Introduction to UNIX

Adapted by

Fredj Tekaia

from

Mario Dantas course

Federal University of Santa Catarina, Florianopolis

Institut Pasteur - EMBO course, June 30 - July 12, 2008

Florianopolis, Brasil.
Objective

• In this course will introduce the basics of UNIX programming.
Course Outline

- Introduction
- Operating system overview
- UNIX utilities
- Scripting languages
- Programming tools
Course Outline

- Introduction
- Operating system overview
- UNIX utilities
- Scripting languages
- Programming tools
Commercial Success

- AIX
- SunOS, Solaris
- Ultrix, Digital Unix
- HP-UX
- Irix
- UnixWare -> Novell -> SCO -> Caldera -> SCO
- Xenix: -> SCO
- Standardization (Posix, X/Open)
The UNIX Philosophy

• Small is beautiful
  – Easy to understand
  – Easy to maintain
  – More efficient
  – Better for reuse

• Make each program do one thing well
  – More complex functionality by combining programs
  – Make every program a filter
The UNIX Philosophy

• Portability over efficiency
  – Most efficient implementation is rarely portable
  – Portability better for rapidly changing hardware

• Use flat ASCII files
  – Common, simple file format (yesterday’s XML)
  – Example of portability over efficiency

• Reusable code
  – Good programmers write good code;
    great programmers borrow good code
The UNIX Philosophy

• Scripting increases leverage and portability

```
print $(who | awk '{print $1}' | sort | uniq) | sed 's/ /,/g'
```

List the logins of a system’s users on a single line.

<table>
<thead>
<tr>
<th>Command</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>who</td>
<td>755</td>
</tr>
<tr>
<td>awk</td>
<td>3,412</td>
</tr>
<tr>
<td>sort</td>
<td>2,614</td>
</tr>
<tr>
<td>uniq</td>
<td>302</td>
</tr>
<tr>
<td>sed</td>
<td>2,093</td>
</tr>
</tbody>
</table>

9,176 lines

• Build prototypes quickly (high level interpreted languages)

..continued
The UNIX Philosophy

..continued

- Avoid captive interfaces
  - The user of a program isn’t always human
  - Look nice, but code is big and ugly
  - Problems with scale
- Silence is golden
  - Only report if something is wrong
- Think hierarchically
UNIX Highlights / Contributions

- Portability (variety of hardware; C implementation)
- Hierarchical file system; the file abstraction
- Multitasking and multiuser capability for minicomputer
- Inter-process communication
  - Pipes: output of one programmed fed into input of another
- Software tools
- Development tools
- Scripting languages
- TCP/IP
The Operating System

• The government of your computer
• Kernel: Performs critical system functions and interacts with the hardware
• Systems utilities: Programs and libraries that provide various functions through systems calls to the kernel
Kernel Basics

• The kernel is …
  – a program loaded into memory during the boot process, and always stays in physical memory.
  – responsible for managing CPU and memory for processes, managing file systems, and interacting with devices.
UNIX Structural Layout

User Space
- shell scripts
- system calls

Kernel
- signal handler
- device drivers
- scheduler
- swapper

Devices
- terminal
- disk
- printer
- RAM

Utilities
- C programs
- compilers
Kernel Subsystems

• Process management
  – Schedule processes to run on CPU
  – Inter-process communication (IPC)

• Memory management
  – Virtual memory
  – Paging and swapping

• I/O system
  – File system
  – Device drivers
  – Buffer cache
System Calls

• Interface to the kernel
• Over 1,000 system calls available on Linux
• 3 main categories
  – File/device manipulation
    • e.g. mkdir(), unlink()
  – Process control
    • e.g. fork(), execve(), nice()
  – Information manipulation
    • e.g. getuid(), time()
Logging In

• Need an account and password first
  – Enter at login: prompt
  – Password not echoed
  – After successful login, you will see a shell prompt

• Entering commands
  – At the shell prompt, type in commands
    • Typical format: **command options arguments**
    • Examples: **who, date, ls, cat myfile, ls -l**
  – Case sensitive

• *exit* to log out
Remote Login

• Use Secure Shell (SSH)

• UNIX-like OS
  – ssh name@access.cims.nyu.edu
Unix System Structure
Kernel Subsystems

• File system
  – Deals with all input and output
    • Includes files and terminals
    • Integration of storage devices

• Process management
  – Deals with programs and program interaction
    • How processes share CPU, memory and signals
    • Scheduling
    • Interprocess Communication
    • Memory management

• UNIX variants have different implementations of different subsystems.
What is a shell?

• The user interface to the operating system
• Functionality:
  – Execute other programs
  – Manage files
  – Manage processes
• A program like any other
• Executed when you log on
Most Commonly Used Shells

- `/bin/sh` The Bourne Shell / POSIX shell
- `/bin/csh` C shell
- `/bin/tcsh` Enhanced C Shell
- `/bin/ksh` Korn shell
- `/bin/bash` Free ksh clone

Basic form of shell:

```bash
while (read command) {
    parse command
    execute command
}
```
Shell Interactive Use

When you log in, you interactively use the shell:

– Command history
– Command line editing
– File expansion (tab completion)
– Command expansion
– Key bindings
– Spelling correction
– Job control
Shell Scripting

• A set of shell commands that constitute an executable *program*
• A shell script is a regular text file that contains shell or UNIX commands
• Very useful for automating repetitive task and administrative tools and for storing commands for later execution
Simple Commands

• **simple command**: sequence of non-blanks arguments separated by blanks or tabs.

• 1st argument (numbered zero) usually specifies the name of the command to be executed.

• Any remaining arguments:
  – Are passed as arguments to that command.
  – Arguments may be filenames, pathnames, directories or special options (up to command)
  – Special characters are interpreted by shell
A simple example

$ ls -l /bin
-rwxr-xr-x 1 root  sys  43234 Sep 26 2001 date
$

- prompt  command  arguments

• Execute a basic command
• Parsing into command in arguments is called *splitting*
Getting Help on UNIX

- **man**: display entries from UNIX online documentation
- **whatis, apropos**
- Manual entries organization:
  - 1. Commands
  - 2. System calls
  - 3. Subroutines
  - 4. Special files
  - 5. File format and conventions
  - 6. Games

NAME
ls - list files and/or directories

SYNOPSIS
ls [options] [file ...]

DESCRIPTION
For each directory argument ls lists the contents; for each file argument the name and requested information are listed. The current directory is listed if no file arguments appear. The listing is sorted by file name by default, except that file arguments are listed before directories.

OPTIONS
-a, --all
List entries starting with .; turns off --almost-all.
-F, --classify
Append a character for typing each entry.
-l, --long/verbose
Use a long listing format.
-r, --reverse
Reverse order while sorting.
-R, --recursive
List subdirectories recursively.

SEE ALSO
chmod(1), find(1), getconf(1), tw(1)
Fundamentals of Security

• UNIX systems have one or more users, identified with a number and name.

• A set of users can form a group. A user can be a member of multiple groups.
  
  • A special user (id 0, name root) has complete control.
  • Each user has a primary (default) group.
How are Users & Groups used?

