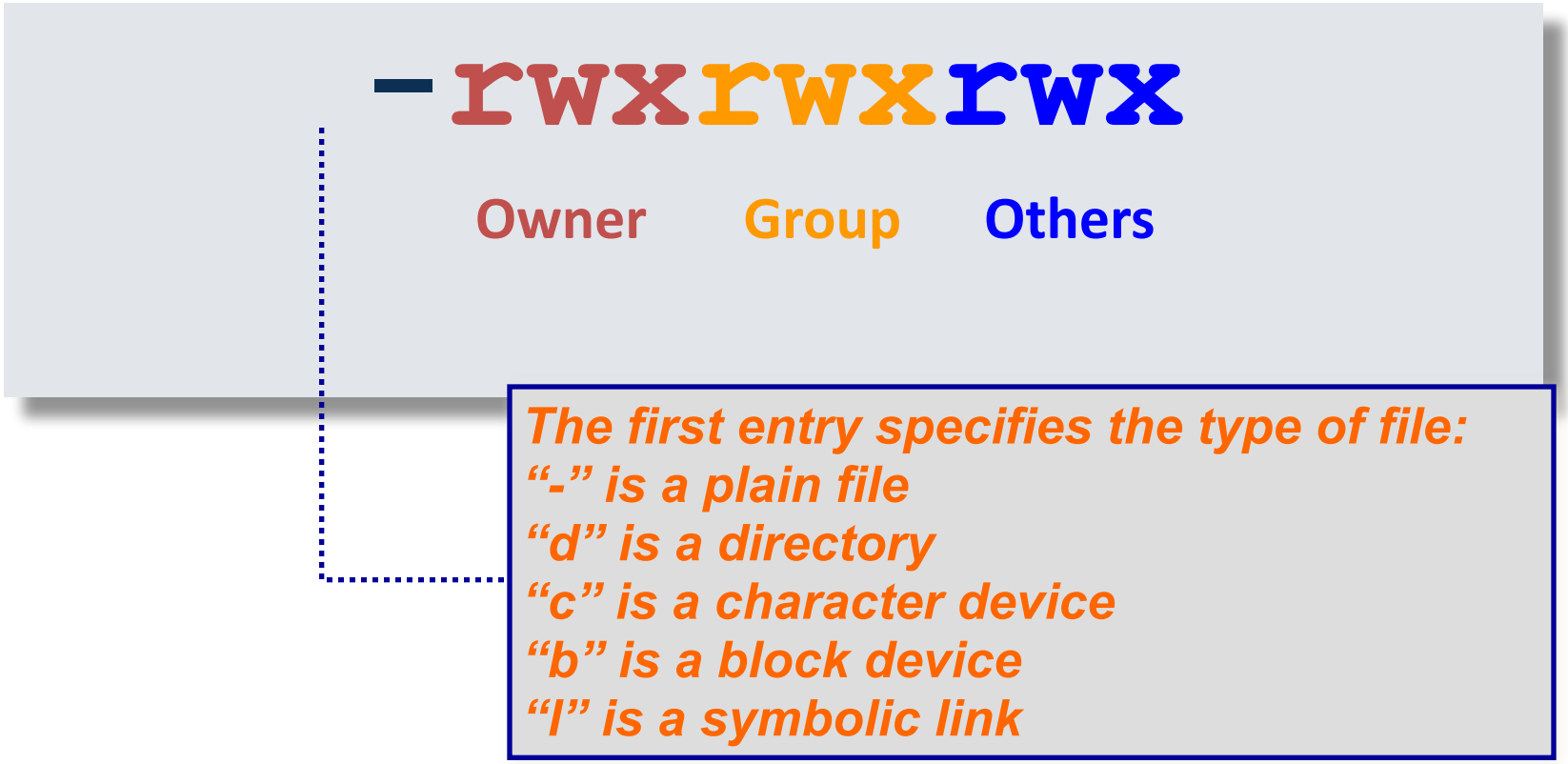
- Time Attributes:
  - ls -l shows when the file was last changed
  - ls -lc shows when the file was created
  - ls -lu shows when the file was last accessed

- Special names exist for these date-related attributes:
  - mtime (last modification time)
  - ctime (last change time, ie. when changes were made to the file or directory's inode: owner, permissions, etc.
  - atime (last access time)
  - Display with ‘stat’ command

File Permissions

- Each file has a set of permissions that control who can *access* the file
- There are three different types of permissions:
  - read abbreviated *r*
  - write abbreviated *w*
  - execute abbreviated *x*
- In Unix, there are permission levels associated with three types of people that might access a file:
  - owner (you)
  - group (a group of other users that you set up)
  - world (anyone else browsing around on the file system)

File Permissions Display Format



What is this *rwx* Craziness?

