Introduction to Unix

UVic SEng 265

Daniel M. German
Department of Computer Science
University of Victoria

What is Unix

- Unix is:
  - is a computer operating system.
  - is a multi-user, multi-tasking operating system.
  - is a machine independent operating system.
- The “Unix” trademark:
  - Owned by AT&T
  - Passed to Unix Systems Laboratories (USL)
  - Passed to Novell
  - Current owner of exclusive rights: X/Open Co. LTD
- So every manufacturer calls its Unix something else:
  - IBM: AIX, HP: HP-UX, Sun: Solaris, SCO: Xenix, Linux, etc.

Unix basics: a brief history

- Created at Bells Labs (part of AT&T now called Lucent) by Ritchie and Thompson around 1970 for their own use
- Academic & research operating system
- Source code given away for a nominal fee
- Became first “portable” OS
- A community effort

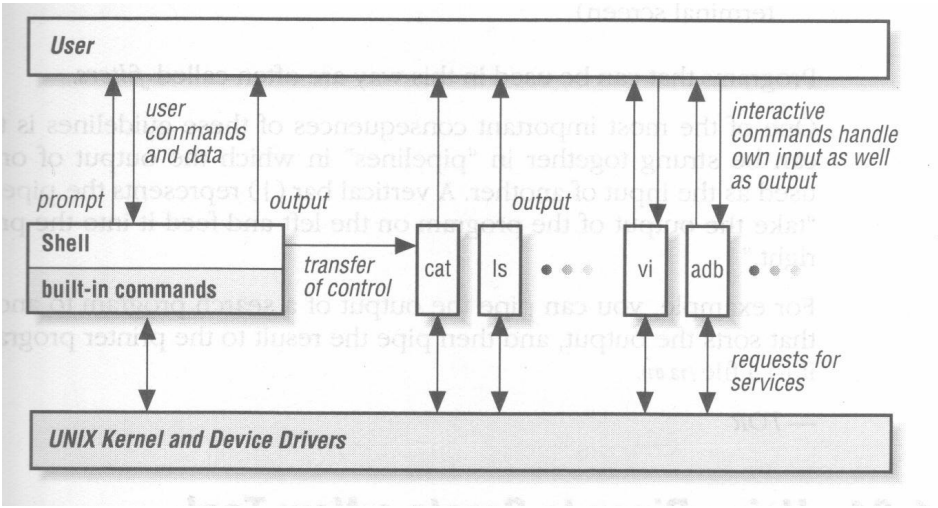
Brief history...

- Berkeley Standard Distribution (BSD)
  - Free!
  - Licence allowed anybody to use it and to modify it
  - First Unix to include standard network support
- Free Software Foundation and GNU Unix
- Linux
- Unix flavours today:
  FreeBSD; NetBSD; XENIX; SunOS; Solaris; SunOS; HP- UX; Linux; A/UX; AIX; Mac OS X; Mach; ...

Why Unix?

- Multiuser
- Multitasking
- Remote processing
- Safe
- Very modular
- Some versions are free (as in freedom, not only as in free beer)

Unix Model



Kernel

- The kernel is the core of the operating system
- Main responsibilities
  - Memory allocation
  - File system
  - Load and executes programs
  - Communication with devices
  - Starts the system

Shell

- It is the responsible for the communication between the user and the kernel
- Reads user commands and acts accordingly:
- Controls program execution
- Multiple varieties
  - `sh, csh, tcsh, zsh, ksh, bash...`
  - In this course we will learn `bash`

