Unix System Programming

Introduction

Outline

- UNIX History
- UNIX Today?
- UNIX Processes and the Login Process
- Shells: Command Processing, Running Programs
- The File
- The Process
- System Calls and Library Routines

UNIX History

- Developed in the late 1960s and 1970s at Bell Labs (the most versatile, powerful an flexible OS in the word). K. Thomson, D. Ritchie, McIlroy, Ossanna (nroff) and later Canaday
- UNICS - a pun on MULTICSn time share system (Multiplexed Information and Computer Service) which was supposed to support 1000 on line users but only handled a few (barely 3). (MULTI-UNiplexed)
- Thomson writes first version of UNICS in assembler for a PDP-7 in one MONTH which contains a new type of file system (initial motivation was the game space travel)
  - » Kernel (notion of processes)
  - » shell
  - » editor and the
  - » assembler
- 1969 Thomson writes interpreter B based on BCPL -- Ritchie improves on B and called it "C" (but first NB).
- 1972 UNIX is rewritten in C to facilitate porting

UNIX History (cont)

- 1973 UNIX philosophy developed:
  - » Write programs that do one thing and do it well
  - » Write programs that work together
  - » Write programs that handle text streams, because that is the universal interface

Dennis Ritchie (standing) and Ken Thomson

Thomson

Ritchie

K.I.S.S.

Keep It Simple, Stupid!

UNIX Today

- Supports many users running many programs at the same time, all sharing the same computer system
- Information Sharing
- Geared towards facilitating the job of creating new programs
- Sun: SunOS, Solaris; GNU: Linux; SGI: IRIX; Free BSD; Hewlett Packard: HP-UX; Apple: OS X (Darwin)

What Unix Gets Wrong (Raymond)

- UNIX files have no structure above byte level
- File deletion is irrevocable
- Unix security model is too primitive
- There are too many different kind of names for things
- Having a file system at all may have been the wrong choice
- Final choices are pushed to the as far toward the user as possible (user know better than OS designers what their own need are)
  - » Loosing non-technical users
  - » But maybe longevity because competitors are more tied to one soet of policy or interface choicdes that fades from view

What Unix Gets Right (Raymond)

- Evidence - the Linux revolution
- Open Source Software (cooperative, re-usable)
  - » Key to UNIX's success
  - » David Eckel' agree, his books are freely available and the most profitable!
- Cross-Platform portability an open standards
  - » Consistent API across heterogeneous mix of computers
  - » Scales
- Internet and the WWW
  - » DoD contract for TCP/IP production went to the UNIX development group because of its open source!
- Flexibility all the way down (glue program together)
- Unix is fun to hack
- The lessons of UNIX can be applied elsewhere

User UNIX Interface: SHELL

- Provides command line as an interface between the user and the system
- Is simply a program that starts automatically when you login
- Uses a command *language*
  - » Allows programming (shell scripting) within the shell environment
  - » Uses variables, loops, conditionals, etc.
  - » Accepts commands and often makes *system calls* to carry them out

Various UNIX shells

- sh (Bourne shell)
- ksh (Korn shell)
- csh (C shell)
- tcsh
- bash
- …
- Differences mostly in scripting details

The Korn Shell (ksh)

- I will frequently be using ksh as the standard shell for examples in this class
- Language is a superset of the Bourne shell (sh)

Changing Shell

- On most UNIX machines:
  - » `which ksh` (note path)
  - » `chsh`
- On the some machines:
  - » `which ksh` (note path /bin/ksh)
  - » `ypchsh`
  - » May need to contact system administrator

Environment variables

- A set of variables the shell uses for certain operations
- Variables have a name and a value
- Current list can be displayed with the `env` command
- A particular variable's value can be displayed with `echo $<var_name>`
- Some interesting variables: `HOME`, `PATH`, `PS1`, `USER`, `HOSTNAME`, `PWD`

Setting environment variables

- Set a variable with
  - » Ksh/bash: <name>=<value>
  - » tcsh: setenv <name> <value>
- Examples:
  - » TERM=vt100
  - » PS1=myprompt>
  - » PS1=$USER@$HOSTNAME:
  - » PS1="multiple word prompt> "
  - » PATH=$PATH:$HOME
  - » DATE=`date`