- Meaning for Files:
  - **`r`** – allowed to read
  - **`w`** – allowed to write
  - **`x`** – allowed to execute
- Meaning for Directories:
  - **`r`** – allowed to see the names of the files
  - **`w`** – allowed to add and remove files
  - **`x`** – allowed to enter the directory

Changing File Permissions

- The **`chmod`** command changes the permissions associated with a file or directory
- Basic syntax is: **`chmod mode file`**
- The *mode* can be specified in two ways:
  - symbolic representation
  - octal number
- Both methods achieve the same result (*user's choice)*
- Multiple symbolic operations can be given, separated by commas

chmod: Symbolic Representation

- *Symbolic* Mode representation has the following form:

```
[ugoa][+-=][rwxX…]
```

u=user	**+ add permission**	**r=read**
g=group	**- remove permission**	**w=write**
o=other	**= set permission**	**x=execute**
a = all	X= *see below*	

- The **X** permission option is very handy - it sets to execute only if the file is a directory or already has execute permission (*you really want to remember this one when using recursively*)

chmod Symbolic Mode Examples

```
> ls -al foo
-rw-------  1 karl support ...

> chmod g=rw foo
> ls -al foo
-rw-rw----  1 karl support ...

> chmod u-w,g+x,o=x foo
> ls -al foo
-r--rwx--x  1 karl support ...
```

chmod: Octal Representation

- Octal Mode uses a single argument string which describes the permissions for a file (3 digits)
- Each digit of this number is a code for each of the three permission levels (user,group,world)
- Permissions are set according to the following numbers:
  - Read = 4
  - Write = 2
  - Execute = 1
- Sum the individual permissions to get the desired combination

0 = no permissions whatsoever;
1 = execute only
2 = write only
3 = write and execute (1+2)
4 = read only
5 = read and execute (4+1)
6 = read and write (4+2)
7 = read and write and execute (4+2+1)

chmod Octal Mode Examples

```
> ls -al foo
-rw-------  1 karl support ...

> chmod 660 foo
> ls -al foo
-rw-rw----  1 karl support ...

> chmod 417 foo
> ls -al foo
-r----xrwx  1 karl support ...
```

Basic Commands

- Some basic commands for interacting with the Unix file system are:
  - ls - pwd - touch
  - cd - cp - mkdir
  - df - awk - rmdir
  - cat - rm - find
  - more (less) - chmod - grep
  - head - tail - chown/chgrp
- Let's cruise through some interactive examples....

UNIX Commands: find

- At its simplest, find searches the filesystem for files whose name matches a specific pattern
- However, it can do a lot more and is one of the most useful commands in Unix (as it can find specific files and then perform operations on them)

- Here is a simple example:

```
> ls
dir1   foo   foo2

> find . -name foo -print
./foo
```

UNIX Commands: find

- Find can also scan for certain file types. Here are some simple examples:

```
> find . -type d -print    (find directories)
> find . -type f -print    (find files)
```

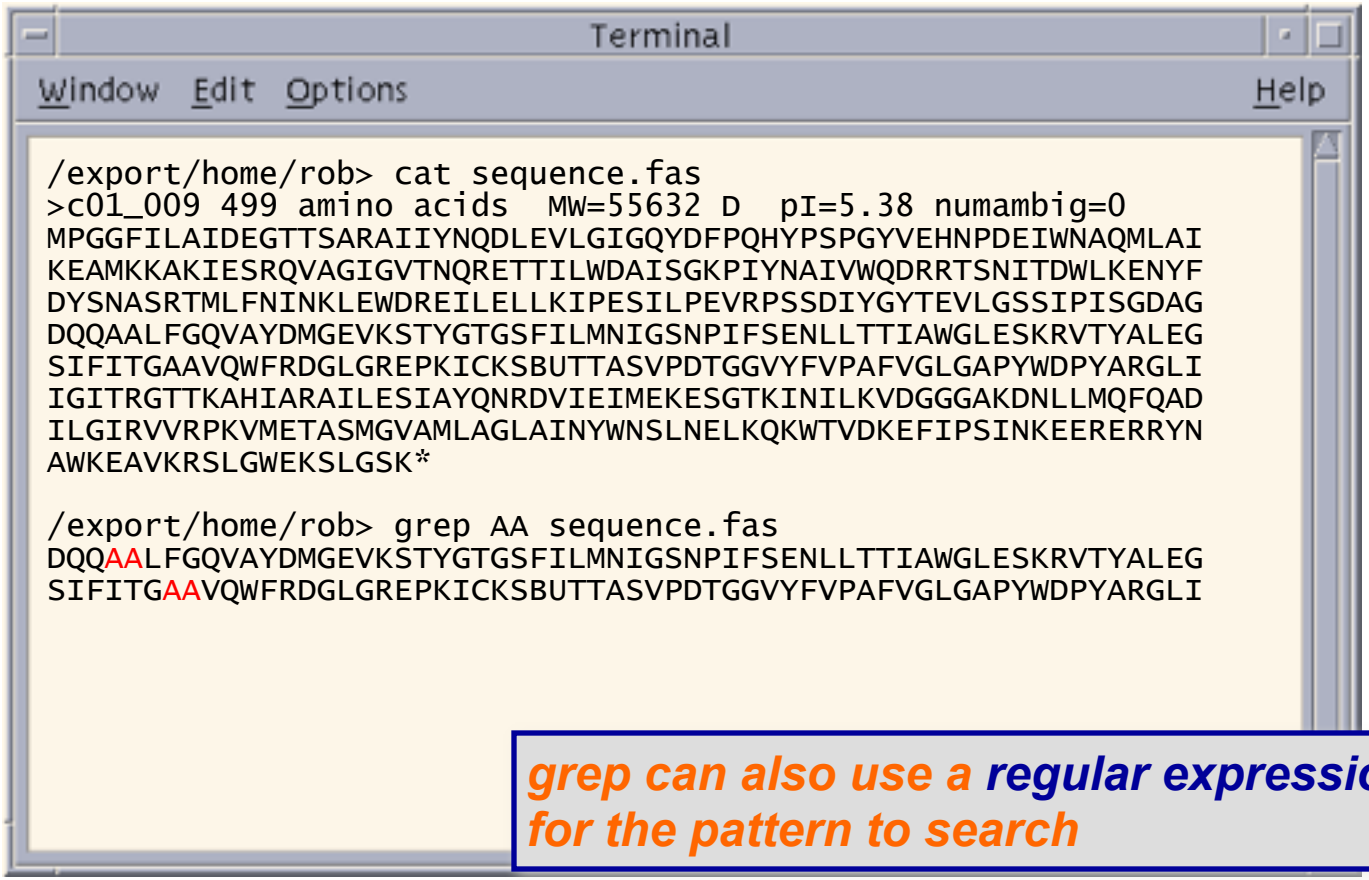
- Particularly powerful commands can be built using the exec option to issue commands on found files

```
>  find . -type f -exec wc -l {} \;
```