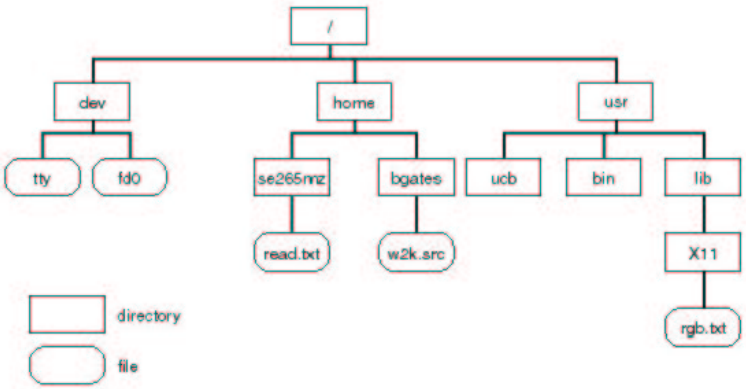
Unix filesystem

- Heart of Unix computing model
- Everything abstracted into a file (devices, programs, data, memory, etc.)
- Responsible for abstracting chucks of raw disk into a logical units
- A file is a sequence of blocks, chained together
- Maps a filename to its chain of blocks
- Provides methods to access the data

Unix filesystem ...

- Arranged in a hierarchy (tree-like structure)
- Folder directory
- Forward slash "/" is used to separate directory and file components (not backslash "\")

Part of a Unix filesystem tree



Some properties of directories

- Directories are ordinary files
- Information contained in a directory file simply has special format
- Each directory contains two specials entries
  - ".." refers to parent directory in hierarchy
  - "." refers to same directory
- System directory is almost a tree
- By using links (built with the command ln), entries in two or more directories can refer to same file

Commands used with directories

- Listing directories

```
% ls -F
se265mz bgates/
% ls bgates
w2k.src
```

- Relative pathnames

```
% lpr bgates/w2k.src
% lpr ./bgates/w2k.src
% lpr ./bgates/../bgates/w2k.src
```


Directory commands...

- Traversing directories

```
% cd /usr
% ls -F
ucb/ bin/ lib/
% cd ..
% ls
dev home usr
```


Some survival tips

- Unix file names are case-sensitive
- e.g., `myFile` and `myfile` are two different names, and the `logout` command cannot be typed as `Logout`
- Unix needs to know your terminal type
- You can display what Unix thinks it is with:

```
% echo $TERM
```

- And reset it to another terminal type with a command like:

```
% export TERM=vt100
```


More survival tips

- Several keyboard characters have special functions
- The command

```
% stty all
```

  displays a list of special control characters.
- You can use `stty` to change these, e.g.,

```
% stty erase ^X
```

  sets the erase (backspace) character on your keyboard to be control-X.
- Well talk a bit more about special character in a few slides

File access

- Access control
- Every file and directory has “attributes”:
  - user (owner of file)
  - group (for sharing)
  - creation time, modification time
  - file type (file, directory, device, link)
  - permissions

```
% ls -l unix_intro.tex
-rw-r--r--  1 dmg  users 20630 Aug 29 14:32 unix.txt
```


File Access ...

- Access to files is controlled by:
  - Ownership
    - association between user and file
    - owner has full control (can set permissions)
- Permissions, three basic types:
  - read (“r”)
  - write (“w”)
  - execute (“x”)
  - (ignore for now: “X”, “s”, “t”)

```
% ls -l unix.tex test
-rwxr-xr-x  1 joe  users    200 Aug 29 14:39 test
-rw-r--r--  1 dmg  users  21009 Aug 29 14:39 unix.tex
```


Permissions

- Permissions can be set for
  - user (“u”): the file owner
  - group (“g”): group for sharing
  - other (“o”): any other
  - all (“a”): user + group + other
- user: the owner of the file or directory; has full control over permissions
- group: entire groups of users can be given access
- other: any user that is not the owner and doesn’t belong to the group having ownership

```
% ls -l unix.tex test
-rwxr-xr-x  1 joe  users    200 Aug 29 14:39 test
```


How permissions affect:

- files:
  - read: allows file to be opened and read
  - write: allows file to be modified
  - executable: tells Unix the file is a program
- directories:
  - read: allows directory contents to be listed
  - write: allows the directory contents to be modified, deleted, or created
  - executable: allows users to access files in that directory

The Shell

- The shell is the intermediary between you and the kernel
- It interprets your typed commands in order to do what you want:
  - The shell reads the next input line
  - It interprets the line (expands arguments, substitutes aliases, etc.)
  - Acts
- There are different shells.
  - 2 main families: `sh` based shells, and `csh` based shells
  - They vary slightly in their syntax and functionality
  - We'll use `bash`, the Bourne Again SHell (derivative of `sh`)
  - Tip. You can find out which shell you are using by typing:
    `echo $SHELL`