- Used to determine if file or process operations can be performed:
  - Can a given file be read? written to?
  - Can this program be run?
  - Can I use this piece of hardware?
  - Can I stop a particular process that’s running?
A simple example

$ ls -l /bin
-rwrxr-xr-x 1 root  sys  43234 Sep 26  2001 date
$

read  write  execute
The UNIX File Hierarchy
Hierarchies are Ubiquitous
Definition: Filename

A sequence of characters other than slash.  
Case sensitive.
Definition: Directory

Holds a set of files or other directories. Case sensitive.
Definition: Pathname

A sequence of directory names followed by a simple filename, each separated from the previous one by a /
Definition: Working Directory

A directory that file names refer to by default.

One per process.
Definition: Relative Pathname

A pathname relative to the working directory (as opposed to absolute pathname)

.. refers to parent directory
. refers to current directory

../wm4/.profile
./.profile
../usr/.profile
Files and Directories

• Files are just a sequence of bytes
  – No file types (data vs. executable)
  – No sections
  – Example of UNIX philosophy

• Directories are a list of files and status of the files:
  – Creation date
  – Attributes
  – etc.
Tilde Expansion

- Each user has a *home* directory
- Most shells (ksh, csh) support ~ operator:
  - ~ expands to my home directory
    - ~/myfile \rightarrow /home/kornj/myfile
  - ~user expands to user’s home directory
    - ~unixtool/file2 \rightarrow /home/unixtool/file2
- Useful because home directory locations vary by machine
Printing File Contents

• The `cat` command can be used to copy the contents of a file to the terminal. When invoked with a list of file names, it concatenates them.

• Some options:
  - `-n` number output lines (starting from 1)
  - `-v` display control-characters in visible form (e.g. `^C`)

• Interactive commands `more` and `less` show a page at a time
Common Utilities for Managing files and directories

- **pwd** print process working dir
- **ed, vi, emacs**… create/edit files
- **ls** list contents of directory
- **rm** remove file
- **mv** rename file
- **cp** copy a file
- **touch** create an empty file or update
- **mkdir** and **rmdir** create and remove dir
- **wc** counts the words in a file
- **file** determine file contents
- **du** directory usage
File Permissions

• UNIX provides a way to protect files based on users and groups.

• Three **types** of permissions:
  • read, process may read contents of file
  • write, process may write contents of file
  • execute, process may execute file

• Three **sets** of permissions:
  • permissions for owner
  • permissions for group (1 group per file)
  • permissions for other
Directory permissions

• Same types and sets of permissions as for files:
  – **read**: process may a read the directory *contents* (i.e., list files)
  – **write**: process may add/remove files in the directory
  – **execute**: process may open files in directory or subdirectories
Utilities for Manipulating file attributes

- **chmod** change file permissions
- **chown** change file owner
- **chgrp** change file group
- **umask** user file creation mode mask

- only owner or super-user can change file attributes

- upon creation, default permissions given to file modified by process **umask** value
Chmod command

- Symbolic access modes \( \{u,g,o\} / \{r,w,x\} \)
  - example: `chmod +r file`

- Octal access modes

<table>
<thead>
<tr>
<th>octal</th>
<th>read</th>
<th>write</th>
<th>execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>1</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>6</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>7</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
Redirection

• Before a command is executed, the input and output can be changed from the default (terminal) to a file
  – Shell modifies file descriptors in child process
  – The child program knows nothing about this
Redirection of input/output

- **Redirection of output:** >
  - example: `$ ls > my_files`

- **Redirection of input:** <
  - example: `$ mail kornj <input.data`

- **Append output:** >>
  - example: `$ date >> logfile`

- **Bourne Shell derivatives:** `fd>`
  - example: `$ ls 2> error_log`
Using Devices

• Redirection works with devices (just like files)
• Special files in /dev directory
  – Example: /dev/tty
  – Example: /dev/lp
  – Example: /dev/null
    • cat big_file > /dev/lp
    • cat big_file > /dev/null
Symbolic links

- **Symbolic** links are created with `ln -s`
- Can be thought of as a directory entry that points to the *name* of another file.
- Does not change link count for file
  - When original deleted, symbolic link remains
Tree Walking

- How can we find a set of files in the hierarchy?
- One possibility:
  - `ls -l -R /`
- What about:
  - All files below a given directory in the hierarchy?
  - All files since Jan 1, 2001?
  - All files larger than 10K?
find utility

• **find** pathlist expression
• **find** recursively descends through pathlist and applies expression to every file.
• expression can be:
  - `-name pattern`
    • true if file name matches pattern. Pattern may include shell patterns such as *, must be in quotes to suppress shell interpretation.
  - Eg: `find / -name '*.c'`
find utility (continued)

- **-perm [+-]mode**
  - Find files with given access mode, mode must be in octal.
    Eg: find . 755

- **-type ch**
  - Find files of type ch (c=character, b=block, f for plain file, etc..). Eg: find /home -type f

- **-user userid/username**
  - Find by owner userid or username

- **-group groupid/groupname**
  - Find by group groupid or groupname

- **-size size**
  - File size is at least size

- **many more...**
find: actions

- **-print** prints out the name of the current file (default)

- **-exec cmd**
  - Executes *cmd*, where *cmd* must be terminated by an escaped semicolon (\; or ';').
  - If you specify {} as a command line argument, it is replaced by the name of the current file just found.
  - **exec** executes *cmd* once per file.
  - Example:
    - `find -name "*.o" -exec rm "{}" ";"`
find Examples

• Find all files beneath home directory beginning with f
  - `find ~ -name 'f*' -print`

• Find all files beneath home directory modified in last day
  - `find ~ -mtime 1 -print`

• Find all files beneath home directory larger than 10K
  - `find ~ -size 10k -print`

• Count words in files under home directory
  - `find ~ -exec wc -w {} \; -print`

• Remove core files
  - `find / -name core -exec rm {} \;`
diff: comparing two files

- **diff**: compares two files and outputs a description of their differences
  - Usage: `diff [options] file1 file2`
  - `-i`: ignore case

```bash
$ diff test1 test2
3c3
< walnuts
---
> grapes
```
Course Outline

- UNIX utilities
Background Jobs

• By default, executing a command in the shell will wait for it to exit before printing out the next prompt
• Trailing a command with & allows the shell and command to run simultaneously

$ /bin/sleep 10 &
[1] 3424
$
Program Arguments

• When a process is started, it is sent a list of strings
  – argv, argc
Ending a process

• When a process ends, there is a return code associated with the process
• This is a positive integer
  – 0 means success
  – >0 represent various kinds of failure, up to process
Process Information Maintained

- Working directory
- Process id
  - number used to identify process
- Process group id
  - number used to identify set of processes
- Parent process id
  - process id of the process that created the process
Environment of a Process

• Common examples:
  – **PATH**: Where to search for programs
  – **TERM**: Terminal type
The PATH environment variable

• Colon-separated list of directories.
• Non-absolute pathnames of executables are only executed if found in the list.
  – Searched left to right

• Example:
  $ myprogram
  sh: myprogram not found
  $ PATH=/bin:/usr/bin:/home/kornj/bin
  $ myprogram
  hello!
Having . In Your Path

$ ls
foo
$ foo
sh: foo: not found

$ ./foo
Hello, foo.
Shell Variables

• Shells have several mechanisms for creating variables. A variable is a name representing a string value. Example: **PATH**
  – Shell variables can save time and reduce typing errors

