



Curtin College

in association with



Curtin University


# Advanced Bash

Computer Systems (CS2000)

Trimester 2 2020

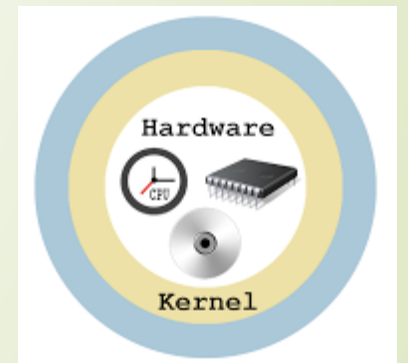


# Three Main OS Components

- **Kernel** – manages the operation of the computer
  - **Shell** – provides for interaction between the user and the computer
  - **Filesystem** – provides a way to organise and manage all information on the computer's disk(s)
- 

# Role of the Kernel

- Main program loaded at startup that manages the operation of the computer
  - Manages devices, memory, and processes
  - Handles switching of applications
- Similar to an air traffic controller at an airport





# The Shell

- The shell is a program that allows the user to type commands, options, and arguments
- Many shell programs exist
- Most popular shell is the “Bash” (Bourne Again Shell)
- A user's shell can be changed by the *usermod* command by root
- User's default shell stored in */etc/passwd*



# What is a Command?

- A program executed on the command line
- Includes:
  - Built-in shell commands
  - Binary commands stored in files
  - Aliases
  - Functions
  - Scripts



# Basic Command Syntax

- Syntax: `command [options...] [arguments...]`
- Commands, options, and arguments are all case sensitive



# Specifying Options

- Short options are specified with a hyphen and a single character such as -a or -x
  - Short options can be combined: -ax
- Long options are specified with a double hyphen and a “word”: --all
  - Long options can not be combined
- Both short and long options can be used together



# Specifying Options

- The option "--", by itself, means "no more options"
- BSD\* style options use no hyphen, just a single character: i.e. "a"
- \*Berkely Standard Distribution (UNIX)





# Specifying Arguments

- Arguments follow options
- Arguments are normally file or directory names
- Some commands require arguments
- Use single quotes around arguments if they contain special (non-alphanumeric) characters



# Command Completion

- Saves time typing large command names
  - Start by typing part of command name and then press the Tab key
  - Will complete the rest of the command name if what has been typed so far is unique
- If not unique, press the tab key a second time.
- Also works to complete file and directory names when used as arguments



# The file command

- Only text files should be displayed, not binary files
- Displaying binary file can cause terminal corruption
- Either logout or run the reset command to fix terminal corruption
- To view file contents type, use the file command:

```
$file filename
```



# The split command

- The split command will break large files into smaller files
  - Useful for file transfer when large files create problems
- Syntax:

```
split [OPTION]...[INPUT [PREFIX]]
```
- INPUT is a file or stdin
- The default size for each split file is 1000 lines



# The split command

- By default the new files will be named with a prefix of x and a suffix of aa, ab, etc.
- For example, the first file would be called xaa, the second file would be called xab, etc
- The -d option splits files to have a numeric suffix instead of a default alphabetic suffix
- The file names will start with the PREFIX, if specified; if not specified, then "x" is used



# The nl command

- The nl command will number the lines of its output
- By default will only number lines that are not blank
- To number every line use:


```
$nl -ba
```



# The paste & join commands

- The paste command merge the lines of one or more files, line by line, using a tab delimiter by default
- Use `-d` to specify a different delimiter
- The *join* command merges files, matching the values of fields to determine which lines to combine
- Use `-t` to specify a different delimiter
- Try *diff* as well.





```
user@ubuntu:~$ cat fname
Ankur
Anubhav
Charul
Himanshu
Ishaan
Piyush
Shubham
Vishal
user@ubuntu:~$ cat lname
Patel
Goyal
Sharma
Grover
Gupta
Tyagi
Gupta
Chaudhary
user@ubuntu:~$
```

```
user@ubuntu:~$ paste fname lname
Ankur   Patel
Anubhav Goyal
Charul  Sharma
Himanshu      Grover
Ishaan  Gupta
Piyush  Tyagi
Shubham Gupta
Vishal  Chaudhary
```