Aliases

- Aliases are used as shorthand for frequently-used commands
- Syntax:
  - » ksh: alias <shortcut>=<command>
  - » tcsh: alias <shortcut> <command>
- Examples:
  - » alias ll="ls -lF"
  - » alias la="ls -la"
  - » alias m=more
  - » alias up="cd .."
  - » alias prompt="echo $PS1"

Repeating commands

- Use `history` to list the last 16 commands
- tcsh: traverse command history:
  - » <CNTRL>-P previous history
  - » <CNTRL>-N next history
- ksh: ESC, then k (up), j (down) RETURN

Editing on the command line

- Some command lines can be very long and complicated - if you make a mistake you don't want to start all over again
- You can interactively edit the command line in several ways
  - » `set -o vi` allows you to use vi commands to edit the command line (ksh)
  - » `set -o vi-tabcomplete` also lets you complete commands/filenames by entering a TAB

Login scripts

- You don't want to enter aliases, set environment variables, set up command line editing, etc. each time you log in
- All of these things can be done in a script that is run each time the shell is started
- For ksh:
  - » `~/.profile` - is read for a login shell
  - » `~/.kshrc`
- For tcsh
  - » ~/.login
  - » ~/.cshrc

Example .profile (partial)

```
# set ENV to a file invoked each time sh is started for
# interactive use.
ENV=$HOME/.kshrc; export ENV
HOSTNAME=`hostname`; export HOSTNAME
PS1="$USER@$HOSTNAME>"

alias 'll'='ls -l'
alias 'la'='ls -la'
alias 'ls'='ls -F'
alias 'rm'='rm -i'
alias 'm'='more'

set -o vi
echo ".profile was read"
```

stdin, stdout, and stderr

- Each shell (and in fact all programs) automatically open three "files" when they start up
  - » Standard input (stdin): Usually from the keyboard
  - » Standard output (stdout): Usually to the terminal
  - » Standard error (stderr): Usually to the terminal
- Programs use these three files when reading (e.g. `scanf()`), writing (e.g. `printf()`), or reporting errors/diagnostics

Redirecting stdout

- Instead of writing to the terminal, you can tell a program to print its output to another file using the > operator
- >> operator is used to append to a file
- Examples:
  - » `man ls > ls_help.txt`
  - » `echo $PWD > current_directory`
  - » `cat file1 >> file2`

Redirecting stderr

- Instead of reading from the terminal, you can tell a program to read from another file using the:
  - » `ksh:     2> operator`
  - » `tcsh:    &> operator`
- Example: suppose j is a file that does not exist

```
{atlas} ls j
ls: j: No such file or directory
{atlas} ls j &> hello.txt
{atlas} cat hello.txt
ls: j: No such file or directory
```


Redirecting stdin

- Instead of reading from the terminal, you can tell a program to read from another file using the < operator
- Examples:
  - » `mail user@domain.com < message`
  - » `interactive_program < command_list`

Pipes and filters

- Pipe: a way to send the output of one command to the input of another
- Filter: a program that takes input and transforms it in some way
  - » `wc` - gives a count of words/lines/chars
  - » `grep` - searches for lines with a given string
  - » `more`
  - » `sort` - sorts lines alphabetically or numerically

Examples of piping and filtering

- `ls -la | more`
- `cat file | wc`
- `man ksh | grep "history"`
- `ls -l | grep "maria" | wc`
- `who | sort > current_users`

UNIX Tutorial

- http://www.ee.surrey.ac.uk/Teaching/Unix/

UNIX File system

- The file system is your interface to
  - physical storage (disks) on your machine
  - storage on other machines
  - output devices
  - etc.
- *Everything* in UNIX is a file (programs, text, peripheral devices, terminals, …)
- There are no drive letters in UNIX! The file system provides a *logical* view of the storage devices

Working directory

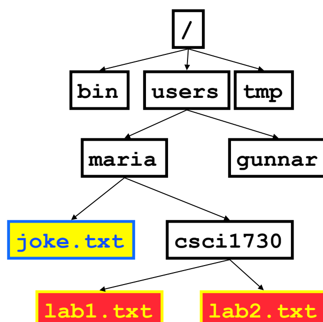
- The current directory in which you are working
- `pwd` command: outputs the absolute path (more on this later) of your working directory
- Unless you specify another directory, commands will assume you want to operate on the working directory

