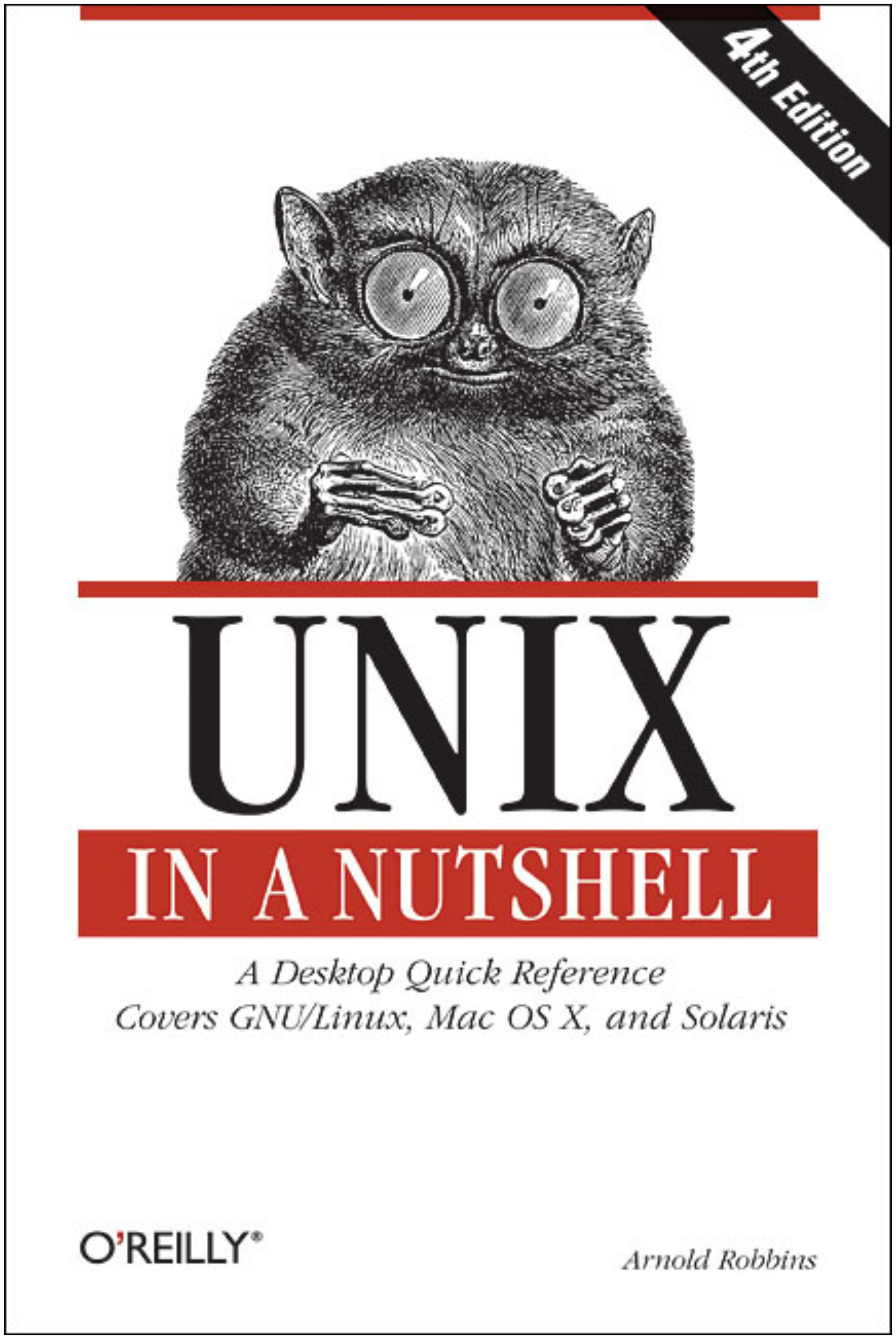
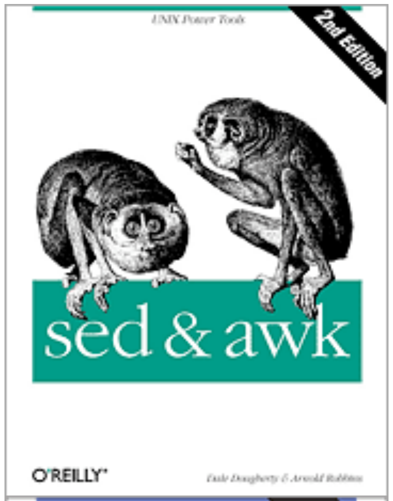
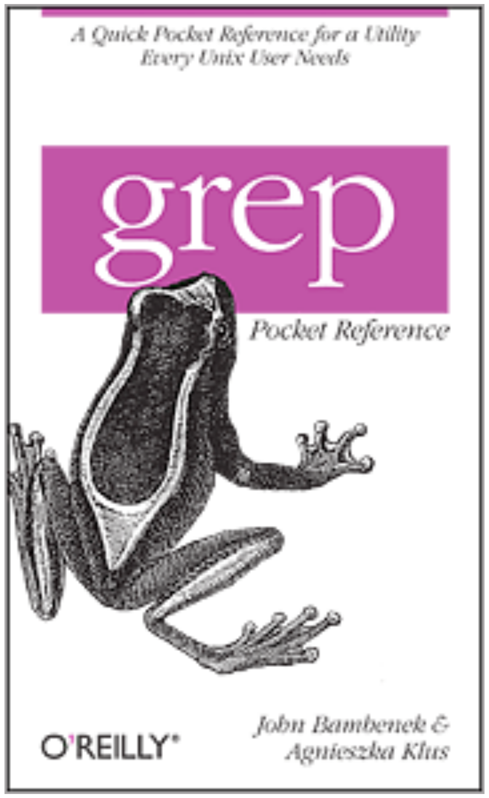
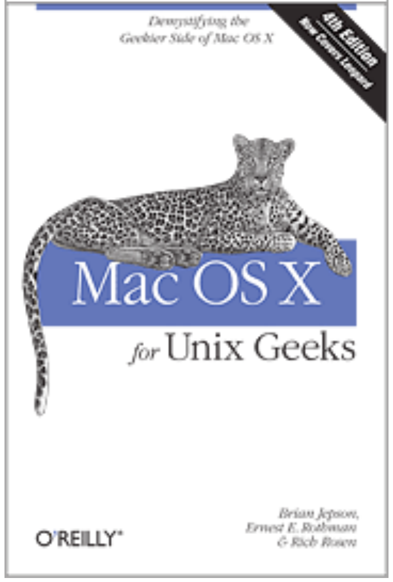