Paste

Join

```
user@ubuntu:~$ cat 10batch
1201   Ashok
1202   Ajay
1203   Arvind
1204   Asha
1205
1206   Arun
user@ubuntu:~$ cat 11batch
1201   Binny
1202   Bunny
1203   Buddy
1204   Bonny
1205   Bishal
1206
user@ubuntu:~$ join 10batch 11batch
1201 Ashok Binny
1202 Ajay Bunny
1203 Arvind Buddy
1204 Asha Bonny
1205 Bishal
1206 Arun
user@ubuntu:~$
```



# The sort command

- The *sort* command displays a file sorted on a specific field of data
- To specify the fields to sort from first to last, you can use one or more *-k* options
- Use the "n" value to perform numeric sorts

**\$sort -t',' -k1n os.csv**

1970,Unix,Richie

1970,Unix,Thompson

1987,Minix,Tanenbaum

1991,Linux,Torvalds

```
root@Ubuntu:~# cat alpha.txt
b
D
C
A
C
B
d
a
root@Ubuntu:~# sort alpha.txt
a
A
b
B
C
C
d
D
```



# The sort command

- Use the "r" value to reverse sort order
- Use the -u option to remove duplicate lines
  - lines starting with a number will appear before lines starting with a letter;
  - lines starting with a letter that appears earlier in the alphabet will appear before lines starting with a letter that appears later in the alphabet;
  - lines starting with a lowercase letter will appear before lines starting with the same letter in uppercase.



# The `uniq` command

- The *uniq* command removes duplicate lines from sorted documents
- Use `-c` to get a count of each line

# Variables Again

- Builtin variables:
  - variables affecting bash script behaviour
- \$BASH
  - The path to the Bash binary itself

```
A0004358:~ 210799e$ bash
bash-3.2$ echo $BASH
/bin/bash
bash-3.2$
```

# Variables Again

## ➤ \$BASH\_VERSION[n]

- A 6-element array containing version information about the installed release of Bash. This is similar to \$BASH\_VERSION

```
# Bash version info:
for n in 0 1 2 3 4 5
do
    echo "BASH_VERSIONINFO[$n] = ${BASH_VERSIONINFO[$n]}"
Done
# BASH_VERSIONINFO[0] = 3 # Major version no.
# BASH_VERSIONINFO[1] = 00 # Minor version no.
# BASH_VERSIONINFO[2] = 14 # Patch level.
# BASH_VERSIONINFO[3] = 1 # Build version.
# BASH_VERSIONINFO[4] = release # Release status.
# BASH_VERSIONINFO[5] = i386-redhat-linux-gnu # Architecture
```

# Variables Again

- ▶ `$BASH_VERSION`
- ▶ The version of Bash installed on the system

```
bash$ echo $BASH_VERSION
3.2.25(1)-release

tcsh% echo $BASH_VERSION
BASH_VERSION: Undefined variable.
```
- ▶ This is a good method of determining which shell is running.
- ▶ `$SHELL` does not necessarily give the correct answer.

```
bash-3.2$ tcsh
[M-A0012092-S:/etc] 210799e% echo $SHELL
/bin/bash
[M-A0012092-S:/etc] 210799e% echo $BASH_VERSION
BASH_VERSION: Undefined variable.
[M-A0012092-S:/etc] 210799e%
```



# Variables Again

- \$UID
  - User ID number
  - Current user's user identification number, as recorded in /etc/passwd
  - This is the current user's real id, even if she has temporarily assumed another identity through su.
- \$UID is a read only variable, not subject to change from the command line or within a script and is the counterpart to the id builtin.



