Section I:
Introduction to Command Line Tools

CEG 333: Introduction to UNIX

Dr. Travis Doom, Associate Professor
Department of Computer Science and Engineering
Wright State University
 Acknowledgements

These slides were developed with the aid of examples found in:
- “Your UNIX” – Sumitabha Das/McGraw Hill
- “A practical guide to Solaris” – Mark G. Sobell
- “Practical UNIX programming” – Robbins & Robbins
Why UNIX?

- For computer scientists, by computer scientists
- Open source
- Runs on (nearly) everything
- Basis for most modern OSes

- Computer Science students everywhere are expected to be able to operate comfortably and program in the UNIX environment
- Data structures, Operating Systems, Distributed Systems, etc.
What is an Operating System?

HARDWARE
- CPU
- Memory
- I/O Devices

APPLICATION PROGRAMS
- Compilers
- Databases
- Games
- Productivity Tools

SOFTWARE

Limited Resources

Many Demands

How do we use the resources?

OS

Limited Resources

Many Demands

How do we use the resources?

OS
What is an Operating System?

Do we want all programs to have access to all instructions?

The OS is a program that acts as an intermediary between the application programs and the hardware resources
  - All communication requires hardware resources, thus the OS is also an intermediary between users and applications

The purpose of any OS is to provide an environment in which:
  - users can (conveniently) execute programs and access data
  - application programs can (efficiently and fairly) access system resources (processor time, memory, file space, I/O devices, etc.)

The OS need not perform any other useful function: it is a control environment (kernel) controls access to all resources
  - All other software is an application program
  - How does the existence of an OS simplify coding an app?
  - Do you trust others to protect your rights and data?
Historic Perspective: 1950’s

- Early Systems were non-interactive single-user systems
  - Input:
    - Card Reader (later: tape drives)
    - Systems had precious little memory - everything needed for the “job”
      had to be included with the set of cards: Control Cards, Program, Data, etc.
  - Output:
    - Card Printer (later: line printers)
    - Results of program or memory dump

- Fairly simple OS (Resident Monitor)
  - Only task: transfer control from one job to the next
  - Always resident in memory
  - Secure (no sharing issues!)

- Problems? OS rereads program with every job.
Simple Batch Systems

- How can we better utilize the limited hardware resources?
- Reduce setup time by “batch”-ing similar jobs
  - Hire an operator to sort input/output cards
- First rudimentary operating system
  - Initial control in monitor, always in memory (resident)
  - Automatic job sequencing: automatically transfers control from one job to another.
    - When job completes control transfers back to monitor
  - Control card interpreter – responsible for reading and carrying out instructions on the cards.
  - Loader – loads systems programs and applications programs into memory.
  - Device drivers – know special characteristics and properties for each of the system’s I/O devices.
Simple Batch Systems

- Problem: Slow Performance – I/O and CPU could not overlap; card reader very slow.
- Solution: Off-line operation – speed up computation by loading jobs into memory from tapes and card reading and line printing done off-line.
  - Remote Job Entry
  - Specialized front-end and back-end systems
- Better Solution: Spooling - Simultaneous Peripheral Operation On-Line
  - Faster I/O devices (disk drives) allow the input and output to be buffered on-line
Simple Batch Systems

- With Spooling:
  - The CPU can perform three tasks simultaneously: (1) output Job#1 from disk to output device; (2) process Job#2 from disk to disk; (3) input Job#3
  - Cost: Disk space, administration of disk space by OS
  - Job pool – data structure that allows the OS to select which job to run next in order to increase CPU utilization.
Problem: In general, process execution consists of a cycle of CPU execution (CPU burst) and I/O wait (I/O burst). How can we more efficiently utilize the CPU?

Solution: Several jobs are kept in main memory at the same time, and the CPU is multiplexed among them

- The CPU is never idle (when there are jobs ready to run)
- When one job becomes I/O dependent, it is swapped out by the OS and another job starts

Cost: Complexity of the OS (and CPU overhead)

- CPU Scheduling (Fairness, Starvation)
- Resource Allocation (Deadlock)
- Memory Management (Security)
- I/O routine provided by the system
Time-Sharing Systems – Interactive Computing

- **Problem:** Batch systems with Multiprogramming are efficient from the CPUs point of view, but not necessarily from the users
  - Non-batch systems had a single user at the console
  - Batch systems had an operator at the console, all user interaction must be handled a priori via control cards
    - Consider the effect on multi-step jobs (compile and execute)
    - Debugging is static (from dumps), no tracing
    - Programmers fear to experiment
- **Solution:** Use multiple I/O devices (CRT, Keyboard) and timeshare.
  - The CPU is multiplexed among several jobs that are kept in memory and on disk (the CPU is allocated to a job only if the job is in memory)
Time-Sharing Systems – Interactive Computing

- On-line communication between the user and the system is provided; when the operating system finishes the execution of one command, it seeks the next “control statement” not from a card reader, but rather from the user’s keyboard

- **Cost:** A multitude of “on-line” OS chores
  - On-line file system (with human friendly names/directories) must be available for users to access data and code
  - Security
  - Fairness? How do we handle resource limitations?
Personal Computer Systems

- **Personal computers** – computer system dedicated to a single user
  - Affordable due to decreasing hardware costs
  - I/O devices – keyboards, mice, display screens, small printers.
- New OS Goals
  - Can adopt technology developed for larger operating system
  - User convenience and responsiveness valued at the price of efficiency
  - Often individuals have sole use of computer and do not need advanced CPU utilization of protection features.
    - Multitasking?
    - Security?
Why an Operating System (OS)

• OS interacts with hardware and manages programs.

• Programs not expected to know which hardware they will run on.

• Must be possible to change hardware without changing the programs.

• Programs can’t manage themselves.