• Allow you to store and manipulate information
  – Eg: `ls $DIR > $FILE`

• Two types: **local** and **environmental**
  – **local** are set by the user or by the shell itself
  – **environmental** come from the operating system and are passed to children
Variables (con’t)

• Syntax varies by shell
  - `varname=value` # sh, ksh
  - `set varname = value` # csh

• To access the value: `$varname`

• Turn local variable into environment:
  - `export varname` # sh, ksh
  - `setenv varname value` # csh
## Environmental Variables

<table>
<thead>
<tr>
<th>NAME</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>$HOME</td>
<td>Absolute pathname of your home directory</td>
</tr>
<tr>
<td>$PATH</td>
<td>A list of directories to search for</td>
</tr>
<tr>
<td>$MAIL</td>
<td>Absolute pathname to mailbox</td>
</tr>
<tr>
<td>$USER</td>
<td>Your user id</td>
</tr>
<tr>
<td>$SHELL</td>
<td>Absolute pathname of login shell</td>
</tr>
<tr>
<td>$TERM</td>
<td>Type of your terminal</td>
</tr>
<tr>
<td>$PS1</td>
<td>Prompt</td>
</tr>
</tbody>
</table>
Inter-process Communication

Ways in which processes communicate:

• Passing arguments, environment
• Read/write regular files
• Exit values
• Signals
• Pipes
Process Subsystem utilities

- **ps**: monitors status of processes
- **kill**: send a signal to a pid
- **wait**: parent process wait for one of its children to terminate
- **nohup**: makes a command immune to the hangup and terminate signal
- **sleep**: sleep in seconds
- **nice**: run processes at low priority
Pipes

One of the cornerstones of UNIX
Pipes

• General idea: The input of one program is the output of the other, and vice versa

A   B

• Both programs run at the same time
Pipes (2)

- Often, only one end of the pipe is used

- Could this be done with files?
File Approach

• Run first program, save output into file
• Run second program, using file as input

- Unnecessary use of the disk
  - Slower
  - Can take up a lot of space
• Makes no use of multi-tasking
More about Pipes

• Pipes are often chained together
  – Called *filters*
Pipelines

- Output of one program becomes input to another
  - Uses concept of UNIX pipes
- Example: `$ who | wc -l`
  - counts the number of users logged in
- Pipelines can be long
Introduction to Filters

• A class of Unix tools called filters.
  – Utilities that read from standard input, transform the file, and write to standard out

• Using filters can be thought of as data oriented programming.
  – Each step of the computation transforms data stream.

\[ \text{filter} \ < \ abc \ > \ xyz \]
Examples of Filters

- **sort**
  - Input: lines from a file
  - Output: lines from the file sorted

- **grep**
  - Input: lines from a file
  - Output: lines that match the argument
cat: The simplest filter

• The cat command copies its input to output unchanged (identity filter). When supplied a list of file names, it concatenates them onto stdout.

• Some options:
  – -n number output lines (starting from 1)
  – -v display control-characters in visible form (e.g. ^C)

```
cat file*

ls | cat -n
```
head

- Display the first few lines of a specified file
- Syntax: `head [-n] [filename...]`
  - `-n` - number of lines to display, default is 10
  - `filename...` - list of filenames to display
- When more than one filename is specified, the start of each files listing displays
  `==>filename<==`
tail

• Displays the last part of a file
• Syntax: `tail +|-number [lbc] [f] [filename]
  or:   tail +|-number [l] [rf] [filename]
  – `+number` - begins copying at distance `number` from beginning of file, if `number` isn’t given, defaults to 10
  – `-number` - begins from end of file
  – `l,b,c` - `number` is in units of lines/block/characters
  – `r` - print in reverse order (lines only)
  – `f` - if input is not a pipe, do not terminate after end of file has been copied but loop. This is useful to monitor a file being written by another process
head and tail examples

head /etc/passwd
head *.c
tail +20 /etc/passwd
ls -lt | tail -3
head -100 /etc/passwd | tail -5
tail -f /usr/local/httpd/access_log
# Unix Text Files: Delimited Data

<table>
<thead>
<tr>
<th>Tab Separated</th>
<th>Pipe-separated</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>99</td>
</tr>
<tr>
<td>Anne</td>
<td>75</td>
</tr>
<tr>
<td>Andrew</td>
<td>50</td>
</tr>
<tr>
<td>Tim</td>
<td>95</td>
</tr>
<tr>
<td>Arun</td>
<td>33</td>
</tr>
<tr>
<td>Sowmya</td>
<td>76</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Colon-separated</th>
</tr>
</thead>
<tbody>
<tr>
<td>root:ZH0lHAH5zw8As2:0:0:root:/root:/bin/ksh</td>
</tr>
<tr>
<td>jas:nJz3ru5a/44Ko:100:100:John Shepherd:/home/jas:/bin/ksh</td>
</tr>
<tr>
<td>cs1021:iZ3sO9005eZY6:101:101:COMP1021:/home/cs1021:/bin/bash</td>
</tr>
<tr>
<td>cs2041:rX9KwSSPqkLyA:102:102:COMP2041:/home/cs2041:/bin/csh</td>
</tr>
<tr>
<td>cs3311:mLRiClvmtI9O2:103:103:COMP3311:/home/cs3311:/bin/sh</td>
</tr>
</tbody>
</table>
cut: select columns

• The cut command prints selected parts of input lines.
  – can select columns (assumes tab-separated input)
  – can select a range of character positions

• Some options:
  – `-f listOfCols`: print only the specified columns (tab-separated) on output
  – `-c listOfPos`: print only chars in the specified positions
  – `-d c`: use character `c` as the column separator

• Lists are specified as ranges (e.g. 1-5) or comma-separated (e.g. 2,4,5).
cut examples

cut -f 1 < data

cut -f 1-3 < data

cut -f 1,4 < data

cut -f 4- < data

cut -d'|' -f 1-3 < data

cut -c 1-4 < data

Unfortunately, there's no way to refer to "last column" without counting the columns.
The paste command displays several text files "in parallel" on output.

If the inputs are files a, b, c

- the first line of output is composed of the first lines of a, b, c
- the second line of output is composed of the second lines of a, b, c

Lines from each file are separated by a tab character.

If files are different lengths, output has all lines from longest file, with empty strings for missing lines.
paste example

cut -f 1 < data > data1

cut -f 2 < data > data2

cut -f 3 < data > data3

paste data1 data3 data2 > newdata
sort: Sort lines of a file

• The sort command copies input to output but ensures that the output is arranged in ascending order of lines.
  – By default, sorting is based on ASCII comparisons of the whole line.