- What will the above do? (**Counts the # of lines in each file**)

UNIX Commands: grep

grep extracts lines from a file that match a given string or pattern



Regular Expressions

- In addition to grep, a number of Unix commands support the use of *regular expressions* to describe patterns:
  - sed
  - awk
  - perl
- General search pattern characters:
  - Any character (*except a metacharacter*) matches itself
  - "." matches any character except a newline
  - "*" matches zero or more occurrences of the single preceding character
  - "+" matches one or more of the proceeding character
  - "?" matches zero or one of the proceeding character
- Additional special characters:
  - "()" parentheses are used to quantify a sequence of characters
  - "|" works as an OR operator
  - "{}" braces are used to indicate ranges in the number of occurrences

Regular Expressions

- If you really want to match a period '**.**', you need to escape it with a backslash "**\.**"

Regexp	Matches	Does not match
a.b	axb	abc
a\.b	a.b	axb

Regular Expressions

- A *character class*, also called a character set can be used to match only one out of several characters
- To use, simply place the characters you want to match between square brackets []
- You can use a hyphen inside a character class to specify a range of characters
- Placing a caret (^) after the opening square bracket will negate the character class. The result is that the character class will match any character that is *not* in the character class
- Examples:

[abc]	matches a *single* a b or c
[0-9]	matches a *single* digit between 0 and 9
[^A-Za-z]	matches a single character as long as it is not a letter

Regular Expressions

- Since certain character classes are used often, a series of shorthand character classes are available for convenience:

  \d a digit. eg [0-9]
  \D a non-digit, eg. [^0-9]
  \w a word character (matches letters and digits)
  \W a non-word character
  \s a whitespace character
  \S a non-whitespace character

Regular Expressions

- More shorthand classes are available for *matching boundaries*:

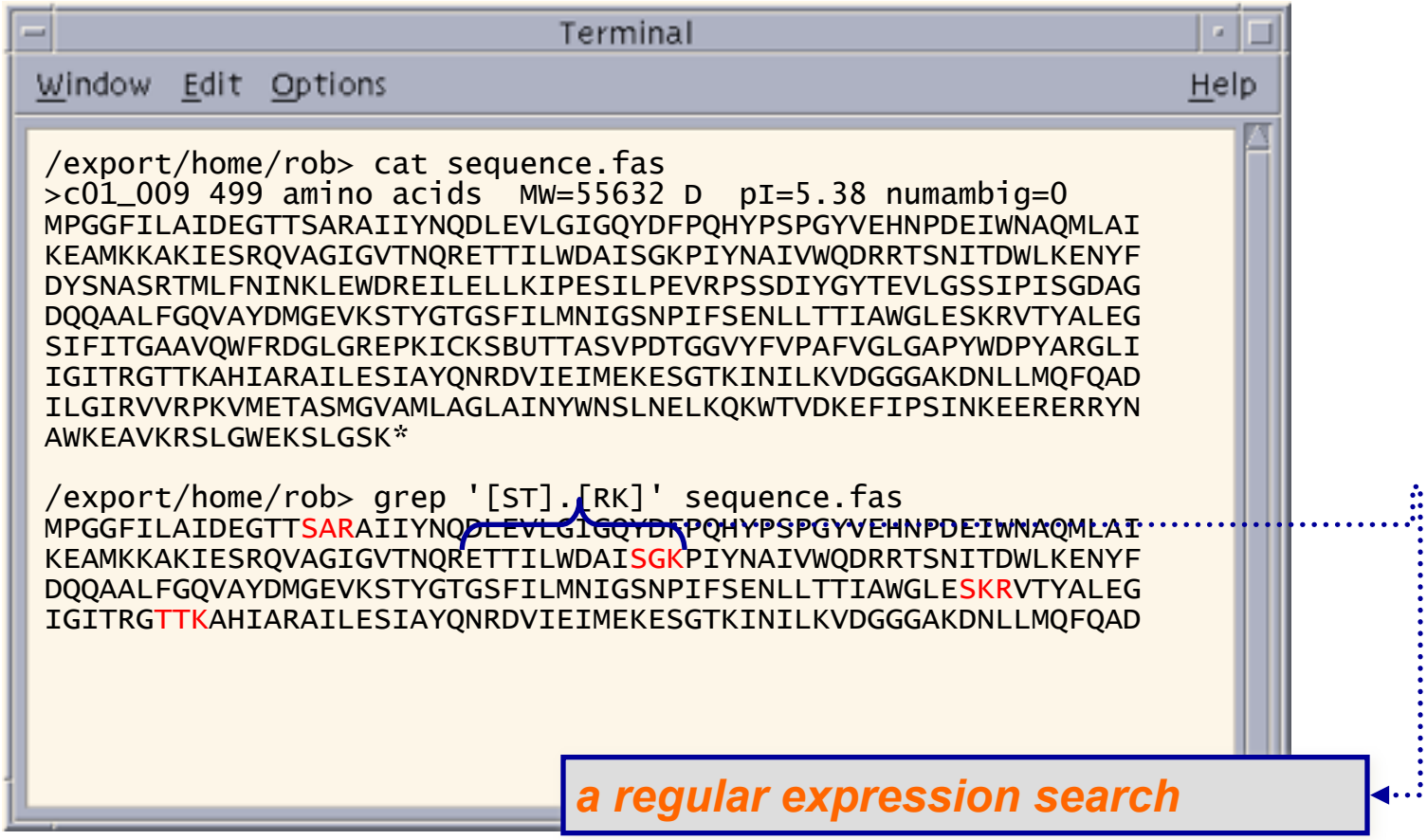
^	the beginning of a line
$	the end of a line
\b	a word boundary
\B	a non-word boundary