Basic command syntax

- Commands will eventually seep into your finger tips
- Lettering will wear off keys from popular command (e.g., `cd`)
- May want to make your own index card of commands, and keep by your terminal as you work
- When you discover a useful incantation, and think you might use it again sometime, write it down

Basic command syntax (contd)

`command [<opt1> <opt2> ...] [arg1 arg2 ...]`

- `<opt#>`
  - `-option`, e.g., `-l` or `-long`
  - `--token`, e.g., `--file`, `--verbose`
- Some options might require an argument: `<opt> arg`
  - `--size 32`, `-name foo -n 5`
- `arg`
  - `intro.txt`, `10`, `'This is the end'`
- For help, try `<command> -h` or `<command> --help`
- For many commands, you can use the *man pages*
  `man <command>`

The `man` pages

- You can ask for help in several ways:
- `man <command>` displays a long description of the command
- `whatis <command>` displays a one liner describing the command (not available for built in commands)

```
gcc       gcc (1)   - GNU project C and C++ Compiler
```

- `apropos <keyword>` displays a list of commands related to the keyword:

```
% apropos cdrom
autorun  (1) - automatically mounts/unmounts CDROMs...
xplaycd  (1) - X based audio cd player for cdrom drives
```

Commands

- are either:
  - built into the shell (very few; examples are `cd`, `pushd`, `popd`, `history`, `fg`, etc.)
  - aliases created by the user or on behalf of the user (on aserver: `h` for `history`; `cp` for `cp -i`; `rm` for `rm -i`)
  - an executable file
    - binary (compiled from source code)
    - script (system-parsed text file)
- to find out the kind of command: `type <command>`

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


Input & output streams

- Each Unix program has access to three I/O streams when it runs:
  - input ("standard input", aka stdin)
  - normal output ("standard output", aka stdout)
  - error output ("standard error", aka stderr)
- by default streams are connected to terminal (your keystrokes for stdin, text on screen for stdout and stderr)
- the shell has mechanisms for overriding this default, allowing stream "redirection"
- very powerful feature

Stream redirection

- Redirection allows you to:
  - create more complex command built up from simpler ones
  - capture output of commands for later review
- Redirecting from files and to files done by the shell
  - stdin:
    `command < file`
  - stdout:
    `command > file`
    `ls -la > dir.listing`

Redirection...

- stdout and stderr together
  `% command >& file`
  `% grep 'hello' program.c >& hello.txt`
- redirection of both stdin and stdout
  `% command < infile > outfile`
  `% sort <unsorted.data > sorted.data`
- Be aware:
  - symbols used for redirection depend on shell you are using
  - our work will be with the Bash shell (bash, sh)
  - slight differences with C shell (csh, tcsh)

A few basic commands

- `% cat [file1 file2 ...]`
  (concatenate) copy the files to stdout, in order listed
- `% more [filename]`
  browse through a file, one screenful at a time
- `% date`
  displays current date and time
- `% wc [filename]`
  (word count) counts the number of lines, words and characters in the input
- `% clear`
  clear the screen

Redirecting amongst programs

- For this we use “pipes”
- The output of one prgram can become the input of another one
- E.g., given a log (“calls.log”) of incoming telephone calls, determine from the last 100, how many different callers have a ’721’ prefix
- the long way:

```
% tail -100 calls.log > temp0.txt
% sort -u temp0.txt > temp1.txt
% grep ’721-’ temp1.txt > temp2.txt
% wc temp2.txt
```


Redirecting...