• OS provides a safe environment for programs to run.
How a Program Runs on a Computer

- OS loads program from disk and allocates memory and CPU.
- Instructions in program are run on CPU and OS keeps track of last instruction executed.
- If program needs to access the hardware, OS does the job on its behalf.
- OS saves the state of the program if program has to leave CPU temporarily.
- OS cleans up memory and registers after process has completed execution.
Getting started

- Log in
- Editing files with vi, emacs, pico
- Printing (lpr –Pecs_russ1 <filename>)
- man (man <command name>)

Warning: EVERYTHING in UNIX is case-sensitive!

Exiting:
- ^D End of data stream; EOF/EOT; exit/logoff
- ^C Interrupt
- Logout Leave the system
- Exit Leave the shell
Key Concepts

- Everything in the system is represented as a file.
- Work gets done by processes.
- Workload shared by two separate programs (kernel and shell).
- Kernel uses system calls to do most of the work.
- All UNIX systems use the same system calls.
- C and UNIX
UNIX Architecture: The Kernel

- Program always resides in memory.
- Has direct access to the hardware.
- Handles file I/O.
- Manages processes.
- Only one copy shared by all users.
UNIX Architecture: The Shell

- A program or command invoked only when the user logs in.
  - “owned” by user
- Accepts user input, examines and rebuilds the command line.
- Makes calls to the kernel for all other functions.
- At least one shell is invoked by every user.
- User has a choice of shells.
Structure of a Command

Command [-option 1] [argument 1] [-option 2] [argument 2] ...

e.g.  ls -l

• Command filenames need no specific extensions.

• A command’s behavior is determined by its arguments and options.

• Command and arguments must be separated by whitespace.

• Generally possible to combine multiple options into a single one (like 
  ls -l -u -t == ls -lut)

• Order of combining is generally not important (like ls -lut == ls -utl)

• Case sensitive!
Types of Commands

- External program on disk which could be:
  - a binary executable (written in C, C++).
  - a script file (like a shell or perl script).

- Internal command of the shell which could be
  - a builtin (like cd, pwd, etc.)
  - an alias defined by the user that invokes the disk or internal version in a specific manner.
UNIX identity

- The system identifies you by user and group numbers (UID and GID)
  - Automatically translated by the OS

- Commands of interest
  - id, groups, whoami, pwd
Using man

• Displays documentation of commands, configuration files, system calls and library functions.

• Organized in a number of sections. Commands are found in Section 1.

• May need to use section number when entry exists in multiple sections (e.g. `man passwd` and `man -s 5 passwd`).

• man documentation not available for most internal commands of the shell.

• Use `man man` first to know how `man` should be used.
Understanding a man page

Example: **wc** Syntax/Synopsis

```
wc [ -c | -m | -C ] [ -lw ] [ file ... ]
```

- Most useful information available in SYNOPSIS and DESCRIPTION.

- When options grouped in [ ] without a |, one or more of them can be used. (-l, -w and -lw are valid.)

- The | signifies an OR condition. (Only one of -c, -m or -C can be used.)

- The ... means that multiple occurrences of the preceding item are possible. (**wc** can be used with multiple files.)

- EXIT STATUS indicates values returned on error.
# File display commands

<table>
<thead>
<tr>
<th>Command/Syntax</th>
<th>What it will do</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cat [options] file</code></td>
<td>concatenate (list) a file</td>
</tr>
<tr>
<td><code>echo [text string]</code></td>
<td>echo the text string to stdout</td>
</tr>
<tr>
<td><code>head [-number] file</code></td>
<td>display the first 10 (or number of) lines of a file</td>
</tr>
<tr>
<td><code>more [options] file</code></td>
<td>page through a text file</td>
</tr>
<tr>
<td><code>tail [options] file</code></td>
<td>display the last few lines (or parts) of a file</td>
</tr>
</tbody>
</table>
The Hierarchical Structure of the File System

- A single hierarchical structure that contains all files.
- Top signified by root (/).
- Parent of any file must be a directory.
- Files accessed with pathnames (like /etc/passwd).
Pathnames: Two Types

- **Absolute pathname**: Specifies location with reference to the file system top (like `cat /etc/passwd`).

- **Relative pathname**: Specifies location with reference to the user’s current location (like `cd ../include`).

- Both commands and filename arguments can be represented in either form.
Absolute Pathname

• Begins with a / (like /etc/passwd).

• First / signifies the root directory.

• System configuration files that normally don’t change location should be addressed in absolute manner.

• Used with a command that
  • doesn’t feature in PATH.
  • resides in two or more directories of PATH.
Relative Pathname

• Uses . to signify the current directory.

• Uses .. to signify the parent directory.

• Used to refer to files that are either impossible or inconvenient to access in an absolute manner.

• Has a synonym for a filename argument that doesn’t have a /. (cat foo is the same as cat ./foo.)

• Same synonym doesn’t automatically exist for commands. (cat foo MAY NOT be the same as ./cat foo.)
The UNIX File

- Filename limited to 255 characters. Can’t contain / or NULL character.
  - Like C strings, quotes can be used as delimiters

- Filenames are case-sensitive; chap and Chap are two different filenames.

- Group of filenames held together in a directory.

- Directory contains name of the file.

- Both files and directories are subject to access control.
File Types

- **Ordinary** or **regular** file: Contains data. This file can be a
  - text file (program sources, configuration files).
  - binary file (executables, graphic and multimedia files).

- **Directory**: Contains the filename and a number (inode number).

- **Device file**: Contains no data whatsoever.

- **Symbolic link**: Contains the location of another file.
The Home Directory

- Directory where user is placed on login.