Regular Expressions Examples

- “notice” a string that has the text "notice" in it
- “F.” matches an “F” followed by any character
- “a.b” matches “a” followed by any 1 char followed by “b”
- “^The” matches any string that starts with "The"
- “oh boy$” matches a string that ends in the substring "oh boy";
- “^abc$” matches a string that starts and ends with "abc" -- that could only be "abc" itself!
- “ab*” matches an “*a*” followed by zero or more “*b*”'s ("a", "ab", "abbb", etc.)
- “ab+” similar to previous, but there's at least one “*b*” ("ab", "abbb", etc.)
- “(b|cd)ef” matches a string that has either "bef" or "cdef"
- “a(bc)*” matches an “*a*” followed by zero or more copies of the sequence "bc"
- “ab{3,5}” matches an “*a*” followed by three to five “*b*”'s ("abbb", "abbbb", or "abbbbb")
- “[Dd][Aa][Vv][Ee]” matches "Dave" or "dave" or "dAVE“, does not match "ave" or "da"

UNIX Commands: grep

grep extracts lines from a file that match a given string or pattern



regex: another unix culture

http://xkcd.com/208/

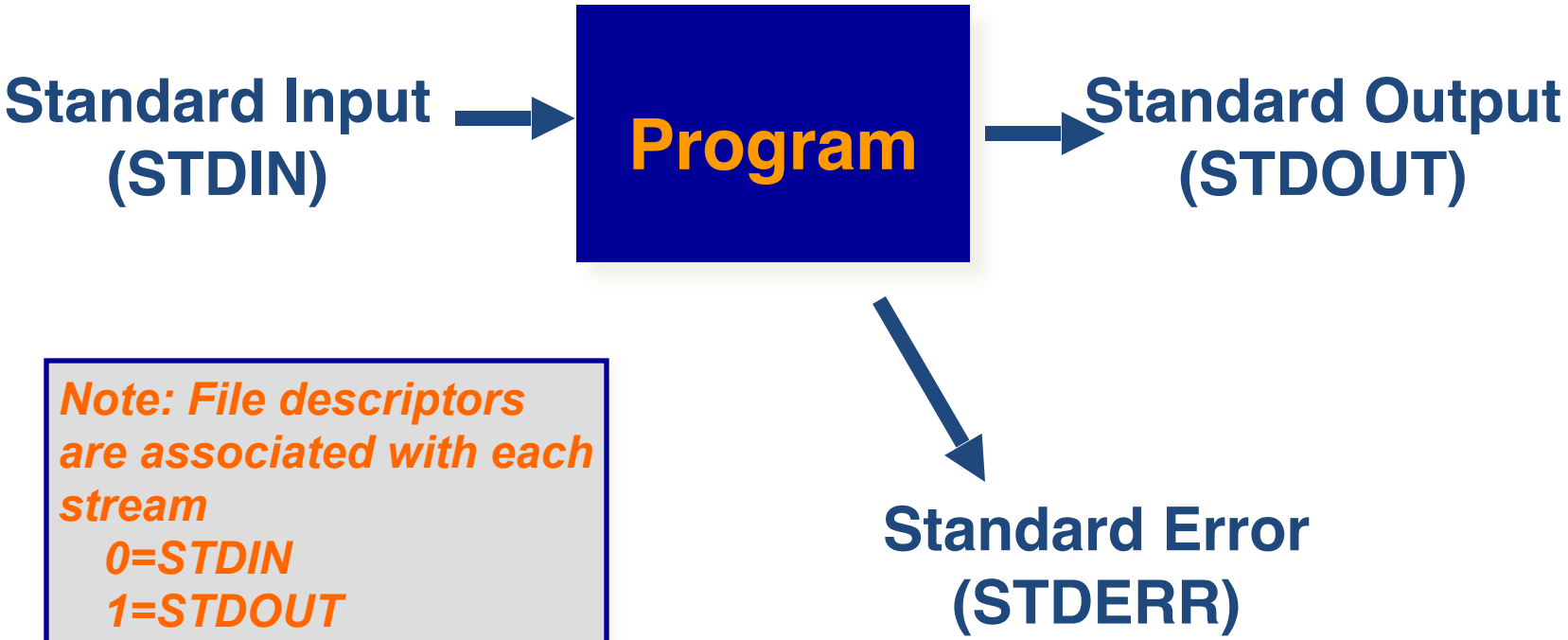
Shell Customization

- Each shell supports some customization.
  - user prompt settings
  - environment variable settings
  - aliases
- The customization takes place in *startup* files which are read by the shell when it starts up
  - Global files are read first - these are provided by the system administrators (eg. /etc/profile)
  - Local files are then read in the user's HOME directory to allow for additional customization

Shell Startup Files

```
sh,ksh:
   ~/.profile
bash:
   ~/.bash_profile
   ~/.bash_login
   ~/.profile
   ~/.bashrc
   ~/.bash_logout
tcsh:
   ~/.tshrc
   ~/.cshrc
   ~/.login
   ~/.logout
```

Programs and Standard I/O



Note: File descriptors are associated with each stream

0=STDIN
1=STDOUT
2=STDERR

Defaults for I/O

- When a shell runs a program for you:
  - standard input is your keyboard
  - standard output is your screen or window
  - standard error is your screen or window
- If standard input is your keyboard, you can type stuff in that goes to a program
- To end the input you press Ctrl-D (^D) on a line by itself, this ends the input *stream*
- The shell is a program that reads from standard input
- Any idea what happens when you give the shell ^D?

Shell Stream Redirection

- A very powerful function in Unix is redirection for input and output:
  - The shell can attach things other than your *keyboard* to *standard input (stdin)*
    - A file (the contents of the file are fed to a program as if you typed it) - common in scientific programming
    - A pipe (the output of another program is fed as input as if you typed it)
  - The shell can attach things other than your *screen* to *standard output (stderr)*
    - A file (the output of a program is stored in file)
    - A pipe (the output of a program is fed as input to another program

Stream Redirection