4th Edition
UNIX
IN A NUTSHELL
A Desktop Quick Reference
Covers GNU/Linux, Mac OS X, and Solaris
O'REILLY
Arnold Robbins

Unix Tools

Jeff Freymueller




A Quick Pocket Reference for a Utility Every Unix User Needs
grep
Pocket Reference
O'REILLY
John Bambenek & Agnieszka Klus


Demystifying the Geekier Side of Mac OS X
4th Edition
Now Covers Leopard
Mac OS X
for Unix Geeks
O'REILLY
Brian Jepson, Ernest E. Rothman & Rich Rosen

Resources

- Unix and Linux
  - It comes with the package
- Macintosh
  - Really a Unix machine with fancy wrapping
  - Terminal.app
- Windows
  - “The requirements said: Windows 2000 or better. So I got a Macintosh.”
  - Or you could try Cygwin: http://www.cygwin.com/

Outline

- What is Unix, what is the shell?
- Everything is scriptable
- Files, redirecting input/output, pipes
- `awk` and `grep`: your socially-awkward best friends
- Variables and control
- Power tools: GMT, etc. in future lectures
- Need more power? Upgrade to `perl`
    - `perl` = Practical Extraction and Reporting Language
    - `perl` = Pathologically Extensible Rubbish Lister

Unix

- Unix is the most common operating system for “serious computers”
  - Developed in 1969 at Bell Labs (of the old AT&T)
  - At first, could support two simultaneous users!
  - Rewritten in C in 1973 (before that, assembly language)
- From Wikipedia:
  - Unix was designed to be portable, multi-tasking and multi-user in a time-sharing configuration. Unix systems are characterized by various concepts: the **use of plain text for storing data**; a **hierarchical file system**; **treating devices** and certain types of inter-process communication (IPC) as **files**; and the use of a large number of software tools, **small programs that can be strung together through a command line interpreter using pipes, as opposed to using a single monolithic program that includes all of the same functionality**. These concepts are known as the Unix philosophy.