- Determined by sixth field in /etc/passwd:

  w001ted:x:26845:100000:TravisDoom:/common/users2/cse/w001ted:/bin/tcsh

- Can also be referred to by:
  - the shell variable $HOME (e.g. cat $HOME/foo).
  - tilde expansion in most shells: ~ (e.g. cat ~/foo).

- cd command used without arguments returns user to home directory.

- User can create and remove files in their home directory but not in other directories (by default)
# Directory navigation and control

<table>
<thead>
<tr>
<th>Command/Syntax</th>
<th>What it will do</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cd [directory]</code></td>
<td>change directory</td>
</tr>
<tr>
<td><code>ls [options] [directory or file]</code></td>
<td>list <code>directory</code> contents or <code>file</code> permissions</td>
</tr>
<tr>
<td><code>ls -l</code></td>
<td>list long (show permissions)</td>
</tr>
<tr>
<td><code>ls -a</code></td>
<td>list all (show dot-files)</td>
</tr>
<tr>
<td><code>mkdir [options] directory</code></td>
<td>make a <code>directory</code></td>
</tr>
<tr>
<td><code>pwd</code></td>
<td>print working (current) directory</td>
</tr>
<tr>
<td><code>rmdir [options] directory</code></td>
<td>remove a <code>directory</code></td>
</tr>
</tbody>
</table>
The Inode

- System of organizing file attributes separately from content.
- Identified by **inode number** but inode doesn't contain this number.
- Inode number displayed by `ls -i`.
- Both inode and directory entries are looked up by inode number.
- Possible to consume all inodes even when there is adequate disk space.
File Attributes Stored in Inode

• **Type:** Whether ordinary, directory, device, etc.

• **Permissions:** Determines who can read, write or execute a file.

• **Links:** Number of names a file can have. A program can be designed to behave differently depending on the name by which it is invoked.

• **Owner:** A file is owned by a user, by default its creator. The owner can change many file attributes and set the permissions.

• **Group Owner:** The group which owns the file. The owner by default belongs to this group.

• **File Size:** Number of bytes of data contained.

• **File Time Stamps:**
  • Date and time of last modification
  • Date and time of last access
File Permissions

- A file has three types of permissions (read, write and execute).
- Available to three categories of users (user, group and others).
- Only file owner or superuser can change file permissions.
- Permissions can be assigned or removed in
  - relative manner (e.g. `chmod +x foo.sh`).
  - absolute manner (e.g. `chmod 744 foo.sh`).
- Note: the absolute manner is the bit-vector as stored in the i-node
- Significance of permissions different for file and directory.
Directory Permissions

- Read permission: Whether filenames in directory can be listed by a program (like `ls`).

- Write permission: Whether files and directories can be created in the directory.

- Execute or search permission: Whether one can pass through directory to search for filenames.

- Desirable permission setting: 755
# How a Directory Influences File Permissions

Examining only the user category

<table>
<thead>
<tr>
<th>File</th>
<th>Directory</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>r--r--r--</td>
<td>rwxr-xr-x</td>
<td>A write-protected file; can’t be modified but can be removed.</td>
</tr>
<tr>
<td>rw-r--r--</td>
<td>r-xr-xr-x</td>
<td>A write-protected directory; file can’t be removed but can be modified.</td>
</tr>
<tr>
<td>r--r--r--</td>
<td>r-xr-xr-x</td>
<td>A write-protected file and directory; file can’t be modified or removed.</td>
</tr>
<tr>
<td>rw-r--r--</td>
<td>rwxr-xr-x</td>
<td>Normal setting; file can be modified and removed.</td>
</tr>
<tr>
<td>rw-r--r--</td>
<td>rw-r-xr-x</td>
<td>File can’t be removed even though directory is writable. (An unusual setting)</td>
</tr>
</tbody>
</table>
An Ownership-Permissions Problem

Assumption: romeo and juliet belong to the users group.
Ownership and Permissions of File foo and its Directory

$ who am i
romeo
$ ls -l foo
-r-x-w-r-x 1 juliet users 7017 2005-09-14 13:53 foo
$ ls -ld .

Note: foo is owned by juliet but directory is owned by romeo.

juliet:
• can’t edit foo without changing the permissions.
• can change permissions (as owner) and then edit foo.
• can’t delete foo (directory write-protected for group).

romeo:
• can edit or delete foo.
• can’t change permissions of foo.
• can’t display or copy foo.
# File maintenance commands

<table>
<thead>
<tr>
<th>Command/Syntax</th>
<th>What it will do</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cp [options] file1 file2</code></td>
<td>copy <code>file1</code> into <code>file2</code>; <code>file2</code> shouldn't already exist. This command creates or overwrites <code>file2</code>.</td>
</tr>
<tr>
<td><code>mv [options] file1 file2</code></td>
<td>move <code>file1</code> into <code>file2</code></td>
</tr>
<tr>
<td><code>rm [options] file</code></td>
<td>remove (delete) a file or directory (-r recursively deletes the directory and its contents) (-i prompts before removing)</td>
</tr>
<tr>
<td><code>chmod [options] file</code></td>
<td>change file or directory access permissions</td>
</tr>
<tr>
<td><code>chgrp [options] group file</code></td>
<td>change the group of the file</td>
</tr>
<tr>
<td><code>chown [options] owner file</code></td>
<td>change the ownership of a file; can only be done by the superuser</td>
</tr>
</tbody>
</table>
File Systems

- System of organizing files into multiple manageable units.

- Each file system has a separate directory structure with a top.

- For a file to be visible, its file system must be attached to the main file system.

- Two files in two file systems *may* have the same inode number.

- Not easy to understand whether a directory structure comprises multiple file systems.
**(Hard) Links**

- Mechanism by which a file is allowed to have multiple names.
- Linked filenames share inode but have separate directory entries.
- Each link increments link count in inode by 1 and adds an entry to the directory.
- File considered to be deleted and inode freed only when link count drops to 0.
- Linked filenames equivalent in all respects.
(Hard) Links - 2

• Advantages:

  • Backup: Prevention from accidental deletion.