- To tell the shell to store the *output* of your program in a file, follow the command line for the program with the "**>**" character followed by the filename:

```
ls > lsout
```

- The command above will create a file named **`lsout`** and place the output of the **`ls`** command in the file

Stream Redirection

- To have the shell get standard *input* from a file, use the "<" character:

```
sort < nums
```

- The command above would sort the lines in the file **nums** and send the result to *stdout*

- The beauty of redirection is that you can do both forms together:

```
sort < nums > sortednums
```

Modes of Output Redirection

- There are two modes of output redirections:
  - "**>**" the create mode
  - "**>>**" the append mode

- For example:
  - the command **`ls > foo`** will create a new file named foo (deleting any existing file named foo).
  - if you use "**>>**" instead, the output will be appended to foo:

```
ls /etc >> foo
ls /usr >> foo
```

Stream Redirection

- Many commands send error messages to *standard error (stderr)* which is different from *stdout*.
- However, the “>” output redirection only applies to *stdout* (not *stderr*)
- To redirect *stderr* to a file you need to specify the request direclty (note that this syntax is shell dependent):
  - BASH
    - **“2>”** redirects *stderr* (eg. `ls foo blah gork 2> erroroutput )`
    - **“&>”** redirects *stdout* and *stderr* (eg. `ls foo &> /dev/null )`
    - **“>> filename 2>&1”** merges *stdout* and *stderr* and appends to filename

Example of stderr/out

```
[albook:~/tst] %% cat errout.c
#include <stdlib.h>
#include <stdio.h>

int main()
{
  fprintf(stdout,"a1\n");
  fprintf(stderr,"b2\n");
  return 0;
}
[albook:~/tst] %% cat erroutf.f
      program errout
      write(6,*) "a1"
      write(0,*) "b2"
      end program
```

```
[albook:~/tst] %% cc -o errout errout.c
[albook:~/tst] %% errout
a1
b2
[albook:~/tst] %% errout > what.out
b2
[albook:~/tst] %% cat what.out
a1
[albook:~/tst] %% errout 1> out.out 2> err.out
[albook:~/tst] %% cat out.out
a1
[albook:~/tst] %% cat err.out
b2
[albook:~/tst] %% errout > all.out 2>&1
[albook:~/tst] %% cat all.out
b2
a1
[albook:~/tst] %% errout &> all.out
[albook:~/tst] %% cat all.out
b2
a1
```

Note: this only works this way in sh/bash

Wildcards for Filename Abbreviation

- When you type in a command line the shell treats some characters as special (*metacharacters)*
- These special characters make it easy to specify filenames
- The shell processes what you give it, using the special characters to replace your command line with new strings

The special character *

- “*” matches anything.
- If you give the shell “*” by itself (as a command line argument), the shell will remove the * and replace it with all the filenames in the current directory.
- “**a*b**” matches all files in the current directory that start with **a** and end with **b**.
- This looks like regular expressions but isn’t quite the same.