Home directory

- A special place for each user to store personal files
- When you log in, your working directory will be set to your home directory
- Your home directory is represented by the symbol ~ (tilde)
- The home directory of “user1” is represented by `~user1`

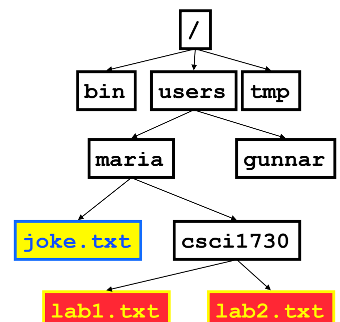
UNIX file hierarchy

- Directories may contain plain files or other directories
- Leads to a tree structure for the file system
- Root directory: /



Path names

- Separate directories by /
- Absolute path
  - start at root and follow the tree
  - e.g. `/users/maria/joke.txt`
- Relative path
  - start at working directory
  - “..” refers to level above
  - “.” refers to working directory
  - If `/users/maria/csci1730` is working dir, all these refer to the same file
    - `../joke.txt`
    - `~/joke.txt`
    - `~maria/joke.txt`



Changing directories

- Change the working directory with the `cd` command
  - » `cd <dir_name>`
  - » Use absolute or relative path names
  - » `cd` by itself equivalent to `cd ~`

Output of `ls -lF`

```
total 4
lrwxr-xr-x 1 maria user 18 Aug 28 13:41 home -> /usr/people/maria/
-rw-r--r-- 1 maria user 94 Aug 28 13:42 nothing.txt
drwxr-xr-x 2 maria user  9 Aug 28 13:40 test_dir/
```

File type, Permissions, Number of links, Owner, Group, Modify date, File name

Types of files

- Plain (-)
  - » Most files
  - » Includes binary and text files
- Directory (d)
  - » A directory is actually a file
  - » Points to another set of files
- Link (l): A pointer to another file or directory
- Special: e.g. peripheral devices

Creating links

- `ln -s <existing_file> <link_name>`
- This command creates a symbolic link
- The file “link_name” will be a pointer to the “existing_file” which may be in another directory or even on another physical machine

File permissions

- Permissions used to allow/disallow access to file/directory contents
- Read (r) 4, write (w) 2, and execute (x) 1
- For owner, group, and world (everyone)
- `chmod <mode> <file(s)>`
  - » `chmod 700 file.txt`
  - » `chmod g+rw file.txt`

File ownership

- Each file has a single owner
- `chown` command can be used to change the owner (usually only root user can use this command)
- There are also various *groups* to which users can belong
- Groups may have different permissions than everyone else

File modification date

- Last time the file was changed
- Useful information when
  - There are many copies of a file
  - Many users are working on a file
- `touch` command can be used to update the modification date to the current date, or to create a file if it doesn't exist

Looking at file contents

- `cat <filename(s)>`
  - "concatenate"
  - output the contents of the file all at once
- `more <filename(s)>`
  - Output the contents of a file one screen at a time
  - Allows forward and backward scroll and search

Moving, renaming, copying, and removing files

- `mv <file1> <file2>` (rename)
- `mv <file1> <dir>` (move)
- `mv <file1> <dir/file2>` (move & rename)
- `cp <file1>`
  `[<file2>|<dir>|<dir/file2>]` (copy)
- `rm [-i] <file(s)>` (remove)

Creating and removing directories

- `mkdir <dir_name>`
  - Create a subdirectory of the current directory
- `rmdir <dir_name>`
  - Remove a directory (only works for empty directories)
- `rm -r <dir_name>`
  - Remove a directory and all of its contents, including subdirectories

Wildcards in file names

- All of the commands covered here that take file names as arguments can also use wildcards
  - `*` for any string, e.g. `*.txt`, `obj*`, `a*.*`
  - `?` for any character, e.g. `doc?`
  - `[]` around a range of characters, e.g. `[a-c]*`