  • Allows the same file to be executed as two similar but separate programs.

  • Takes care of old programs that accesses a file whose name or location has changed.

• Disadvantages:

  • Can’t link directories.

  • Can’t link across file systems.
Symbolic Links (Symlinks)

- Separate file type and having its own inode.
- Contains the pathname of another file or directory.
- Can link across file systems.
- Link and file pointed to are not equivalent.
- Pathname may be stored either in inode (or in a separate file).
## Print commands

<table>
<thead>
<tr>
<th>Command/Syntax</th>
<th>What it will do</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lpr (lp)</strong> [options] file1 ...</td>
<td>add to defined (or default) print queue.</td>
</tr>
<tr>
<td><strong>lpq (lpstat)</strong> [options]</td>
<td>show the status of print jobs.</td>
</tr>
<tr>
<td><strong>lprm (cancel)</strong> [options]</td>
<td>remove a print job from the print queue.</td>
</tr>
<tr>
<td><strong>enscript</strong> [options]</td>
<td>create complex printout (as postscript)</td>
</tr>
</tbody>
</table>

- **Useful examples**
  - `lpstat -a`
  - `lpr -c -Pecs_russ1 filename` (-c copies to queue before printing)
  - `lpstat -Pecs_russ1`
  - `lprm -Pecs_russ1 (jobnumber | username)`
  - `enscript -2rG -Pecs_russ1 filename.c`
* Write a simple perl script to parse a genbank file (via gsub)
* and produce the string of translated amino acids that are identified in
* the file

```perl
#input = open($input);
chomp($input);

foreach $line (@input) {
    if ($intranslatedregion eq 0) {
        if ($line =~ /translating\s*:\s*\{\s*\\d\d\s*\}/) {
            $polypeptide = $1;
            $intranslatedregion = 1;
        } else {
            if ($line =~ /\s*\\\d\d\s*\}/) {
                $polypeptide = $polypeptide . $1;
            } else {
                $polypeptide = $polypeptide . $line;
            }
        } # end if
    } # end foreach
}

# remove spaces and tabs from scalar
$polypeptide =~ s/\s+/\//g;

# print so characters/line
while ($position <= length($polypeptide)) {
    print substr($polypeptide, $position, 8);
    print \n
    $position += 8;
} # end while
```
# System resources

<table>
<thead>
<tr>
<th>Command/Syntax</th>
<th>What it will do</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpr (lp) [options] file1 ...</td>
<td>add to defined (or default) print queue.</td>
</tr>
<tr>
<td>df [options] [resource]</td>
<td>report disk block/inode usage</td>
</tr>
<tr>
<td>du [options] [directory or file]</td>
<td>report amount of disk space in use</td>
</tr>
<tr>
<td>uname [options]</td>
<td>display name of machine</td>
</tr>
<tr>
<td>whereis [options] command</td>
<td>report location for named command</td>
</tr>
<tr>
<td>which command</td>
<td>report path to command or alias</td>
</tr>
<tr>
<td>who (w)</td>
<td>report who is logged in/running processes</td>
</tr>
</tbody>
</table>
The Shell

• Program that constantly runs at terminal after a user has logged in
• Interprets command line and makes arrangements for its execution.
• Generally waits for command to complete execution.
• Some commands are built into the shell.
• Killed on logging out.
Shells

- The shell sits between the user and the OS
- Program that constantly runs at terminal after a user has logged in
- Interprets command line and makes arrangements for its execution
- Killed on logging out.

- **sh**
  - The original shell was the Bourne shell (default prompt $)
  - Good I/O features, but not “user friendly”

- **csh**
  - Developed to be user friendly (easy command editing, command history, job control, etc.) (default prompt %)
  - I/O can be awkward

- Other shells: bash (Bourne Again), ksh (Korn), tcsh, cshe (extended)
# Common built-in shell functions

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias/unalias</td>
<td>assign/unassign a name to a function</td>
</tr>
<tr>
<td>echo</td>
<td>write a string to stdout</td>
</tr>
<tr>
<td>foreach</td>
<td></td>
</tr>
<tr>
<td>history</td>
<td>print command history</td>
</tr>
<tr>
<td>cd</td>
<td>change working directory (runs alias cwdcmd)</td>
</tr>
<tr>
<td></td>
<td>cd - returns to previous directory</td>
</tr>
<tr>
<td>pushd</td>
<td>push PWD on directory stack and then cd</td>
</tr>
<tr>
<td>popd</td>
<td>cd to top directory on directory stack</td>
</tr>
<tr>
<td>source</td>
<td>execute shell commands stored in a file</td>
</tr>
</tbody>
</table>
**Environment variables**

- Used to provide information to the programs that you use

**Command**

- **printenv:** Display all current shell variables
- **Setenv NAME value** Set shell variable
- **set variable=value** Set temporary/local variable

**Common Variables**

- **DISPLAY** Which graphical display to use, e.g. doom:0.0
- **EDITOR** Your default editor, e.g. /usr/bin/vi
- **HOME** Your home directory
- **PATH** Path to be searched for commands
The PATH

- A shell variable (or environment variable) that specifies a list of directories to search.

- Shell looks at PATH only when command is not used with a pathname and is also not a shell builtin.

- Command can still be executed if not in PATH by
  - Using a pathname.
  - Modifying PATH to include the directory containing the command.