• Other features of sort:
  – understands text data that occurs in columns. (can also sort on a column other than the first)
  – can distinguish numbers and sort appropriately
  – can sort files "in place" as well as behaving like a filter
  – capable of sorting very large files
sort: Options

- Syntax: `sort [-dftnr] [-o filename] [filename(s)]`
  - `-d` Dictionary order, only letters, digits, and whitespace are significant in determining sort order
  - `-f` Ignore case (fold into lower case)
  - `-t` Specify delimiter
  - `-n` Numeric order, sort by arithmetic value instead of first digit
  - `-r` Sort in reverse order
  - `-o filename` - write output to filename, filename can be the same as one of the input files

- Lots of more options...
sort Examples

sort +2nr < data
sort -k2nr data
sort -t: +4 /etc/passwd
sort -o mydata mydata
uniq: list UNIQue items

• Remove or report adjacent duplicate lines

• Syntax: `uniq [ -cdu] [input-file] [ output-file]`
  – `-c` Supersede the -u and -d options and generate an output report with each line preceded by an occurrence count
  – `-d` Write only the duplicated lines
  – `-u` Write only those lines which are not duplicated

– The default output is the union (combination) of -d and -u
wc: Counting results

• The word count utility, `wc`, counts the number of lines, characters or words

• Options:
  - `-l` Count lines
  - `-w` Count words
  - `-c` Count characters

• Default: count lines, words and characters
wc and uniq Examples

who | sort | uniq -d
wc my_essay
who | wc
sort file | uniq | wc -l
sort file | uniq -d | wc -l
sort file | uniq -u | wc -l
tr: TRanslate Characters

• Copies standard input to standard output with substitution or deletion of selected characters
• Syntax: \textit{tr} \([-cds\) \] \[ \textit{string1}\] \[ \textit{string2} \]
  - \texttt{-d} delete all input characters contained in \textit{string1}
  - \texttt{-c} complements the characters in \textit{string1} with respect to the entire ASCII character set
  - \texttt{-s} squeeze all strings of repeated output characters in the last operand to single characters
tr (continued)

• *tr* reads from standard input.
  – Any character that does not match a character in *string1* is passed to *standard output* unchanged
  – Any character that does match a character in *string1* is translated into the corresponding character in *string2* and then passed to *standard output*

• Examples
  – *tr s z* replaces all instances of *s* with *z*
  – *tr so zx* replaces all instances of *s* with *z* and *o* with *x*
  – *tr a-z A-Z* replaces all lower case characters with upper case characters
  – *tr –d a-c* deletes all a-c characters
tr uses

- Change delimiter
  `tr 'l' ':'`
- Rewrite numbers
  `tr , . ,`
- Import DOS files
  `tr -d '' < dos_file`
- Find printable ASCII in a binary file
  `tr -cd '\n-zA-Z0-9 ' < binary_file`
• Basic UNIX Commands
  – Files: rm, cp, mv, ls, ln
  – Processes: ps, kill
• Unix Filters
  – cat, head, tail, tee, wc
  – cut, paste
  – find
  – sort, uniq
  – comm, diff, cmp
  – tr
• Regular Expressions
  – Allow you to search for text in files
  – `grep` command
• Stream *manipulation*:
  – `sed`
Regular Expressions
What Is a Regular Expression?

• A regular expression (regex) describes a set of possible input strings.
• Regular expressions descend from a fundamental concept in Computer Science called finite automata theory.
• Regular expressions are endemic to Unix
  – vi, ed, sed, and emacs
  – awk, tcl, perl and Python
  – grep, egrep, fgrep
  – compilers
Regular Expressions

- The simplest regular expressions are a string of literal characters to match.
- The string *matches* the regular expression if it contains the substring.
UNIX Tools rocks.

UNIX Tools sucks.

UNIX Tools is okay.
Regular Expressions

• A regular expression can match a string in more than one place.
Regular Expressions

- The . regular expression can be used to match any character.

For me to poop on.

match 1

match 2
Character Classes

- Character classes \([\text{[]}]\) can be used to match any specific set of characters.

```
regular expression \(b\ [\text{eor}]\ a\ t\)
```

```
\textbf{beat} a \textbf{brat} on a \textbf{boat}
```

- match 1
- match 2
- match 3
Negated Character Classes

• Character classes can be negated with the \[^\] syntax.

```
regular expression  \b[^eo]at
```

```
beat a \textbf{brat} on a boat
```
More About Character Classes

- \[\text{aeiou}\] will match any of the characters a, e, i, o, or u
- \[\text{kK}\orn\] will match korn or Korn

• Ranges can also be specified in character classes
  - \([1-9]\) is the same as \([123456789]\)
  - \([\text{abcde}]\) is equivalent to \([\text{a-e}]\)
  - You can also combine multiple ranges
    • \([\text{abcde123456789}]\) is equivalent to \([\text{a-e1-9}]\)
  - Note that the – character has a special meaning in a character class **but only** if it is used within a range, \([-123]\) would match the characters –, 1, 2, or 3
Named Character Classes

• Commonly used character classes can be referred to by name (alpha, lower, upper, alnum, digit, punct, cntrl)

• Syntax [[:name:]]
  - [a-zA-Z] [[:alpha:]]
  - [a-zA-Z0-9] [[:alnum:]]
  - [45a-z] [45[:lower:]]

• Important for portability across languages
Anchors

• Anchors are used to match at the beginning or end of a line (or both).
  • ^ means beginning of the line
  • $ means end of the line
beat a brat on a boat

regular expression: `^ b [eor] a t$

match: beat a brat on a boat

regular expression: `b [eor] a t $`

match: beat a brat on a boat

regular expression: `^ word$`

match: ^word$
Repetition

• The * is used to define **zero or more** occurrences of the *single* regular expression preceding it.
I got mail, yaaaaaaaaaaaaay!

For me to poop on.
Match length

- A match will be the longest string that satisfies the regular expression.
Repetition Ranges

• Ranges can also be specified
  – { } notation can specify a range of repetitions for the immediately preceding regex
  – {n} means exactly n occurrences
  – {n,} means at least n occurrences
  – {n,m} means at least n occurrences but no more than m occurrences

• Example:
  – .{0,} same as .*
  – a{2,} same as aaa*
Subexpressions

• If you want to group part of an expression so that * or {   } applies to more than just the previous character, use ( ) notation

• Subexpressions are treated like a single character
  – a* matches 0 or more occurrences of a
  – abc* matches ab, abc, abcc, abccc, …
  – (abc) * matches abc, abcabc, abcabcabc, …
  – (abc) {2,3} matches abcabc or abcabcabc
grep

- `grep` comes from the `ed` (Unix text editor) search command "global regular expression print" or `g/re/p`.
- This was such a useful command that it was written as a standalone utility.
- There are two other variants, `egrep` and `fgrep` that comprise the `grep` family.
- `grep` is the answer to the moments where you know you want the file that contains a specific phrase but you can’t remember its name.
• `grep` - uses regular expressions for pattern matching
Protecting Regex Metacharacters

- Since many of the special characters used in regexes also have special meaning to the shell, it’s a good idea to get in the habit of single quoting your regexes
  - This will protect any special characters from being operated on by the shell
  - If you habitually do it, you won’t have to worry about when it is necessary
Escaping Special Characters

• Even though we are single quoting our regexs so the shell won’t interpret the special characters, some characters are special to `grep` (eg `*` and `. `)

• To get literal characters, we *escape* the character with a `\` (backslash)

• Suppose we want to search for the character sequence `a*b*`
  – Unless we do something special, this will match zero or more ‘a’s followed by zero or more ‘b’s, *not* what we want
  – `a\*b\*` will fix this - now the asterisks are treated as regular characters
grep: Backreferences