- The short way:

```
% tail -100 calls.log | sort -u | grep ’721-’ | wc
```

- Pipes help save time by eliminating the need for immediate files
- Pipes can be extremely long and complex
- All commands are executed at the same time!
- If any processing errors occurs in a complex pipe, the entire pipe will need to be re-executed

Command sequencing

- Multiple commands can be executed one after the other on the same line
- E.g. command1; command2; command3

```
% date; who; pwd
```

- May group sequenced commands together and redirect output

```
% (date; who; pwd) > logfile
```

More about stdin

- If a command expects input from stdin and no redirection is done, the program will take input from the keyboard
- A command that expects input on stdin will appear to "hang" –use Control-D (EOF) at the beginning of a line to end input
- E.g.,you type `wc`
- It expects a text file from stdin; typing a string of characters followed by a new line and ^D will give output, or ^C to abort.
- Be aware: this is different from MS-DOS systems where Control-Z has the same meaning; ^Z on a Unix will suspend the current job (more about that in a few lectures)

Software Flow Control

- A method of controlling the rate at which one device sends data to another
- XON/XOFF
- XON: transmission on Ctrl-Q
- XOFF: transmission off Ctrl-S
- If you accidentally hit Ctrl-S and your terminal mysteriously freezes, try Ctrl-Q

Remaining topics

- Filename expansion
- Command aliases
- Quoting & backslash protection
- bash command history
- bash variables
- Process control (signals, concurrent processes)

Filename expansion

- Shorthand (read: finger saver) for referencing multiple files on a command line
- * any number of characters
- ? exactly one of any character
- $[jp3]$ any one of "j", "p" and "3"
- Can be combined together

Filename expansion contd.

- Examples:
- Count lines in all .c files

```
% wc -l *.c
```

- List detailed information about all files with a single character suffix

```
% ls -l *.?
```

- Send all Chap* and chap* files to the printer

```
% lpr [Cc]hap*
```


Filename expansion...

- * matches any string of characters (except an initial period)

```
% rm *.o         # remove all files ending in '.o'
% rm *           # remove all files in directory
% rm ../*-old*.c
```

- ? matches any single character (except an initial period)

```
% rm test.?     # remove test.c and test.o (etc.)
```


Command aliases

- A shorthand mechanism to save even more typing (Unix users are amazingly lazy by nature)
- They allow a string to be substituted for a word when it is used as the first word of command.
- syntax

```
% alias mycommand 'command [opt] [arg]'
```

- examples:

```
% alias more=less
% alias ls='ls -F'
% alias cd='cd $*; ls' # $* == all parameters to shell
```

- to see a list of existing aliases:

```
% alias
```


Quoting

- Used to control bash's interpretation of certain characters
- What if you wanted to pass $ as part of an argument?
- Strong quotes ': All characters inside a pair of ' ' are untouched
- Weak quotes ": Some characters are interpreted inside " " ($, ‘)

```
% echo $SHELL *
/bin/bash file1 file2 file3
% echo '$SHELL' '*'
$SHELL *
% echo "$SHELL" "*"
/bin/bash *
```

- Until you understand the exceptions for weak quotes, use only strong ones

Backslashes and backticks

- Single characters can be “protected” from expansion by prefixing with a backslash (\)
- Example:

```
command \* == command '*'
```

- Backticks (`) used to substitute a command’s output ”in-place” (command substitution)

```
% echo "Current date is" `date`
Current date is Thu Aug 30 16:24:03 PDT 2001
```


More Unix shell substitution examples

-
```
% cp ~sue/junk \*  # make copy of junk
                   # named '*'
% rm '*'           # remove file named '*'
                   # not "delete all files"
```

- Single quotes around a string turn off the special meanings of all characters

```
% rm 'dead letter'   #
% cp ~sue/junk '*'   # equivalent to earlier item
```


bash Command History

- bash (and other shells like sh, tcsh, ksh, csh) maintain a history of executed commands
- History will show list of recent commands
- The default size of the history is 500 commands
- Using the history
  - Simple way: use up and down arrows
  - Using readline
  - Using the “!” csh history expansion:

```
% !!        #  repeat last command
% !foo      #  last command that started with foo
```


csh Style History Expansion...