Unix, Unices, Linux

- Numerous Unix variants have sprung up over the years, some academic and some commercial.
  - BSD, Solaris, HP-UX, …
  - Linux is unix-like, not Unix
    - Started as a hobby project by Linus Torvalds
    - Made useful by existence of free, open source software tools (like the Gnu project)
- OS consists of two major parts: kernel and everything else
  - Kernel: master control program, starts and stops processes, handles low-level file/disk access, etc.
  - Many modular tools and programs
  - Every program runs within its own ***process***
  - User interacts through a command shell

Programs = Little Black Boxes

- Just about every thing you use in Unix/Linux is really an external program (not a shell command or part of the kernel).
- Most of these communicate with the outside world in just 4 ways
  - They get arguments on the command line
  - They receive input from standard input
  - They send output to standard output
  - (They also send error messages to standard error).
- Small, reusable pieces that you can assemble in any way you like to do complex tasks.

Examples of Tools

- `ls` print a listing of files in a directory
- `mv` move or rename files
  - Example:

```
jeff% ls SRTM
SRTM1/ SRTM3/
```

    - "jeff%" is a prompt from the shell, telling me it is ready for input
    - I typed "`ls SRTM`"
    - The ls program produced some output: "SRTM1/ SRTM3/"
- Note that Unix treats even directories, inputs and outputs as files.
- You might think these are "low level" functions of the operating system, but each exists as an independent program. They take parameters (given as command line arguments), and send output to a "standard output" file

Strengths and Weaknesses

- Strengths
  - Underlying philosophy has proven wildly successful
  - Unix is a ***very*** robust OS (Linux approximately so)
  - Simple tools can be linked together to do complex things; Unix makes this easy
- Weaknesses
  - Names of many commands/programs are famously cryptic
  - Online help in the form of ***man pages***, which are really designed to remind experts of details they have forgotten, not teach novices how things work.

What is the Shell?

- The shell is a user interface. It is a program that interprets the commands you type and executes them. It also provides output in some useful form (to a window on your screen)
  - It send output to “**standard output**”
- The shell doesn’t care whether its input comes from the keyboard or from a file
  - It takes input from “**standard input**”
  - As far as Unix is concerned, the keyboard is just another file. You the user are a program and your standard output is the shell’s standard input.
- Many shells can be running at once, each with its own little world inside it.
  - You can start up one or more *sub-shells*, which do something and report output back to your shell.

Which Shell?

- There are many different shells
  - Bourne shell (sh)
  - C shell (csh) syntax is more like the C language
  - tcsh (tcsh) is really like the C shell, except it is free
  - Bash (the Bourne Again SHell) popular with Linux
  - More shells: ksh, zsh, …
- Which shell is the best?
  - *Which is better, rock or jazz?*
  - I will use tcsh in my examples
- Which is your default shell?
  - Seislab Suns: tcsh
  - Geodesy Lab Linux: tcsh
  - Mike West's Linux: bash
  - You can change your default shell

Some basics: directories

- Files are organized into directories, like in every other computer system. You might refer to a file like this:

  /srtm/version2/SRTM3/Africa/S35E025.hgt.zip

- ***Names are case-sensitive!*** Jeff.dat ≠ jeff.dat
- Unix has two particular ways of specifying files or directories:
  – Full pathnames

    `/home/jeff/junk_files/bork.dat`

  – Relative pathnames

    `bork.dat`

    `junk_files/bork.dat`

    - Partial pathnames are relative to the current directory

Some basics: current directory

- The ***current directory*** is the directory you are sitting in right now
  - In a graphical system, this is the top window in your "Windows Explorer" or "Finder".
  - If you create a new file, it will be in this directory unless you tell the shell otherwise.
- Some directory commands:

`cd junk_files`	(change current directory)
`pwd`	(print working directory)

- Special directory symbols:

`.`	(the current directory)
`..`	(one level up)
`~`	(your home directory)

Some basics: wildcards

- It is really useful to be able to match several filenames at once. For example

  ```
  mv bork.dat fubar.* junk_files
  ```

  - The wildcard * matches any number of characters, so the line above would match these files:
    - fubar.txt , fubar.job , fubar.1
  - But ***not*** the file fubar1.txt
- Wildcards:

`*`	(match 0 or more characters)
`?`	(match 1 character)

- What does this match?

  ```
  09*alaska*.??d
  ```

- Here is a fancier wildcard, which matches a range of characters:

  ```
  clgo20090[1-6]??.dat
  .[a-z]*
  ```

Wildcard Quiz

	foo1.dat	bork.dat	foobar.txt	My_files.txt
foo*				
*.txt				
_				
?o??.*				

Did you ace the quiz?