Understanding *

- The **echo** command prints out whatever you tell it:

```
> echo hi
hi

> ls
dir1   foo   foo2
```

- What will the following command do?

```
> echo *
dir1 foo foo2
```

Understanding ?

- The **?** matches one single character:

```
> ls
dir1   foo1   foo2
```

- What will the following command do?

```
> ls foo?

foo1 foo2
```

Job Control

- **The shell allows you to manage *jobs***
  - place jobs in the *background*
  - move a job to the *foreground*
  - suspend a job
  - kill a job
- If you follow a command line with "&", the shell will run the *job* in the background
  - this is you useful if you don't want to wait for the job to complete
  - you can type in a new command right away
  - you can have a bunch of jobs running at once

```
> cat foo | sort | uniq > saved_sort &
```

Background jobs

- Handy for programs you need throughout a session:
  `emacs &`
- For commands that take a lot of time:
  `make all &> make.out &`
- If the job will run longer than your session:
  `nohup make all &> make.out &`

Listing Your Jobs

- The command *jobs* will list all background jobs:

```
> jobs
[1] Running  cat foo | sort | uniq >
saved_ls &
```

- The shell assigns a number to each job (in this case, the job number is 1)

Managing Jobs

- You can *kill* the foreground job by pressing ^C (Ctrl-C).
- You can also kill a job in the background using the *kill* command (and the job index)

```
> kill %1
```

Note: it's important to include the "%" sign to reference a job number.

Moving Jobs between fore/background

- Turn a foreground process into background:
  - Use ^-Z to suspend the command
  - Use the **bg** command to send the job to the background

```
> sleep 60
Suspended
> jobs
[1]   + Suspended                    sleep 60
> bg
[1]     sleep 60 &
> jobs
[1]     Running                      sleep 60
```

- The **fg** command will move a job to the foreground.
  - You give **fg** a job number (as reported by the **jobs** command)

```
> jobs
[1] Stopped          ls -lR > saved_ls &
> fg %1
ls -lR > saved_ls
```

Unix Environment Variables

- Unix shells maintain a list of environment variables which have a unique name and a value associated with them
  - some of these parameters determine the behavior of the shell
  - also determine which programs get run when commands are entered (and which libraries they link against)
  - provide information about the execution environment to programs
- We can access these variables:
  - set new values to customize the shell
  - find out the value of some to help accomplish a task

Environment Variables

- To view environment variables, use the env (or printenv)command
- If you know what you are looking for, you can use your new friend grep:

```
> env | grep PWD
PWD=/home/karl
```

- Use the echo command to print variables; the "$" prefix is required to access the value of the variable:

```
> echo $PWD
/tmp
```

- Can also use environment variables in arbitrary commands:

```
Koomie@canyon--> ls $PWD
foo1  foo2
```

Special Environment Variable: PATH

- Each time you provide the shell a command to execute, it does the following:
  - Checks to see if the command is a built-in shell command
  - If it is not a build-in command, the shell tries to find a program whose name matches the desired command
- How does the shell know where to look on the filesystem?
- The **`PATH`** variable tells the shell where to search for programs (non built-in commands)

Special Environment Variable: PATH

- Example PATH Definition:

```
-> echo $PATH
/home/karl/bin/krb5:/opt/intel/compiler70/ia32/
bin:/home/karl/bin:/usr/local/apps/mpich/icc/
bin:/usr/kerberos/bin:/usr/local/bin:/bin:/usr/
bin:/usr/X11R6/bin
```

- The **PATH** is a list of directories delimited by colons (":“)
  - It defines a list and *search order*
  - Directories specified earlier in the **PATH** take precedent; once the matching command is found, the search terminates
- You can add more search directories to your PATH by changing the shell startup files
  - **BASH: export PATH="$PATH":/home/karl/bin**

Other Important Variables

PWD *current working directory*

MANPATH *determines where to find man pages*

HOME *home directory of user*

MAIL *where your email is stored*

TERM *what kind of terminal you have*

PRINTER *specifies the default printer name*

EDITOR *used by many applications to identify your choice of editors (eg. vi or emacs)*

LD_LIBRARY_PATH *specifies a search path for dynamic runtime libraries*

Setting Environment Variables

- The syntax for setting Unix environment variables depends on your shell:
  - **BASH**: use the *export* command

```
> export PRINTER=scully
> echo $PRINTER
scully
```

  - **TCSH**: use the *setenv* command

```
> setenv PRINTER mulder
> echo $PRINTER
mulder
```

- Note: environment variables that you set interactively are only available in your current shell
  - If you spawn a new shell (or login again), these settings will be lost
  - To make permanent changes, you should alter the login scripts that affect your particular shell (eg. *.login, .bashrc, .cshrc, etc...)*

Modules on TACC Machines

- Modules are used to setup and remove various environment variables along with PATH, LD_LIBRARY_PATH declarations
- They are used to setup environments for packages & compilers.

```
hbar% module                                {lists options}
hbar% module list                           {lists loaded modules}
hbar% module avail                          {lists available modules}
hbar% module del <module>                   {removes a module}
hbar% module add <module>                   {add a module}
hbar% module switch <mod1> <mod2>               {switch modules}
```

- **Demonstration...**

Text Editors

Text Editors

- For programming and changing of various text files, we need to make use of available Unix text editors
- The two most popular and available editors are *vi* and *emacs*
- You should familiarize yourself with at least one of the two (*and this let's you enter into the editor wars which is a never-ending debate in the programming community)*

  http://en.wikipedia.org/wiki/Editor_war
- We will have very short introductions to each....