- Sometimes it is handy to be able to refer to a match that was made earlier in a regex
- This is done using backreferences
  - \n  is the backreference specifier, where \n  is a number
- Looks for \n  th subexpression
- For example, to find if the first word of a line is the same as the last:
  - ^\(([[[:alpha:]]){1,}\}) .* \1$
  - The \(([[[:alpha:]]){1,}\}) matches 1 or more letters
grep Family

• Syntax

```
grep [-hilnv] [-e expression] [filename]
```

- `-h` Do not display filenames
- `-i` Ignore case
- `-l` List only filenames containing matching lines
- `-n` Precede each matching line with its line number
- `-v` Negate matches
- `-x` Match whole line only (`fgrep` only)
- `-e expression` Specify expression as option
grep Examples

• grep 'men' GrepMe
• grep 'fo*' GrepMe
Other Notes

• Use `/dev/null` as an extra file name
  – Will print the name of the file that matched
    • `grep test bigfile`
      – This is a test.
    • `grep test /dev/null bigfile`
      – `bigfile:This is a test`.

• Return code of `grep` is useful
  – `grep fred filename > /dev/null && rm filename`
<table>
<thead>
<tr>
<th>Ordinary characters match themselves (NEWLINES and metacharacters excluded)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordinary strings match themselves</td>
</tr>
<tr>
<td>Matches literal character \textit{m} \begin{itemize} \item[] Start of line \item[] End of line \item[] Any single character \item[] Any of \textit{x}, \textit{y}, ^, $, or \textit{z} \item[] Any single character other than \textit{x}, \textit{y}, ^, $, or \textit{z} \item[] \begin{itemize} \item[] \textit{a} - \textit{z} \item[] \textit{r*} \item[] \textit{r1r2} \end{itemize} \end{itemize}</td>
</tr>
<tr>
<td>Matches \textit{r} followed by \textit{r2}</td>
</tr>
<tr>
<td>Tagged regular expression, matches \textit{r}</td>
</tr>
<tr>
<td>Set to what matched the \textit{n}th tagged expression \begin{itemize} \item[] \textit{n} = 1-9 \item[] Repetition \end{itemize}</td>
</tr>
<tr>
<td>One or more occurrences of \textit{r} \begin{itemize} \item[] \textit{r?} \item[] \textit{r1</td>
</tr>
<tr>
<td>Zero or one occurrences of \textit{r} \begin{itemize} \item[] \textit{r1</td>
</tr>
<tr>
<td>Either \textit{r1} or \textit{r2} \begin{itemize} \item[] Either \textit{r1} or \textit{r2} \item[] \textit{r2r1}, \textit{r1r1r2r1},... \end{itemize}</td>
</tr>
</tbody>
</table>

Quick Reference
Sed: Stream-oriented, Non-Interactive, Text Editor

- Look for patterns one line at a time, like `grep`
- *Change* lines of the file
- Non-interactive text editor
  - Editing commands come in as `script`
  - There is an interactive editor `ed` which accepts the same commands
- A Unix filter
  - Superset of previously mentioned tools


- Commands in a `sed` script are applied in order to each line.
- If a command changes the input, subsequent command will be applied to the *modified* line in the pattern space, not the original input line.
- The input file is unchanged (sed is a filter).
- Results are sent to standard output unless redirected.
Scripts

• A script is nothing more than a file of commands
• Each command consists of up to two addresses and an action, where the address can be a regular expression or line number.
Sed Flow of Control

- **sed** then reads the next line in the input file and restarts from the beginning of the script file.
- All commands in the script file are compared to, and potentially act on, all lines in the input file.

```
input  |  script  |  output

cmd 1  |  cmd 2   |  . . .  |  cmd n

Executed if line matches address

print command

only without -n
```
sed Syntax

• Syntax: sed [-n] [-e] ['command'] [file...]  
sed [-n] [-f scriptfile] [file...]
  – `-n` - only print lines specified with the print command (or the ‘p’ flag of the substitute (‘s’) command)
  – `-f scriptfile` - next argument is a filename containing editing commands
  – `-e command` - the next argument is an editing command rather than a filename, useful if multiple commands are specified
  – If the first line of a scriptfile is “#n”, sed acts as though `-n` had been specified
sed Commands

• sed commands have the general form
  – `[address[, address]][!]command [arguments]`

• sed copies each input line into a *pattern space*
  – If the address of the command matches the line in the *pattern space*, the command is applied to that line
  – If the command has no address, it is applied to each line as it enters *pattern space*
  – If a command changes the line in *pattern space*, subsequent commands operate on the modified line

• When all commands have been read, the line in *pattern space* is written to standard output and a new line is read into *pattern space*
Commands

- *command* is a single letter
- Example: Deletion: `d`
- `[address1][,address2]d`
  - Delete the addressed line(s) from the pattern space; line(s) not passed to standard output.
  - A new line of input is read and editing resumes with the first command of the script.
Address and Command Examples

- `d` deletes the all lines
- `6d` deletes line 6
- `/^$/d` deletes all blank lines
- `1,10d` deletes lines 1 through 10
- `1,/^$/d` deletes from line 1 through the first blank line
- `/^$/,$d` deletes from the first blank line through the last line of the file
- `/^$/,10d` deletes from the first blank line through line 10
- `^ya*y/,/[0-9]$d` deletes from the first line that begins with yay, yaay, yaaay, etc. through the first line that ends with a digit
Sed Commands

• Although sed contains many editing commands, we are only going to cover the following subset:

  • s - substitute
  • a - append
  • i - insert
  • c - change
  • d - delete
  • p - print
  • y - transform
  • q - quit
Substitute

- Syntax:
  \([\text{address(es)}]s/\text{pattern}/\text{replacement}/[\text{flags}]\)
  
  - `pattern` - search pattern
  
  - `replacement` - replacement string for pattern
  
  - `flags` - optionally any of the following
    
    - `n` a number from 1 to 512 indicating which occurrence of `pattern` should be replaced
    
    - `g` global, replace all occurrences of `pattern` in pattern space
    
    - `p` print contents of pattern space
Substitute Examples

• **s/Puff Daddy/P. Diddy/**
  - Substitute P. Diddy for the first occurrence of Puff Daddy in *pattern space*

• **s/Tom/Dick/2**
  - Substitutes Dick for the second occurrence of Tom in the *pattern space*

• **s/wood/plastic/p**
  - Substitutes plastic for the first occurrence of wood and outputs (prints) *pattern space*
Replacement Patterns

• Substitute can use several special characters in the *replacement* string
  – `&` - replaced by the entire string matched in the regular expression for pattern
  – `\n` - replaced by the n<sup>th</sup> substring (or subexpression) previously specified using “\(“ and “\)”
  – `\` - used to escape the ampersand (&) and the backslash (\)
Replacement Pattern Examples

"the UNIX operating system ..."
s/.NI./wonderful &/
"the wonderful UNIX operating system ..."

cat test1
  first:second
  one:two
  sed 's/\((.*):\(.*\)/\2:1/' test1
  second:first
  two:one

  sed 's/\(\[[[:alpha:]]\)\(\^[\n]\*[\n]*/\2\lay/g'
    - Pig Latin ("unix is fun" -> "nixuay siay unfay")
Sed Advantages

• Regular expressions
• Fast
• Concise
Sed Drawbacks

• Hard to remember text from one line to another
• Not possible to go backward in the file
• No way to do forward references like / . . . . /+1
• No facilities to manipulate numbers
• Cumbersome syntax
Course Outline

- Scripting languages
What is a shell?