Extra Credit:

- foo?.*
- [a-z]*
- *t

Some basics: the path

- When you type something, the shell will try to execute it as a program. For example:
  - jeff% `rm bork.dat`
  - The shell breaks it down this way:
    - Program to run: `rm`
    - Argument(s) passed to program: `bork.dat`
  - This particular command removes (`rm`) the file named `bork.dat`
  - How does the shell know where to find the program `rm`?

The path: path or PATH

- It uses a special variable called the ***path***.
- Both csh and tcsh have two ways to set the path. The sh and bash shells do the same thing in a different way.

```
setenv PATH /gipsy/bin:${PATH}
set path = (/gipsy/bin $path)
```

- The shell internally maintains a list of all ***executable programs*** in these directories.
- It looks in the first directory in the list first.
  – If you have two programs with the same name, you need to know the path to know which one will be executed!

Everything is Scriptable

- Why do repetitive tasks yourself? That’s what you have a computer for.
- This is equally true for shell scripts and MATLAB programs (yes, the MATLAB command window is a kind of shell)
- A script doesn’t have to be complicated to be useful, it just has to do something reliably and more easily than typing.
- Here’s a script I use to update the online copy of my website from the master code on my own computer.

```
cd ~/Sites/jeff
rsync --rsh=ssh -av * denali.gps.alaska.edu:/home/jeff/Web
```

The tilde (~) is a special character that means your home directory

Everything is Scriptable

- Don't
  - Type a long series of commands over and over again
  - Copy and paste a long series of commands
- Do!
  - Any set of commands you type at the prompt can be saved and made into a script for repeating later
  - Recording the commands you type can be a good way to get started at making a simple program.
  - Learn to use variables to make your series of commands more general and improve automation.

Parts of a simple script

```
#!/bin/csh
#
##BRIEF
# create directory ./logfiles and  move autoclean
# logfiles there.
#
##AUTHOR
# Ronni Grapenthin
#
##DATE
# 2008-09-10
#
##DETAILS
# Creates directory ./logfiles if it does not
# exist and moves autoclean's
# logfiles there. Logfiles are of the format
# *____*.i* but have to be moved
# separately due to the limitations of 'mv'
#
##CHANGELOG
#
# First version.
#
# 2009-02-02 (Jeff Freymueller): Change "mv"
# command to "/bin/mv -f" to force overwrite of
# duplicate files
```

What program to execute this with

A LOT of comments!

(continued)

```
# move logfiles into directory ./logfiles


if !(-e ./logfiles ) then
   echo "creating './logfiles
   /bin/mkdir ./logfiles
endif

echo "mv *____*.i0.cleanlog ./logfiles"
/bin/mv -f *____*.i0.cleanlog ./logfiles
echo "mv *____*.i0.editing ./logfiles"
/bin/mv -f *____*.i0.editing ./logfiles
echo "mv *____*.i0.postbreak ./logfiles"
/bin/mv -f *____*.i0.postbreak ./logfiles

echo "mv *____*.i2.cleanlog ./logfiles"
/bin/mv -f *____*.i2.cleanlog ./logfiles
echo "mv *____*.i0.editing ./logfiles"
/bin/mv -f *____*.i2.editing ./logfiles
echo "mv *____*.i0.postbreak ./logfiles"
/bin/mv -f *____*.i2.postbreak ./logfiles

echo "mv *____*.i* ./logfiles"
/bin/mv -f *____*.i*  ./logfiles
```

Create a directory if it is not already there

Run some commands

Redirecting Input and Output

- Many programs are designed to take input from **standard input**, and send output to **standard output**.
  - By default, these are the keyboard and screen
  - You can change that!
- This is called I/O redirection
  - > means redirect output
  - < means redirect input
  - You can use both at the same time

Examples:

- Send the output of something to a file:
  - ls *.dat > myfiles
  - psxy infile.dat > map.ps
- Take input from a file
  - sort < myfiles
- Both at the same time:
  - myprogram < commands > output

Unleashing the Shell: the Pipe

- A “Pipe” is a way for the output of one program to be sent as the input to another program.
- A vertical bar (|) indicates a pipe
- You can pipe together as many programs as you like, as long as each one reads from standard input and writes to standard output.