- PATH can be modified in an absolute or relative manner:
  - `setenv PATH=/usr/bin:`      (Absolute)
  - `setenv PATH=$PATH:/usr/local/bin`     (Relative)

- Modified setting is lost after user has logged out unless saved in a startup file.
Command environment

- Tab Can be set up to do filename completion
- Arrow keys Can be set up to scroll through history

- Set noclobber
- Set filec
- Set autolist matchbeep=nomatch autoexpand autocorrect
- Set history=128 savehist=1
## Command history/editing

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>history n</code></td>
<td>Display last n commands (up to max set history)</td>
</tr>
<tr>
<td><code>!n</code></td>
<td>Repeat command number n</td>
</tr>
<tr>
<td><code>!-n</code></td>
<td>Repeat command n from last</td>
</tr>
<tr>
<td><code>!!</code></td>
<td>Repeat last command (same as <code>!-1</code>)</td>
</tr>
<tr>
<td><code>!str</code></td>
<td>Repeat last command that started with str</td>
</tr>
<tr>
<td><code>!?str?</code></td>
<td>Repeat last command that included str</td>
</tr>
<tr>
<td><code>!</code></td>
<td>Repeat first argument from last command</td>
</tr>
<tr>
<td><code>!:n</code></td>
<td>Repeat nth argument from last command</td>
</tr>
<tr>
<td><code>!:n-m</code></td>
<td>Repeat n-mth arguments from last command</td>
</tr>
<tr>
<td><code>!$</code></td>
<td>Repeat last argument from last command</td>
</tr>
<tr>
<td><code>^str1^str2</code></td>
<td>replace str1 with str2 in last command</td>
</tr>
<tr>
<td><code>!n:s/str1/str2/</code></td>
<td>substitute str1 with str2 in command n, use g for global</td>
</tr>
</tbody>
</table>
Wildcards

- The shell allows meta-characters (a.k.a wild cards) which are replaced with pattern matches
- For filenames the meta-characters are:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Any single character</td>
</tr>
<tr>
<td>*</td>
<td>Any string of zero or more characters</td>
</tr>
<tr>
<td>[abc...]</td>
<td>Any one of the enclosed characters</td>
</tr>
<tr>
<td>[a-e]</td>
<td>Any one character in the enclosed range</td>
</tr>
<tr>
<td>![def]</td>
<td>Any one character NOT in the enclosed range</td>
</tr>
<tr>
<td>{abc,dcd,cde}</td>
<td>Any one element of the set</td>
</tr>
<tr>
<td>~</td>
<td>Home directory of current user</td>
</tr>
<tr>
<td>~user</td>
<td>Home directory of specified user</td>
</tr>
</tbody>
</table>
Wild-card examples

<table>
<thead>
<tr>
<th>Wild-card</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Any number of characters including none</td>
</tr>
<tr>
<td>ls *.lst</td>
<td>Lists all files with extension .lst.</td>
</tr>
<tr>
<td>?</td>
<td>A single character</td>
</tr>
<tr>
<td>rm ??*</td>
<td>Removes all files comprising at least 2 characters.</td>
</tr>
<tr>
<td>[ch]</td>
<td>A single character that is either a c or h</td>
</tr>
<tr>
<td>cp *.[ch] cprogs</td>
<td>Copies all files with .c or .h extension.</td>
</tr>
<tr>
<td>![ch]</td>
<td>A single character that is not a c or h</td>
</tr>
<tr>
<td>rm <em>![a-zA-Z]</em></td>
<td>Removes files not containing at least one letter.</td>
</tr>
<tr>
<td>ls .??*</td>
<td>Lists all filenames beginning with a dot and comprising at least two more characters.</td>
</tr>
</tbody>
</table>
## Other special command symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>Command separator</td>
</tr>
<tr>
<td>&amp;</td>
<td>run command in background</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND – Run command only if previous succeeds</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>()</td>
<td>start new shell to run as separate process</td>
</tr>
<tr>
<td><code>command </code></td>
<td>Replace with the output of the command</td>
</tr>
<tr>
<td>#</td>
<td>Everything that follows (until newline) is a comment</td>
</tr>
<tr>
<td>$name</td>
<td>Shell variable</td>
</tr>
</tbody>
</table>

- " " Disable most special/meta characters in string (not $ or \)
- `' ' Disable all special characters
- \ Take the next character literally (escape sequence)

Also: At beginning of command suppresses aliasing
## Examples of Shell Behavior

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cat chap*</code></td>
<td>Shell expands * to match all filenames in the current directory that begin with chap.</td>
</tr>
<tr>
<td><code>date &gt; foo</code></td>
<td>Shell sees the &gt; first, opens the file foo and connects the <code>date</code> output to it.</td>
</tr>
<tr>
<td>`who</td>
<td>sort`</td>
</tr>
<tr>
<td><code>ls </code>cat foo`</td>
<td>Shell first runs <code>cat</code> and supplies the output as arguments to <code>ls</code>.</td>
</tr>
<tr>
<td><code>echo $HOME</code></td>
<td>Evaluates <code>$HOME</code> as a variable before running <code>echo</code>.</td>
</tr>
</tbody>
</table>
Escaping (Using a \ before a character)

- Reverses usual meaning of metacharacter following it. (\rm \* removes a file named \*.)

- Can also protect itself. (echo \ prints a \.)

- Protects space and [Enter]. (cd My\ Documents will work.)

- Inconvenient to use when command line contains too many metacharacters that need to be escaped.

- Principle also used by commands in their expressions. (grep "\." foo looks for a dot in foo.)
Quoting

• Protects most metacharacters from interpretation by the shell. (echo “*” prints a *.)

• More convenient than escaping when protecting a group of metacharacters.

• Quoted string understood as a single argument by shell and C programs. (a.out foo “My Documents” has 2 arguments and not 3.)

• Double quotes and single quotes are not equivalent. (echo “$SHELL” not the same as echo ‘$SHELL’)

• Quoting doesn’t protect the \; escaping is also required.
Single Quotes or Double Quotes?