# Variables Again

```
#!/bin/bash
# am-i-root.sh: Am I root or not?
ROOT_UID=0 # Root has $UID 0.
if [ "$UID" -eq "$ROOT_UID" ]
then
    echo "You are root."
else
    echo "You are just an ordinary user"
fi
exit 0
#===== #
```





# To declare or typeset

- The declare or typeset builtins permit modifying the properties of variables.
- This is a very weak form of the typing available in certain programming languages.
- The declare command is specific to version 2 or later of Bash.
  - Not Bourne shell



# Options

- -r readonly
  - declare -r var1
- works the same as
  - readonly var1
- -i integer

```
declare -i number
# The script will treat subsequent occurrences of
"number" as an integer.
number=3
echo "Number = $number" # Number = 3
number=three
echo "Number = $number" # Number = 0
# Tries to evaluate the string "three" as an integer.
```



# Options

- `-a array`
  - declare `-a indices`
  - The variable indices will be treated as an array.
- `-f function(s)`
  - declare `-f`
  - A declare `-f` line with no arguments in a script causes a listing of all the functions previously defined in that script.
- `-x export`
  - declare `-x var3`
  - This declares a variable as available for exporting outside the environment of the script itself.
- `-x var=$value`
  - declare `-x var3=373`
  - The declare command permits assigning a value to a variable in the same statement as setting its properties.



# Using the Web

- The lynx Web and file browser can be used inside a script (with the -dump option) to retrieve a file from a Web or ftp site noninteractively.

```
lynx -dump http://www.xyz23.com/file01.html >$SAVEFILE
```

- With the -traversal option
  - lynx starts at the HTTP URL specified as an argument, then "crawls" through all links located on that particular server.
  - Used together with the -crawl option, outputs page text to a log file.

# Colour and Graphics

- Uses ANSI Escape sequences
- NOTE: \E not portable, may use \033

```
echo -e '\E[37;44m'" \033[1mContact\033[0m"  
# White on blue background
```

```
PS1="\[\033[;37;44m\u@\033[0;35;43m\h:\033[0;33;41m\w$\033[0m  
\]"1;37;44m\u@\033[0;35;43m\h:\033[0;33;41m\w$\033[0m\]"
```

```
[M-A0012092-S:/etc] 210799e% echo $shell  
/bin/tcsh  
[M-A0012092-S:/etc] 210799e% bash  
bash-3.2$ PS1="\[\033[;37;44m\u@  
> \033[0;35;43m\h:\033[0;33;41m\w$\033[0m  
> \]"1;37;44m\u@  
> \033[0;35;43m\h:\033[0;33;41m\w$\033[0m\]"  
210799e@  
M-A0012092-S:/etc$  
1;37;44m210799e@  
M-A0012092-S:/etc$
```

# Colour and Graphics

```
echo -e '\033[37;44m'" \033[1mContact\033[0m"
```

- \033 starts the escape sequence
- [ marks beginning the colour definition
- The following 37;44m specifies the colour code.
- 37 is the foreground colour, which is white
- 44m is the background colour which is light blue
- The string will be enclosed in \[ and \] to prevent the text of the escape sequence from showing up in the display of the shell

# Colours in scripts

## ANSI colour control codes

- 0 = black
- 1 = red
- 2 = green
- 3 = yellow
- 4 = blue
- 5 = magenta
- 6 = cyan
- 7 = white

## ANSI effect control codes

- 0 = reset to normal
- 1 = bold
- 2 = faint
- 3 = italic
- 4 = underline
- 5 = slow blink
- 6 = fast blink
- 7 = reverse foreground/background colours
- 8 = invisible text





# Maths



- Bash can't handle floating point calculations
  - lacks operators for certain important mathematical functions.
- BC
  - arbitrary precision calculation utility, bc offers many of the facilities of a programming language.
  - It has a syntax vaguely resembling C.
- Since it is a fairly well-behaved UNIX utility, and may therefore be used in a pipe, bc comes in handy in scripts.
- Quick usage guide  
[https://en.wikipedia.org/wiki/Bc\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Bc_(programming_language))





# Debug



- The Bash shell contains no built-in debugger
  - only bare-bones debugging-specific commands and constructs.
  - Syntax errors or outright typos in the script generate cryptic error messages that are often of no help in debugging a non-functional script.