Examples:

- Use a pager:
  - ls *.dat | more
- Count files of a given type:
  - ls *.dat | wc -l
- Sort listings:
  - ls –l | sort –nr –k 5
- More
  - ls –l | sort –nr –k 5 | more

grep

- In its simplest form, `grep` finds every line in a plain text file that contains a certain string.
- Its “search string” can be more than a simple string: regular expressions
- You can include special characters and wild cards
  - ^ start of line
  - $ end of line
  - . match exactly one character

Examples:

- Find every line with string “AK”
  - grep AK city+state_list
- A fancier example:
  - wget -O - http://www.cygwin.com/ | grep “Windows 98”
- Using some wildcards:
  - grep "^AB.. " ~/sitevecs
- Search for two strings:
  - grep AK city+state.list | grep River

Variants of grep: egrep, etc

- There are a number of variants of grep, which behave a little differently.
- A useful one is egrep, which allows you to use an “OR” construct to find matches to any of two or more strings:
  - egrep “(string1|string2)” file
- For compressed files, use zgrep, zipgrep, bzgrep
- See man fgrep

- Example:
- List all PBO sites in Alaska
  - egrep "(^AB..|^AC..|^AV..)" ~/sitevecs | more
- Output is:

```
AB01    ATKA ISLAND
AB02    NIKOLSKI
AB04    SAVOONGA PBO
```

awk

- awk is an incredibly powerful text processing language (think food processor)
- If you want to get the third word of text from a line, or want to get the difference between the numbers in the 5th and 6th columns, divided by the square root of the sum of squares of the numbers in the first 3 columns, awk is your tool.
- Named for its authors: Aho, Weinberger, Kernighan
- Use it with pipes to get the full effect!

- "AWK is a language for processing files of text. A file is treated as a sequence of records, and by default each line is a record. Each line is broken up into a sequence of fields, so we can think of the first word in a line as the first field, the second word as the second field, and so on. An AWK program is of a sequence of pattern-action statements. AWK reads the input a line at a time. A line is scanned for each pattern in the program, and for each pattern that matches, the associated action is executed." - Alfred V. Aho

awk Principles

- Every line is broken up into fields. By default, whitespace separates fields
- awk reads each line in its input file (or standard input) and does something based on its command program (a string, or a series of commands in a file)

  awk “command string” file(s)
- The command string is of the form “pattern {action}”
- Example:
  - awk ‘NF > 3 {print $4}’ myfile.txt

Some awk examples

- Print the first field of every line
  - awk ‘{print $1}’ myfile.txt
- Print every line with two fields
  - awk ‘NF == 2 {print $0}’ myfile.txt
- Get the day of the week
  - date | awk '{print $1}'
- Use the character “:” to separate fields
  - awk -F: '{print $1 " has the name " $5}' /etc/ passwd | more

Shell Variables

- Regular variables
  - Purely internal to each shell
  - Generally lowercase
- How to set a variable
  - set counter = 3
  - @ counter = $counter + 1
- How to access it
  - echo $counter
- A neat trick:
  - set datfiles = `ls *.dat`

- Environment variables
  - Can be accessed from within programs started by shell
  - Generally uppercase
- How to set:
  - setenv PRINTER blackburn
- How to access
  - echo $PRINTER

Control Structures

Conditionals

- if (test) then … end
  - if (test) statement
  - if – then – else – end
  - if – then – elseif – end
- Test can be a comparison to a numerical or string value, or special stuff:
  - if ($val == 0) echo “Zero!”
  - if ($count > 1) echo positive
  - if (-d $file) echo “$file is a directory”

Loops

- foreach val (*) … end
- While (test) … end