• Single quotes protect all characters except \. (echo ‘\’ won’t work.)

• Double quotes protect all characters except the \, $ and ` (echo “$” doesn’t print a $.)

• Single quotes protect the “.

• Double quotes protect the ‘.

• Double quotes permit variable evaluation and command substitution.
Command Substitution

• Allows command arguments to be obtained from standard output of another command.

• In `command1 `command2``, command2 is run first and its standard output used as arguments to command1.

• Command enclosed by `` must write to standard output.

• Convenient mechanism for running commands whose arguments are known only at runtime.

• Enabled within double quotes but not in single quotes.
How the Shell Handles Files

• A file is opened by accessing it by its pathname.

• Opening returns a **file descriptor** (an integer).

• Subsequent read/write operations on the file use the descriptor.

• Kernel allocates **lowest** available number as descriptor.

• First three descriptors (0, 1 and 2) are always allocated.
Data transfer is standardized in UNIX; all I/O is file based, always is a standardized manner.

File Descriptors: In addition to “disk” files, there are three standard file descriptors used by all UNIX programs:

<table>
<thead>
<tr>
<th>Name</th>
<th>Filehandle</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdin</td>
<td>0</td>
<td>Standard input from program</td>
<td>keyboard</td>
</tr>
<tr>
<td>stdout</td>
<td>1</td>
<td>Standard output from program</td>
<td>display</td>
</tr>
<tr>
<td>stderr</td>
<td>2</td>
<td>Standard error output from program</td>
<td>display</td>
</tr>
</tbody>
</table>

This standardized handling of data support to key features:

- Output redirection: The output of a command is sent to a (disk) file rather than the display (file).
- Pipeing: The output of a command is sent immediately as input to another command.
File redirection

- Output redirection takes the output of a command and places it into a named file.
- Input redirection reads a file as input to a command.
- Commands have no knowledge of the redirection.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Redirection</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Stdout redirected, create new file (subject to noclobber)</td>
</tr>
<tr>
<td>&gt;!</td>
<td>Stdout redirected, destroys existing file (ignores noclobber)</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Append stdout to existing file (if any) or creates (otherwise)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>Stdin redirection</td>
</tr>
<tr>
<td>&gt;&amp;</td>
<td>Redirect stdout and stderr</td>
</tr>
<tr>
<td>&gt;&gt;&amp;</td>
<td>Append stdout and stderr</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
</tr>
</tbody>
</table>
Common Destinations for StdOut

- Allow it to come to the terminal.
- Redirect it with > and >>.
- Merge it with standard error using >&.
- Force output to /dev/null.
- Force output to /dev/tty.
- Connect it to standard input of another program.
Pipes

- Connects standard output on one command to standard input of another.
- Takes care of flow control; reader and writer work in unison.
- No temporary file is created.
- Used to solve text manipulation problems.
- Commands have no knowledge that they are reading from or writing to pipe.
### Some useful commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cut</td>
<td>cut specified fields/characters from lines in a file</td>
</tr>
<tr>
<td>diff</td>
<td>compare two files and display differences</td>
</tr>
<tr>
<td>grep</td>
<td>find lines that contain a word/pattern/regex</td>
</tr>
<tr>
<td>file</td>
<td>classify the file type</td>
</tr>
<tr>
<td>find</td>
<td>find file matching a pattern</td>
</tr>
<tr>
<td>sort</td>
<td>sort the lines of a file</td>
</tr>
<tr>
<td>tee</td>
<td>copy stdout to one or more files</td>
</tr>
<tr>
<td>tr</td>
<td>translate strings in stdout</td>
</tr>
<tr>
<td>uniq</td>
<td>remove repeated lines in a file</td>
</tr>
<tr>
<td>wc</td>
<td>display word (or character or line) count for input</td>
</tr>
<tr>
<td>dos2unix / unix2dos</td>
<td>Changes CR to CR/NL and visa-versa</td>
</tr>
</tbody>
</table>
Section II: The UNIX programming environment

CEG 333: Introduction to UNIX

Dr. Travis Doom, Associate Professor
Department of Computer Science and Engineering
Wright State University
Preprocessing phase: Modifies original program according to preprocessor directives (these start with the # character). The result is another C program text file (typically with the .i suffix)
  - #include <stdio.h>
  - #define FALSE 0

Compilation phase: text file converted from high-level language to assembly.
  - Regardless of the original high-level language, all programs handled identically from this point onward
  - One assembly instruction corresponds one-to-one with a machine instruction
From C to executable

- Assembly phase: assembly text file converted into machine language binary file and packaged into a relocatable object program.
- Linker phase: multiple object programs are merged to result in an executable object file.
  - For example: a standard library function such as printf might reside in a separate precompiled object file (like printf.o) that exists elsewhere in the system.
C and C++ in UNIX

- C files
  - VERY common in UNIX software
  - Default extension *.c
  - Can compile with gcc
- C++
  - Default extensions include: *.C, *.cc, *.c++
  - Can compile with g++
- Default target executable is a.out
- Default crash/core dump is core
  - Good command to know (man csh): limit coredumpsize 0M
UNIX Programming in C/C++

- Include files are loaded by the preprocessor using the `#include` directive.
  - Angle brackets are used for include files located in the “standard” location (generally `/usr/include`)
    - `#include <stdio.h>`
  - Double quotes are used for other locations:
    - `#include "../common/users2/cse/w001ted/soft/include/doomC.h"`
  - Another way to specify directories to be searched for header files is to use the `-I` option to the C compiler
    - `#include "doomC.h"`
    - `gcc -I ~/w001ted/soft/include`