- The user interface to the operating system
- Functionality:
  - Execute other programs
  - Manage files
  - Manage processes
- Full programming language
- A program like any other
  - This is why there are so many shells
Shell History

- There are many choices for shells
- Shell features evolved as UNIX grew
Most Commonly Used Shells

/bin/csh    C shell
/bin/tcsh   Enhanced C Shell
/bin/sh     The Bourne Shell / POSIX shell
/bin/ksh    Korn shell
/bin/bash   Korn shell clone, from GNU
Ways to use the shell

• **Interactively**
  – When you log in, you interactively use the shell

• **Scripting**
  – A set of shell commands that constitute an executable *program*
Review: UNIX Programs

• **Means of input:**
  – Program arguments [control information]
  – Environment variables [state information]
  – Standard input [data]

• **Means of output:**
  – Return status code [control information]
  – Standard out [data]
  – Standard error [error messages]
Shell Scripts

• A shell script is a regular text file that contains shell or UNIX commands
  – Before running it, it must have execute permission:
    • `chmod +x filename`

• A script can be invoked as:
  – `sh name [ arg ... ]`
  – `sh < name [ args ... ]`
  – `name [ arg ... ]`
Shell Scripts

• When a script is run, the kernel determines which shell it is written for by examining the first line of the script
  – If 1st line starts with `#! pathname-of-shell`, then it invokes `pathname` and sends the script as an argument to be interpreted
  – If `#!` is not specified, the current shell assumes it is a script in its own language
    • leads to problems
Simple Example

```
#!/bin/sh

echo Hello World
```
Scripting vs. C Programming

• Advantages of shell scripts
  – Easy to work with other programs
  – Easy to work with files
  – Easy to work with strings
  – Great for prototyping. No compilation

• Disadvantages of shell scripts
  – Slower
  – Not well suited for algorithms & data structures
Simple Commands

- **simple command**: sequence of non-blanks arguments separated by blanks or tabs.
- 1st argument (numbered zero) usually specifies the name of the command to be executed.
- Any remaining arguments:
  - Are passed as arguments to that command.
  - Arguments may be filenames, pathnames, directories or special options.

```
ls -l /
```

```
/bin/ls
-1
/
```
Background Commands

• Any command ending with "&" is run in the background.

  firefox &

• wait will block until the command finishes
Complex Commands

• The shell's power is in its ability to hook commands together
• We've seen one example of this so far with pipelines:

```bash
    cut -d: -f2 /etc/passwd | sort | uniq
```

• We will see others
Redirection of input/output

- **Redirection of output:** `>`
  - example: `$ ls -l > my_files`

- **Redirection of input:** `<`
  - example: `$ cat < input.data`

- **Append output:** `>>`
  - example: `$ date >> logfile`

- **Arbitrary file descriptor redirection:** `fd>`
  - example: `$ ls -l 2> error_log`
Multiple Redirection

• cmd 2>file
  – send standard error to file
  – standard output remains the same
• cmd > file 2>&1
  – send both standard error and standard output to file
• cmd > file1 2>file2
  – send standard output to file1
  – send standard error to file2
Shell Variables

• To set:
  name=value

• Read: $var

• Variables can be local or environment. Environment variables are part of UNIX and can be accessed by child processes.

• Turn local variable into environment:
  export variable
Variable Example

#!/bin/sh

MESSAGE="Hello World"
echo $MESSAGE
<table>
<thead>
<tr>
<th>NAME</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>$HOME</td>
<td>Absolute pathname of your home directory</td>
</tr>
<tr>
<td>$PATH</td>
<td>A list of directories to search for</td>
</tr>
<tr>
<td>$MAIL</td>
<td>Absolute pathname to mailbox</td>
</tr>
<tr>
<td>$USER</td>
<td>Your login name</td>
</tr>
<tr>
<td>$SHELL</td>
<td>Absolute pathname of login shell</td>
</tr>
<tr>
<td>$TERM</td>
<td>Type of your terminal</td>
</tr>
<tr>
<td>$PS1</td>
<td>Prompt</td>
</tr>
</tbody>
</table>
Positional Parameters

• The arguments to a shell script
  – $1, $2, $3 …

• The arguments to a shell function

• Arguments to the `set` built-in command
  – `set this is a test`
    • $1=this, $2=is, $3=a, $4=test

• Manipulated with `shift`
  – `shift 2`
    • $1=a, $2=test

• Parameter 0 is the name of the shell or the shell script.
Example with Parameters

```bash
#!/bin/sh

# Parameter 1: word
# Parameter 2: file

grep $1 $2 | wc -l

$ countlines ing /usr/dict/words
3277
Special Parameters

- `#$` Number of positional parameters
- `$-` Options currently in effect
- `$?` Exit value of last executed command
- `$$` Process number of current process
- `$!` Process number of background process
- `@` All arguments on command line
- `"$@"` All arguments on command line individually quoted "$1" "$2" . . .
Command Substitution

- Used to turn the output of a command into a string
- Used to create arguments or variables
- Command is placed with grave accents ` ` to capture the output of command

```
$ date
Wed Sep 25 14:40:56 EDT 2001
$ NOW=`date`

$ grep `generate_regexp` myfile.c

$ sed "s/oldtext/`ls | head -1`/g"
$ PATH=`myscript`:$PATH
```
File name expansion

- Used to generate a set of arguments from files
- Wildcards (patterns)
  * matches any string of characters
  ? matches any single character
  [list] matches any character in list
  [lower-upper] matches any character in range lower-upper inclusive
  [!list] matches any character not in list
- This is the same syntax that find uses
Boolean Operators

• Exit value of a program (**exit** system call) is a number
  – 0 means success
  – anything else is a failure code

• `cmd1 && cmd2`
  – executes `cmd2` if `cmd1` is successful

• `cmd1 || cmd2`
  – executes `cmd2` if `cmd1` is not successful

```
$ ls bad_file > /dev/null && date
$ ls bad_file > /dev/null || date
Wed Sep 26 07:43:23 2006
```
Control Structures

if expression
then
    command1
else
    command2
fi
What is an expression?

- Any UNIX command. Evaluates to true if the exit code is 0, false if the exit code > 0
- Special command `/bin/test` exists that does most common expressions
  - String compare
  - Numeric comparison
  - Check file properties
- `[]` often a builtin version of `/bin/test` for syntactic sugar
- Good example UNIX tools working together
Examples

```bash
if test "\$USER" = "kornj"
then
    echo "I know you"
else
    echo "I dont know you"
fi
```