Getting help on UNIX commands

- These notes only give you the tip of the iceberg for these basic commands
- `man <command_name>` shows you all the documentation for a command
- `apropos <keyword>` shows you all the commands with the keyword in their description

The UNIX System - Overview

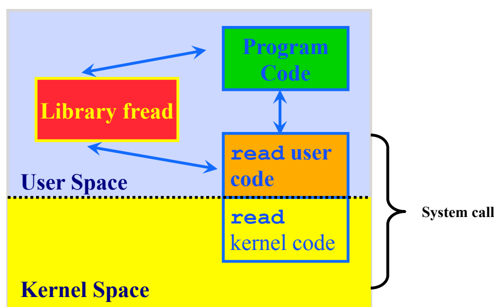
- Kernel – Heart of the OS
  - » Process scheduling
  - » I/O control (accesses)
- Shell – Interpreter between the user and the computer
- Tools and applications
  - » Accessible from shell
  - » Can be run independently of shell

UNIX System Programming

- Programs make *system (primitive), or library subroutine (efficient, special purpose) calls to invoke kernel.*
- Types of system calls
  - » File I/O
  - » Process management
  - » Inter-process communication (IPC) - pipe, signals, shm, sockets, …
  - » Signal handling
- File concept extends to peripheral & IPC
  - » `cat file > /dev/rmt0`
- A process an instance of an executing program

System Calls (Library subroutines)

- System calls: Interface to the kernel



Basic file I/O

- Processes keep a list of open files
- Files can be opened for reading, writing
- Each file is referenced by a *file descriptor* (integer)
- Three files are opened automatically
  - » FD 0: standard input
  - » FD 1: standard output
  - » FD 2: standard error

File I/O system call: `open()`

- `fd = open(path, flags, mode)`
  - » `int open( const char *pathname, int flags, [mode_t mode] )`
  - » `#include <stdlib>`
  - » `#include <fcntl.h>`
- path: string, absolute or relative path
- flags:
  - » O_RDONLY - open for reading
  - » O_WRONLY - open for writing
  - » O_RDWR - open for reading and writing
  - » O_CREAT - create the file if it doesn't exist
  - » O_TRUNC - truncate the file if it exists
  - » O_APPEND - only write at the end of the file
- mode: specify permissions if using O_CREAT

File I/O system call: `close()`

- `retval = close(fd)`
  - » `int close( int filedes );`
- Close an open file descriptor
- Returns 0 on success, -1 on error

File I/O system call: `read()`

- `bytes_read = read(int fd, void *buffer, size_t count )`
- Read up to count bytes from file and place into buffer
- fd: file descriptor
- buffer: pointer to array
- count: number of bytes to read
- Returns number of bytes read or -1 if error

File I/O system call: `write()`

- `bytes_written = write(fd, buffer, count)`
- Write count bytes from buffer to a file
- fd: file descriptor
- buffer: pointer to array
- count: number of bytes to write
- Returns number of bytes written or -1 if error

System call: `lseek()`

- `retval = lseek(fd, off_t offset, whence )`
- Move file pointer to new location
- fd: file descriptor
- offset: number of bytes
- whence:
  - » SEEK_SET - offset from beginning of file
  - » SEEK_CUR - offset from current offset location
  - » SEEK_END - offset from end of file
- Returns offset from beginning of file or -1

UNIX File access primitives

- `open` – open for reading, or writing or create an empty file
- `creat` - create an empty file
- `close`
- `read` - get info from file
- `write` - put info in file
- `lseek` - move to specific byte in file
- `unlink` - remove a file
- `remove` - remove a file
- `fcntl` - control attributes assoc. w/ file

Simple file I/O examples

File I/O using FILEs (C Standard I/O)

- Most UNIX programs use higher-level I/O functions
  - » `fopen()`
  - » `fclose()`
  - » `fread()`
  - » `fwrite()`
  - » `fseek()`
- These use the FILE data type instead of file descriptors
- Need to include `stdio.h`

Using data types with file I/O

- All the functions we've seen so far use raw bytes for file I/O, but program data is usually stored in meaningful data types (int, char, float, etc.)
- `fprintf(), fputs(), fputc()` - used to write data to a file
- `fscanf(), fgets(), fgetc()` - used to read data from a file