- `#define`: used to define symbolic constants (a macro)
  - Provides mapping from symbolic name to replacement text (macro expansion). Improves readability and modification.
    - `#ifndef ALLOC
    #define ALLOC(type,num) ((type *) malloc(sizeof(type) * (num)))
    #endif`
Programming Tools: gcc

- **C (C++) compilers**
  - cc (CC): SunWorkShop Compiler
  - gcc (g++): GNU C Compiler, freeware, cross-platform, ubiquitous

- **Compiler flags**
  - man gcc for details!
  - -l: specify library on command line (must come after all modules to which it applies)
  - -L: specify additional directories to be searched for library (default /usr/lib and /lib)
  - -O: invoke compiler optimizer
  - -o: specify executable name (default a.out)
  - -W: specify warning level (implicit, return-type, unused, comment, format, all, etc)
    - ex: gcc -o lab01 -Wall lab01.c
Programming Tools: gcc

- **-c**: suppress linking phase (create object files (.o) without treating unresolved references as errors)
  
  - ex: gcc -c doomC.c lab01.c (creates doomC.o and lab01.o)
  
  - ex: gcc -o lab01 doomC.o lab01.o (creates executable lab01 by linking object files)

- **-R**: specify location of run-time libraries (default /usr/lib) - Absolute pathnames only!

  - UNIX systems use shared (dynamic) libraries - the library modules are not included in the executable, only the location of the *.so file. Use “ldd” to find out what shared libraries an executable requires.
  
  - Defaults to LD_LIBRARY_PATH and LD_RUN_PATH environment variables.
  
  - The -fPIC flag to gcc can be used to generate position-independent code; combined with the ld -G command, you can create your own shared libraries.
Programming Tools: Make

- Make: Keep a set of programs current
  - C programs depend on a number of files (system header files, user header files, C source files, object files, executable files, etc.)
  - When a changed occurs to a file that others depend on, you MUST recompile all dependent files.
  - The “make” program allows your to specify dependency relationships to automate this process
  - Make looks at at dependency lines in the specified file
    - default: the files Makefile (first priority) or makefile (second priority) in the working directory
    - explicit: specified using the -f flag to make

Rule Syntax:

```
target: prerequisite-list
  TAB  construction-commands
```
Programming Tools: Make

- Each target (often a file) specifies zero or more prerequisite targets (often files). If any of the files in the dependency list have been modified since the target file’s last modification date, then the specified construction commands are invoked.
- Defaults to first rule unless specified explicitly on the command line
- Syntax: target: prerequisite-list
  TAB construction-commands

```
lab01: doomC.o lab01.o
   gcc -o lab01 doomC.o lab01.o
lab01.o: lab01.h lab01.c
   gcc -c lab01.c
doomC.o: doomC.h doomC.c
   gcc -c doomC.c
clean:
   rm -f core *.o
```
Programming Tools: Make

- Comments – Any text from # to end of line
  - # $Id: doomCode, v1.2 2005/09/20

- Macros – simple = pairs. There are lots of defaults, honor conventions!
  - Use make –p to see default macros/variables
  - CFLAGS = -O –systype bsd43
  - make –p will show you the rules/macros that make is using
  - $@ is the name of the file to be made
  - $? is the names of the changed dependents
  - $< is the name of the related file that caused the action
  - $* is the prefix shared by the target and the dependent files
  - SHELL = /bin/sh Vs. SHELL = /bin/tcsh

- Line continuation
  - \ at the end of line of text indicates continuation on the next line.
Programming Tools: Make

- Implied dependencies: If you do not include a dependency line for an object file, make assumes that it depends upon a compiler or assembler source code file with the same name. BEWARE: using implied dependencies requires that you use MACROS to pass necessary flags
  - If no dependency file is specified, only implied dependencies are used
  - If no target is specified, the first dependency in the file is the default

```
CC=gcc
CFLAGS=-Wall
$(SRC)= doomC.c lab01.c
$(OBJ)=($SRC:.c=.o)

lab01: $(SRC) $(OBJ)
    $(CC) -o lab01$CFLAGS $(OBJ)
#(others are implicit)
```
Programming Tools: Make

- Implicit rules
  - How does make default when attempting to build executables from .c files?
  - .c:
    
    $(CC) $(CFLAGS) $@.c $(LDFLAGS) –o $@
  - How about .o files?
  - .o.c:
    
    $(CC) $(CFLAGS) –c $*.c

- People have come to expect certain targets
  - make all: Should compile everything
  - make install: Should install things in the right places (after Macros updated, if necessary)
  - make clean: Should clean things up by getting rid of executable, temporary files, object files, etc.
Programming Tools: Debuggers

- Debuggers
  - `gdb (xgdb, xxgdb - graphical): powerful, freeware, ubiquitous`
    - use help command at gdb prompt for: list, break, run, set args, print, display, up, down, cont
  - `lint`: checks programs for potential bugs and portability problems
  - `truss`: trace system calls and signals
- Other useful commands
  - Standard file commands: `cp, mv, grep, diff, file, ls, mkdir, cd, rm, chmod, ln`
  - Know the uses for: `, `, `|`, `||`, `&`, `&&`, `fg`, `bg`, `jobs`
  - Useful utilities: `script, tar, compress, gzip, which, whereis, apropos, who, w, talk, write, man, man, man`
UNIX system calls

- C programs can easily access the services of the UNIX operating system
- System calls: routines that make operating system services available to programmers
  - creating/deleting files, allocating memory, sending signal to processes
  - e.g. open, read, write, close
- In C, system calls are used in the same way you use ordinary C program modules (functions)
- A variety of libraries have been developed to support programming in C
  - libraries are collections of related functions
  - many libraries functions access basic OS services through system calls
  - default location is generally /usr/lib
- “truss” can provides a system call trace
System calls

- In UNIX/C system calls look like normal library functions
  - `ssize_t write (int fd, const void *buf, size_t count);`
  - routines headers are generally available in `/usr/include` or `/usr/lib/include`
  - Generally return `-1` on error
    - Kernel sets a static global variable (`errno`) to provide more details
- In reality, these functions are _wrappers_ for the system calls that make them easier to use
- System calls invoke a fair bit of overhead as they require a hand-off control to the operating system
  - Be prudent!
What **is** a Process?