# Debug

```
#!/bin/bash
# This is a buggy script.
# Where, oh where is the error?
a=37
if [$a -gt 27 ]
then
    echo $a
Fi
exit $? # 0! Why?
```

```
A0004358:CS210Scripts 210799e$ bash
buggy
buggy: line 6: [37: command not found
A0004358:CS210Scripts 210799e$
```

# Tools for Debug

- Inserting echo statements at critical points in the script to trace the variables
- Using the tee filter to check processes or data flows at critical points.

```

                                (redirection)
                                |----> to file
                                |
=====|=====
command ---> command ---> |tee ---> command ---> ---> output of pipe
=====|=====

cat listfile* | sort | tee check.file | uniq > result.file
#               ^^^^^^^^^^^^^^^^^^  ^^^^

# The file "check.file" contains the concatenated sorted "listfiles,"
#+ before the duplicate lines are removed by 'uniq.'
```



# Tools for Debug

- Setting option flags -n -v -x
  - `bash -n scriptname` checks for syntax errors without running the script.
  - `bash -v scriptname` echoes each command before executing it.
  - `bash -nv scriptname` gives a verbose syntax check.
  - `bash -x scriptname` echoes the result each command, but in an abbreviated manner.



# Tools for Debug

- Using an "assert" function to test a variable or condition at critical points in a script.
- Using the \$LINENO variable and the caller builtin.
- Trapping at exit.
  - The exit command in a script triggers a signal 0, terminating the process, that is, the script itself.
  - It is often useful to trap the exit, forcing a "printout" of variables, for example. The trap must be the first command in the script.

# Tools for Debug

- A signal is a message sent to a process, either by the kernel or another process, telling it to take some specified action (usually to terminate).
- For example, hitting a Control-C sends a user interrupt, an INT signal, to a running program

```
trap '' 2
# Ignore interrupt 2 (Control-C).
trap 'echo "Control-C disabled."' 2
# Message when Control-C pressed.
```

# Tools for Debug

## ➤ Trapping at exit

```
#!/bin/bash
# Hunting variables with a trap.
trap 'echo Variable Listing --- a = $a b = $b' EXIT
# EXIT is the name of the signal generated upon
# exit from a script.
#
# The command specified by the "trap" doesn't execute until
#+ the appropriate signal is sent.
echo "This prints before the \"trap\" -"
echo "even though the script sees the \"trap\" first."
a=39
b=36
exit 0
```



# Tracing a Variable

- The DEBUG argument to trap causes a specified action to execute after every command in a script. This permits tracing variables

```
trap 'echo "VARIABLE-TRACE> \$$variable = \"$$variable\""' DEBUG
```

```
A0004358:CS210Scripts 210799e$ bash tracing
VARIABLE-TRACE> $variable = ""
VARIABLE-TRACE> $variable = "29"
VARIABLE-TRACE> $variable = "29"
  Just initialized $variable to 29 in line number 4.
VARIABLE-TRACE> $variable = "29"
VARIABLE-TRACE> $variable = "87"
VARIABLE-TRACE> $variable = "87"
  Just multiplied $variable by 3 in line number 6.
VARIABLE-TRACE> $variable = "87"
```




# Additions in Ver3 Bash

1. `$BASH_ARGC`
  - Number of command-line arguments passed to script, similar to `$#`.
2. `$BASH_ARGV`
  - Final command-line parameter passed to script, equivalent `${!#}`.
3. `$BASH_COMMAND`
  - Command currently executing.
4. `$BASH_EXECUTION_STRING`
  - The option string following the `-c` option to Bash.
5. `$BASH_LINENO`
  - In a function, indicates the line number of the function call.
6. `$BASH_REMATCH`
  - Array variable associated with `=~` conditional regex matching.
7. `$BASH_SOURCE`
  - This is the name of the script, usually the same as `$0`.
8. `$BASH_SHELL`



# Introduction to Regular Expressions

- Regular expressions are **patterns** that only certain commands are able to interpret
- Patterns should be protected by strong quotes (single quotes)
- Regular expressions are expanded to match certain sequences of characters in text
- The `grep` command is very useful in demonstrating regular expressions