- Only way to repeat commands in `csh`
- People prefer to use tcsh or bash for this only reason
- But it still very useful:

`!!`	repeat last command line
`!string`	repeat last command line that starts with string
`!-N`	repeat N-th previous command line
`!N`	repeat N-th command line
`!*`	repeat all arguments of last command
`!$`	repeat last argument of last command
`!^`	repeat first argument of last command

csh Style History Expansion...

```
% cc -c buggyProg.c
... (many syntax errors reported)
% emacs !$           # !$ gets last argument of previous
                     # equivalent to emacs buggyProg.c
... (fix these errors)
% !cc                # repeat the last cc command
% history            # see history of last commands
...
986  pwd
987  cd Art_Of_Noise/
988  ls
989  mpg123 *.mp3
% !-2                # executes 2 commands before
ls
% !986               # executes command 986 in history
pwd
```


Readline

- Command-line editing interface
- Provides editing and text manipulation
- Part of many GNU applications
- Includes two default modes `emacs` or `vi`
- Emacs mode:
  - `^A` Go go beginning of line
  - `^K` Erase from cursor to end of line
  - `Alt f` Advance one word
  - `Alt b` Go back one word
  - etc.
- You can customize keystrokes

Job Control

- Graphical environments (like X Window) allow users to create many windows, each with its own shell
- Each windows can execute one command at a time
- Alternatively, the shell allows you to execute multiple programs in parallel from one window
- Running a program in the background...

```
% netscape &
[1] 3141
```

- ... and bringing it back to the foreground

```
% fg %1
```

- Or keeping it in the background

```
% bg %1
```


Job Control ...

- Stopping and restarting a program:

```
% emacs hugeprog.c
^Z
Stopped
% jobs
[1] + Stopped  emacs hugeprog.c
% gcc hugeprog.c -o hugeprog
(...compiler errors..)
% fg %1
[1]  emacs hugeprog.c
```

- Terminating (or ”killing”) a job:

```
% kill %1             # use kill -9 %1 to kill
                      # those that _refuse_ to die
```

or

```
% kill %cc            # job that starts with cc
```

Shell Variables

- The shell carries with it a “dictionary” of variables with values
- Some are internal and some are user defined
- Used to customize the shell
- Use `set` to display them:

```
% set
PWD=/home/dmg
GS_FONTPATH=/usr/local/fonts/type1
XAUTHORITY=/home/dmg/.Xauthority
TERM=xterm
HOSTNAME=ag
...
```


Shell Variables...

- Many variables are automatically assigned values at login time
- Variables may be re-assigned values at the shell prompt; new variables may be added; and variables can be discarded
- Assigning or creating a variable:

```
% <var>=<value>
```

- To delete a variable:

```
% unset <var>
```


Shell Variables... PS1

- PS1 is a good example of how a shell var. customizes the shell
- PS1 sets the prompt

```
% PS1='[\u@\h \W]\$ '
[dmg@aluminium dmg]$ PS1='[\t \d \u@\h \w]\$ '
[22:00:50 Tue Sep  4 dmg@aluminium ~/work/photos]$
```


Shell Variables... PATH

- Helps the shell find the commands you want to execute
- Its value is a list of directories separated by :
- When we run a program, it should be in the PATH in order to be found

```
% echo $PATH
PATH=/usr/bin:/usr/sbin:/etc:/usr/ccs/bin
% my_program   # located in /home/dmgerman/bin
bash: my_program: command not found
% PATH=$PATH":/home/dmgerman/bin"
% type my_program
my_program is /home/dmgerman/bin/my_program
```

- The shell searchs sequentially in the order the directories are listed

Environment Variables...

- Some shell variables are exported to every child process
- These are environment variables
- Environment variables are exported to every child process
- “Exporting” shell variables to the environment:

```
% export <var>
% export <var>=<value>
```

- Example:

```
% export EDITOR=emacs
```


Customizing your Shell

- bash reads ~/`bash_profile` every time you login (~ corresponds to your home directory)
- You can put aliases and set variables in that file

```
PS1='[\u@\h \W]\$ '
export EDITOR=emacs
alias cp='cp -i'
alias rm='rm -i'
alias mv='mv -i'
alias ls='ls -F'
```


Epilogue

- This is a quick introduction
- You have to try things
- You have to read `man` pages and other sources of info
- You have to learn from others
- Try linux at home
- With Unix, you learn something new every day