```bash
if [ -f /tmp/stuff ] && [ `wc -l < /tmp/stuff` -gt 10 ]
then
    echo "The file has more than 10 lines in it"
else
    echo "The file is nonexistent or small"
fi
```
test Summary

• **String based tests**
  - `z string` Length of string is 0
  - `n string` Length of string is not 0
  - `string1 = string2` Strings are identical
  - `string1 != string2` Strings differ
  - `string` String is not NULL

• **Numeric tests**
  - `int1 -eq int2` First int equal to second
  - `int1 -ne int2` First int not equal to second
  - `-gt, -ge, -lt, -le` greater, greater/equal, less, less/equal

• **File tests**
  - `-r file` File exists and is readable
  - `-w file` File exists and is writable
  - `-f file` File is regular file
  - `-d file` File is directory
  - `-s file` file exists and is not empty

• **Logic**
  - `!` Negate result of expression
  - `-a, -o` and operator, or operator
  - `( expr )` groups an expression
Arithmetic

- No arithmetic built in to /bin/sh
- Use external command /bin/expr
  - expr expression
    - Evaluates expression and sends the result to standard output.
    - Yields a numeric or string result

```
expr 4 "*" 12
expr "( 4 + 3 )" "*" 2
```

- Particularly useful with command substitution
  `x=\`expr $x + 2`\`
Control Structures Summary

• `if ... then ... fi`
• `while ... done`
• `until ... do ... done`
• `for ... do ... done`
• `case ... in ... esac`
for loops

• Different than C:
  ```
  for var in list
  do
    command
  done
  ```

• Typically used with positional parameters or a list of files:
  ```
  sum=0
  for var in "$@
  do
    sum=`expr $sum + $var`
  done
  echo The sum is $sum

  for file in *.c ; do echo "We have $file"
  done
  ```
Case statement

• Like a C switch statement for strings:

```bash
case $var in
  opt1) command1
      command2
      ;;
  opt2) command
      ;;
  *) command
      ;;
esac
```

• * is a catch all condition
Case Example

#!/bin/sh

for INPUT in "$@
do
case $INPUT in
  hello)
    echo "Hello there."
  ;;
  bye)
    echo "See ya later."
  ;;
  *)
    echo "I'm sorry?"
  ;;
esac
done
echo "Take care."
Case Options

- **opt** can be a shell pattern, or a list of shell patterns delimited by `|`

- Example:

  ```bash
  case $name in
      *[0-9]*)
      echo "That doesn't seem like a name."
      ;;
      J*|K*)
      echo "Your name starts with J or K, cool."
      ;;
      *)
      echo "You're not special."
      ;;
  esac
  ```
Types of Commands

All behave the same way

- Programs
  - Most that are part of the OS in `/bin`
- Built-in commands
- Functions
-Aliases
Parsing and Quoting
How the Shell Parses

• Part 1: Read the command:
  – Read one or more lines as needed
  – Separate into *tokens* using space/tabs
  – Form commands based on token types

• Part 2: Evaluate a command:
  – *Expand* word tokens (command substitution, parameter expansion)
  – *Split words into fields*
  – File expansion
  – Setup redirections, environment
  – Run command with arguments
Useful Program for Testing

/home/unixtool/bin/showargs

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i;
    for (i=0; i < argc; i++) {
        printf("Arg %d: %s\n", i, argv[i]);
    }
    return(0);
}
```
Shell Comments

- Comments begin with an unquoted #
- Comments end at the end of the line
- Comments can begin whenever a token begins
- Examples
  
  # This is a comment
  # and so is this
  grep foo bar # this is a comment
  grep foo bar# this is not a comment
Special Characters

- The shell processes the following characters specially unless quoted:
  \| & ( ) < > ; " ' $ ` space tab newline
- The following are special whenever patterns are processed:
  * ? [ ]
- The following are special at the beginning of a word:
  # ~
- The following is special when processing assignments:
  =
Token Types

• The shell uses spaces and tabs to split the line or lines into the following types of tokens:
  – Control operators (||)
  – Redirection operators (<)
  – Reserved words (if)
  – Assignment tokens
  – Word tokens
Operator Tokens

• Operator tokens are recognized everywhere unless quoted. Spaces are optional before and after operator tokens.

• I/O Redirection Operators:
  
  > > > | > & < << <<<< &

  – Each I/O operator can be immediately preceded by a single digit

• Control Operators:

  | & ; ( ) || && ; ;
Shell Quoting

- Quoting causes characters to lose special meaning.
- `\` Unless quoted, `\` causes next character to be quoted. In front of new-line causes lines to be joined.
- `'...'` Literal quotes. Cannot contain `'
- `"..."` Removes special meaning of all characters except $, ", \ and `. The `\` is only special before one of these characters and new-line.
Quoting Examples

```
$ cat file*
  a
  b

$ cat "file*"
  cat: file* not found

$ cat file1 > /dev/null
$ cat file1 ">" /dev/null
  a
  cat: >: cannot open

FILES="file1 file2"
$ cat "FILES"
  cat: file1 file2 not found
```
Simple Commands

• A simple command consists of three types of tokens:
  – Assignments (must come first)
  – Command word tokens
  – Redirections: redirection-op + word-op
  – The first token must not be a reserved word
  – Command terminated by new-line or ;

• Example:
  – `foo=bar` `z=``
    date``
    echo $HOME
    x=foobar > q$$ $xyz z=3`
Word Splitting

• After parameter expansion, command substitution, and arithmetic expansion, the characters that are generated as a result of these expansions that are not inside double quotes are checked for split characters
  • Default split character is space or tab
  • Split characters are defined by the value of the IFS variable (IFS="" disables)
Word Splitting Examples

FILES="file1 file2"
cat $FILES
a
b

IFS=
cat $FILES
cat: file1 file2: cannot open

IFS=x v=exit
echo exit $v "$v"
exit e it exit
Pathname Expansion

• **After** word splitting, each field that contains pattern characters is replaced by the pathnames that match.
• Quoting prevents expansion.
• `set -o noglob` disables.
  – Not in original Bourne shell, but in POSIX.
DATE=`date` echo $foo > /dev/null

DATE=`date` echo $foo > /dev/null

assignment word param redirection

/bin/echo hello there

/language expansion split by IFS

/dev/null
Script Examples

- Rename files to lower case
- Strip CR from files
- Emit HTML for directory contents
#!/bin/sh

for file in *
do
   lfile=`echo $file | tr A-Z a-z`
   if [ $file != $lfile ]
      then
         mv $file $lfile
   fi
done
Remove DOS Carriage Returns

#!/bin/sh

TMPFILE=/tmp/file$

if [ "$1" = "" ]
then
    tr -d '\r'
    exit 0
fi

trap 'rm -f $TMPFILE' 1 2 3 6 15

for file in "@"
do
    if tr -d '\r' < $file > $TMPFILE
    then
        mv $TMPFILE $file
    fi
done
Korn Shell / bash Features
Command Substitution

• Better syntax with $(command)
  – Allows nesting
  – x=$(cat $(generate_file_list))

• Backward compatible with `...` notation
Expressions

• Expressions are built-in with the \[ [ \ ] \] operator
  
  \[
  \text{if} \ \text{[[ \$var = "" \]]} \quad ... \\
  \]

• Gets around parsing quirks of \texttt{/bin/test}, allows checking strings against \texttt{patterns}

• Operations:
  – \texttt{string == pattern}
  – \texttt{string != pattern}
  – \texttt{string1 < string2}
  – \texttt{file1 -nt file2}
  – \texttt{file1 -ot file2}
  – \texttt{file1 -ef file2}
  – \texttt{&&, ||}
Patterns