Brief history of Unix text editors

- **ed** : line mode editor
- **ex** : extended version of **ed**
- **vi** : full screen version of **ex**
- **vim** - Vi IMproved
- **emacs** : another popular editor, deep GNU,FSF roots
- **ed/ex/vi** share lots of syntax, which also comes back in **sed/awk**: useful to know.

Vi Overview

- Fundamental thing to remember about vi is that it has two different modes of operation:
  - *Insert* Mode
  - *Command* mode
- The *insert* mode puts anything typed on the keyboard into the current file
- The *command* mode allows the entry of commands to manipulate text. These commands are usually one or two characters long, and can be entered with few keystrokes
- Note that vi starts out in the *command* mode by default

Vi Overview

- Quick Start Commands
  - > **vi**
  - Press i to enable insert mode
  - Type text (*use arrow keys to move around*)
  - Press Esc to enable command mode
  - Press :w <filename> to save the file
  - Press :q to exit vi

Useful vi commands

- :q! – exit without saving the document. Very handy for beginners
- :wq – save and exit
- / <string> – search within the document for text. n goes to next result
- dd – delete the current line
- yy – copy the current line
- p – paste the last cut/deleted line
- :1 – goto first line in the file
- :$ - goto last line in the file
- $ – end of current line, ^ – beginning of line
- % – show matching brace, bracket, parentheses

Additional vi References

- http://www.eng.hawaii.edu/Tutor/vi.html
- http://staff.washington.edu/rells/R110/
- Vi Commands Reference card: http://tnerual.eriogerg.free.fr/vimqrc.pdf
- vimtutor – the Vim tutor

Emacs Overview

- Programmer friendly modes for common languages (C/C++, Fortran, shell scripts, etc)
- Different from vi in that emacs has only one-main mode
- Lots of commands and extremely customizable (using LISP)
- Includes some very sophisticated features if you take the time to learn them:
  - Compile your executables within emacs
  - Interact with your revision control process (eg. CVS/subversion)
  - Control RPM software builds
  - Debug your application using gdb

Emacs Overview

- *> emacs myfile opens myfile for editing*
- *Type whatever text you like (use arrow keys to navigate)*
- *C-x C-s (control + x, control + s) – saves the file*
- C-g exits the current command
- C-x u - Undo
- *C-x C-c exit after saving*

Additional Emacs References

- http://www.lib.uchicago.edu/keith/tcl-course/emacs-tutorial.html
- http://www.stolaf.edu/people/humke/UNIX/emacs-tutorial.html
- Emacs includes its own on-line tutorial; to run issue the following:
  - *> emacs*
  - Then, enter "C-h t", to invoke the on-line emacs tutorial (*that's a "Control-h", followed by a "t"*)

Unix tool: sed

- Stream editor: editor commands applied to an input file or stream, giving an output stream

```
%% cat 123
1 one
2 word
3 is is
4 enough
5 words
%% sed s/word/picture/ 123
1 one
2 picture
3 is is
4 enough
5 pictures
%% sed 2,4s/word/picture/ 123
1 one
2 picture
3 is is
4 enough
5 words
%% sed -e 's/word/picture/' -e 's/is$/often/' 123
1 one
2 picture
3 is often
4 enough
5 pictures
%%
```

Unix tool: awk

- Pattern/action pairs, applied successively to each line

```
[albook:~/tst] %% cat awk.in
C from file1.f
        subroutine foo
        call something
        end
C from file2.f
        subroutine bar
        call something(else)
        end
```

```
%% awk '{print $0}' awk.in
C from file1.f
        subroutine foo
        call something
        end
C from file2.f
        subroutine bar
        call something(else)
        end
%% awk '{print $1}' awk.in
C
subroutine
call
end
C
subroutine
call
end
%% awk '/subroutine/ {print $2}' awk.in
foo
bar
```

Unix Scripting

- Scripting is “easy” - you just place all the Unix commands in a file as opposed to typing them interactively
- Handy for automating certain tasks:
  - staging your scientific applications
  - performing limited post-processing operations
  - any repetitive operations on files, etc...
- Shells provide basic control syntax for looping, if constructs, etc...