- An **instance** of a program in execution.
- Identified by a unique PID (Process-id).
- Created by another process as its child.
- One process can be parent of multiple children.
- Can be killed or stopped by sending it a **signal**.

- Parent **forks** a child by first replicating its own process image.
  - Inherits everything except PID and PPID
  - The role of init
- Child **execs** (overwrites) this image with that of another program.
- While child is running, parent may
  - wait for child to complete execution
  - (foreground execution).
  - continue with its other tasks (background execution).
- Process terminates and parent picks up exit status of child.
- Kernel removes entry for dead child from process table.
Orphan processes

- When a child process dies it:
  - Child leaves behind exit status in process table.
  - Child turns to *zombie* state.

- Parent may
  - pick up exit status; child is now completely dead.
  - may not wait; child continues to remain in zombie state.
  - Zombies can’t be killed; shown as `<defunct>` in *ps* output.

- When parent process dies before child
  - Child adopted by *init*.
  - PPID of child changes to 1.
  - When child dies, *init* picks up the exit status.
Signals

- Notification of occurrence of an event.
- Every signal associated with a default action (disposition).
- Process may
  - perform the default action.
  - ignore the signal.
  - catch the signal and invoke a signal-handling function.

- Two signals (SIGSTOP and SIGKILL) can’t be ignored or caught.

- The keyboard and **kill** command generate signals.
Job control

- You can have many jobs running in the background
  - While running background jobs are disconnected from Keyboard/Display

- To run a job in the background:
  - command &
  - or- ^Z to suspend a job, then bg to resume it in the background

- To view background jobs
  - jobs

- Execute commands on running jobs using ps name or %n (jobs) notation
  - fg %2, kill %3, etc.
Common built-in job control functions

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bg/fg</td>
<td>place a job in the background/foreground</td>
</tr>
<tr>
<td>jobs</td>
<td>list active jobs</td>
</tr>
<tr>
<td>kill</td>
<td>send signals to active jobs</td>
</tr>
<tr>
<td>nice command</td>
<td>lower priority of command</td>
</tr>
<tr>
<td>nohup command</td>
<td>do not terminate command on shell exit</td>
</tr>
<tr>
<td>wait</td>
<td>wait for all background jobs to terminate</td>
</tr>
</tbody>
</table>
Section III: Intermediate UNIX

CEG 333: Introduction to UNIX

Dr. Travis Doom, Associate Professor
Department of Computer Science and Engineering
Wright State University
# Basic Regular Expressions

## Repetition symbol

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>zero or more</td>
<td>bo*b: matches bb, bob, boob, etc.</td>
</tr>
<tr>
<td>+</td>
<td>one or more</td>
<td>bo+b: matches bob, boob, etc.</td>
</tr>
<tr>
<td>?</td>
<td>one or zero</td>
<td>bo?b: matches bb and bob.</td>
</tr>
<tr>
<td>{#}</td>
<td>repeat # times</td>
<td>bo{3}b: matches boob.</td>
</tr>
<tr>
<td>{n,m}</td>
<td>repeat n to m times</td>
<td>bo{1,2}b matches bob and boob.</td>
</tr>
</tbody>
</table>

## Character groups

- **[X]** match any character inside the [], can use “|” for “OR”
  - example: any vowel [aeiouAEIOU]
- **[^X]** match any character NOT inside the [^ ]
  - example: any constanten [^aeiouAEIOU]

## Anchors

- **^** Begining of line
- **$** End of line

Ex: `grep ‘ca[t|m]$’ words`
### More Regular Expressions

<table>
<thead>
<tr>
<th>Special symbols</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>any digit char.</td>
<td>[0-9]</td>
</tr>
<tr>
<td>\w</td>
<td>any word char.</td>
<td>[a-zA-Z0-9_]</td>
</tr>
<tr>
<td>\s</td>
<td>any whitespace character</td>
<td>[ \r\t\n\f]</td>
</tr>
<tr>
<td>.</td>
<td>any character except \n</td>
<td></td>
</tr>
<tr>
<td>\D</td>
<td>any non-digit character</td>
<td></td>
</tr>
<tr>
<td>\W</td>
<td>any non-word character</td>
<td></td>
</tr>
<tr>
<td>\S</td>
<td>an non-space character</td>
<td></td>
</tr>
</tbody>
</table>

#### Parenthesis as memory
- \((\) group and memorize
- \1 the first regular expression "memorized"
- \2 the second re memorized
- \3 ... and so on to \9

Ex: grep `'^\(\.)\.*\(\1\)'` words
PERL

- `#!/bin/perl` (which perl to find specific path)
- `chmod +x script`

- THERE IS MORE THAN ONE WAY TO DO IT!
  - www.learnperl.org
  - www.perldoc.com
  - www.cpan.org
  - www.ebb.org/PickingUpPerl

- Replaces sed, awk, sh scripts and many other “intermediate” UNIX filters/tools
Basic System Admin

- What goes on in /etc?
  - /etc/passwd
  - /etc/init.d
  - /etc/rc*.d and runlevels
  - Etc…
- Installing software (tar, gzip, compress)
- crontab
Other useful tools

- latex
- ppm
- xv
- Gnuplot
- uuencode/uudecode
- ...

Wright State University, College of Engineering
Dr. T. Doom, Computer Science & Engineering