• Can be used to do string matching:
  
  ```
  if [[ $foo = *a* ]]
  if [[ $foo = [abc]* ]]
  ```

• Similar to regular expressions, but different syntax
Additonal Parameter Expansion

- ${\#param}$ – Length of param
- ${param\#pattern}$ – Left strip min pattern
- ${param\#\#pattern}$ – Left strip max pattern
- ${param\%pattern}$ – Right strip min pattern
- ${param\%\%pattern}$ – Right strip max pattern
- ${param-value}$ – Default value if param not set
Variables

- Variables can be arrays
  - foo[3]=test
  - echo ${foo[3]}

- Indexed by number

- ${#arr} is length of the array

- Multiple array elements can be set at once:
  - set -A foo a b c d
  - echo ${foo[1]}

  - Set command can also be used for positional params: set a b c d; print $2
Printing

- Built-in `print` command to replace `echo`
- Much faster
- Allows options:
  - `-u#` print to specific file descriptor
Functions

• Alternative function syntax:

        function name {
            commands
        }

• Allows for local variables
• $0$ is set to the name of the function
Additional Features

- **Built-in arithmetic:** Using `$(expression )`
  - e.g., `print $(( 1 + 1 * 8 / x ))`
- **Tilde file expansion**
  - `~` $HOME
  - `~user` home directory of user
  - `~+` $PWD
  - `~-` $OLDPWD
Course Outline

- Programming tools
Parsing and Quoting
Shell Quoting

Quoting causes characters to loose special meaning.

• \ Unless quoted, \ causes next character to be quoted. In front of new-line causes lines to be joined.

• '...' Literal quotes. Cannot contain '.

• "..." Removes special meaning of all characters except $, ", \ and `. The \ is only special before one of these characters and new-line.
Quoting Examples

```
$ cat file*
  a
  b
```

```
$ cat "file*"
cat: file* not found
```

```
$ cat file1 > /dev/null
$ cat file1 ">" /dev/null
  a
cat: >: cannot open
```

```
FILES="file1 file2"
$ cat "$FILES"
cat: file1 file2 not found
```
Shell Comments

• Comments begin with an unquoted #
• Comments end at the end of the line
• Comments can begin whenever a token begins
• Examples
  # This is a comment
  # and so is this
  grep foo bar # this is a comment
  grep foo bar# this is not a comment
How the Shell Parses

• Part 1: Read the command:
  – Read one or more lines a needed
  – Separate into tokens using space/tabs
  – Form commands based on token types

• Part 2: Evaluate a command:
  – Expand word tokens (command substitution, parameter expansion)
  – Split words into fields
  – File expansion
  – Setup redirections, environment
  – Run command with arguments
Useful Program for Testing

```
/home/unixtool/bin/showargs

#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    for (i=0; i < argc; i++) {
        printf("Arg %d: %s\n", i, argv[i]);
    }
    return(0);
}"
Special Characters

- The shell processes the following characters specially unless quoted:
  
  $ | & ( ) < > ; " ' $ ` space tab newline

- The following are special whenever patterns are processed:
  
  * ? [ ]

- The following are special at the beginning of a word:
  
  # ~

- The following is special when processing assignments:
  
  =
Token Types

- The shell uses spaces and tabs to split the line or lines into the following types of tokens:
  - Control operators (||)
  - Redirection operators (<)
  - Reserved words (if)
  - Assignment tokens
  - Word tokens
Operator Tokens

• Operator tokens are recognized everywhere unless quoted. Spaces are optional before and after operator tokens.

• I/O Redirection Operators:
  >  >>  >|  >&  <  <<  <<<-  <&
  – Each I/O operator can be immediately preceded by a single digit

• Control Operators:
  |  &  ;  (  )  ||  &&  ; ;
Simple Commands

• A simple command consists of three types of tokens:
  – Assignments (must come first)
  – Command word tokens
  – Redirections: redirection-op + word-op

• The first token must not be a reserved word
• Command terminated by new-line or ;
• Examples:
  – foo=bar  z=`date`
    echo  $HOME
    x=foobar > q$$  $xyz  z=3
Word Splitting

- After parameter expansion, command substitution, and arithmetic expansion, the characters that are generated as a result of these expansions that are not inside double quotes are checked for split characters.
- Default split character is space or tab.
- Split characters are defined by the value of the IFS variable (IFS="" disables).
Word Splitting Examples

FILES="file1 file2"
cat $FILES
a
b

IFS=
cat $FILES
cat: file1 file2: cannot open

IFS=x v=exit
echo exit $v "$v"
exit e it exit
Script Examples

- Rename files to lower case
- Strip CR from files
Rename files

#!/bin/sh

for file in *
do
    lfile=`echo $file | tr A-Z a-z`
    if [ $file != $lfile ]
       then
        mv $file $lfile
    fi
done
Remove DOS Carriage Returns

#!/bin/sh

TMPFILE=/tmp/file$$

if [ "$1" = "" ]
then
    tr -d '\r'
    exit 0
fi

trap 'rm -f $TMPFILE' 1 2 3 6 15

for file in "$@
    do
        if tr -d '\r' < $file > $TMPFILE
            then
                mv $TMPFILE $file
        fi
    done
Korn Shell / bash Features
Command Substitution

• Better syntax with $(command)
  – Allows nesting
  – `x=$(cat $(generate_file_list))`
• Backward compatible with `...` notation
Expressions

- Expressions are built-in with the `[[ ]]` operator
  
  ```
  if [[ $var = "" ]] …
  ```
- Gets around parsing quirks of `/bin/test`, allows checking strings against `patterns`
- Operations:
  - `string == pattern`
  - `string != pattern`
  - `string1 < string2`
  - `file1 -nt file2`
  - `file1 -ot file2`
  - `file1 -ef file2`
  - `&&, ||`
Patterns

• Can be used to do string matching:
  
  ```
  if [[ $foo = *a* ]]
  if [[ $foo = [abc]* ]]
  ```

• Note: patterns are like a subset of regular expressions, but different syntax
Additional Parameter Expansion

- \$\{ \#param \}$ – Length of param
- \$\{ param\#pattern \}$ – Left strip min pattern
- \$\{ param\#\#pattern \}$ – Left strip max pattern
- \$\{ param\%pattern \}$ – Right strip min pattern
- \$\{ param\%%pattern \}$ – Right strip max pattern
- \$\{ param-value \}$ – Default value if param not set
Variables

- Variables can be arrays
  - foo[3]=test
  - echo ${foo[3]}
- Indexed by number
- ${#arr} is length of the array
- Multiple array elements can be set at once:
  - set -A foo a b c d
  - echo ${foo[1]}
  - Set command can also be used for positional params: set a b c d; print $2
Functions

• Alternative function syntax:
  
  function name {
    commands
  }

• Allows for local variables

• $0$ is set to the name of the function
Additional Features

- Built-in arithmetic: Using `$$(((expression))$$`
  - e.g., `print $$((1 + 1 * 8 / x))$$`
- Tilde file expansion
  - `~ $HOME`
  - `~user` home directory of user
  - `~+ $PWD`
  - `~- $OLDPWD`