Unix Scripting

- Shell scripts must begin with a specific line to indicate which shell should be used to execute the remaining commands in the file:
  - BASH:
    #!/bin/bash
  - TCSH
    #!/bin/tcsh
- Comment lines can be included if they start with #
- In order to run a shell-script, it must have execute permission. Consider the following script:

```
> cat hello.sh
#!/bin/bash
echo "hello world"

> ./hello.sh
./hello.sh: Permission denied.
```

```
> chmod 700 hello.sh
> ./hello.sh
hello world
```

Unix Scripting: Arithmetic Operations

- Simple arithmetic syntax depends on the shell:

```
i1=2
j1=6
k1=$(($i1*$j1))
echo "The multiplication of $i1 and $j1 is $k1"
```

- Note, you can also use the **`expr`** command (for both shells). For example:
  - `z=`expr $i1 + $j1``

- **For floating point use bc**

```
$ echo "scale=4; 2 / 3" | bc -l
.6666
```

*consult man page on **`expr`** and **`bc`** for more details*

Unix Scripting: Conditionals

- Syntax for conditional expressions depends on your choice of shell:
- BASH (general format):

```
if [ condition_A ]; then
        code to run if condition_A true
elif [ condition_B ]; then
        code to run if condition_A false and
    condition_B true
else
        code to run if both conditions false
fi
```

Unix Scripting: String Comparisons

- string1 = string2 Test identity
- string1 !=string2 Test inequality
- -n string the length of *string* is nonzero
- -z string the length of *string* is zero

BASH Example:

```
today="monday"
if [ "$today" = "monday" ] ; then
    echo "today is monday"
fi
```

BASH Integer Comparisons

- int1 –eq int2 Test identity
- int1 –ne int2 Test inequality
- int1 –lt int2 Less than
- int1 –gt int2 Greater than
- int1 –le int2 Less than or equal
- int1 –ge int2 Greater than or equal

BASH Example:

```
x=13
y=25
if [ $x -lt $y ]; then
  echo "$x is less than $y"
fi
```

Unix Scripting: Common File Tests

- -d file Test if file is a directory
- -f file Test if file is not a directory
- -s file Test if the file has non zero length
- -r file Test if the file is readable
- -w file Test if the file is writable
- -x file Test if the file is executable
- -o file Test if the file is owned by the user
- -e file Test if the file exists

BASH Example:

```
if [ -f foo ]; then
  echo "foo is a file"
fi
```

Unix Scripting: For loops

- These are useful when you want to run the same command in sequence with different options
- *sh* example:

```
for VAR in test1 test5 test7b finaltest; do
    runmycode $VAR > $VAR.out
Done
```

- *sh* one-liner

```
for i in `seq 1 5`; do echo $i; done
1
2
3
4
5
```

Quoting in Unix

- We've seen that some metacharacters are treated special on the command line: `* ?`
- What if we don't want the shell to treat these as special - we really mean *, not all the files in the current directory
- To turn off special meaning - surround a string with double quotes:

```
> echo here is a star "*"
here is a star *
```

Use of Quotes

- You have to be careful with the use of different styles of quotes in your commands or scripts

- They have different functions:
  - Double quotes inhibit wildcard replacement only
  - Single quotes inhibit wildcard replacement, variable substitution and command substitution
  - Back quotes cause command substitution

Double Quotes

- Double quotes around a string turn the string in to a *single* command line parameter:

```
> ls
fee file? foo
> ls "foo fee file?"
ls: foo fee file?: No such file or directory
```

- Double quotes only inhibit wildcards; use \ to escape special characters:

```
> echo “This is a quote \" “
This is a quote “
```

Single Quotes

- Single quotes are similar to double quotes, but they also inhibit variable substitution and command substitution
- Means that special characters do not have to be escaped:

```
> echo 'This is a quote \" '
This is a quote \"
```

Back Quotes

- If you surround a string with back quotes, the string is replaced with the result of running the command in back quotes:

```
> echo `ls`
foo fee file?

> echo "It is now `date` and OU is still
questionable”
It is now Tue Sep 19 11:24:25 CDT 2006 and OU is
still questionable
```

More Quote Examples

- Some Quoting Examples:

```
$ echo Today is date
Today is date
$ echo Today is `date`
Today is Thu Sep 19 12:28:55 EST 2002
$ echo ”Today is `date`”
Today is Thu Sep 19 12:28:55 EST 2002
$ echo ‘Today is `date`’
Today is `date`
```

“ “ = double quotes
‘ ‘ = single quotes
` ` = back quotes

Command-Line Parsing

- To build generic shell scripts, consider using command-line arguments to provide the inputs you need internally (syntax again depends on the choice of shell)
- Syntax:
  - $# *refers to the number of command-line arguments*
  - $0 *refers to the name of the calling command*
  - $1, $2, …, $N *refers to the Nth argument*
  - $* *refers to all command-line parameters*

```
echo "Calling command is:          $0"
echo "Total # of arguments is:     $#"
echo "A list of all arguments is: $*"
echo "The 2nd argument is:         $2"

> ./foo.sh texas rose bowl
Calling command is:          ./foo.sh
Total # of arguments is:     3
A list of all arguments is: texas rose bowl
The 2nd argument is:         rose
```

More UNIX Commands for Programmers

– man –k	Search man pages by topic
– time	How long your program took to run
– date	print out current date/time
– test	Compare values, existence of files, etc
– tee	Replicate output to one or more files
– diff	Report differences between two files
– sdiff	Report differences side-by-side
– wc	Show number of lines, words in a file
– sort	Sort a file line by line
– gzip	Compress a file
– gunzip	Uncompress it
– strings	Print out ASCII strings from a (binary)
– ldd	Show shared libraries program is linked to
– nm	Show detailed info about a binary obj
– tar	Archiving utility
– uniq	Remove duplicate lines from a sorted file
– which	Show full path to a command
– file	Determine file type

Text editors – another subculture