Operator	Effect
.	Matches any single character.
?	The preceding item is optional and will be matched, at most, once.
*	The preceding item will be matched zero or more times.
+	The preceding item will be matched one or more times.
{N}	The preceding item is matched exactly N times.
{N,}	The preceding item is matched N or more times.
{N,M}	The preceding item is matched at least N times, but not more than M times.
-	represents the range if it's not first or last in a list or the ending point of a range in a list.
^	Matches the empty string at the beginning of a line; also represents the characters not in the range of a list.
\$	Matches the empty string at the end of a line.
\b	Matches the empty string at the edge of a word.
\B	Matches the empty string provided it's not at the edge of a word.
\<	Match the empty string at the beginning of word.
\>	Match the empty string at the end of word.



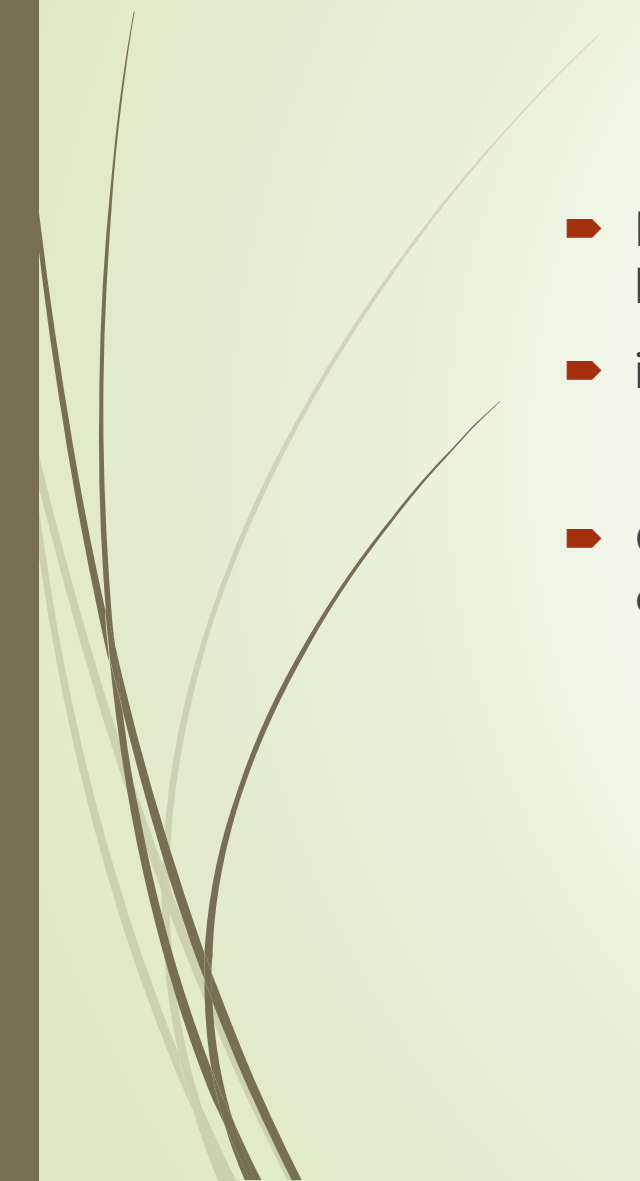
# Regular Expressions



- Two regular expressions may be concatenated
  - the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated subexpressions.
- Two regular expressions may be joined by the infix operator “|”
  - the resulting regular expression matches any string matching either subexpression.



# Basic versus extended regular expressions

- In basic regular expressions the metacharacters "?", "+", "{", "|", "(", and ")" lose their special meaning
  - instead use the backslashed versions "\?", "\+", "\{", "\|", "\(", and "\)".
  - Check in your system documentation whether commands using regular expressions support extended expressions.
- 

# Examples using grep

- `grep` searches the input files for lines containing a match to a given pattern list. When it finds a match in a line, it copies the line to standard output (by default)

```
grep root /etc/passwd  
root:x:0:0:root:/root:/bin/bash  
operator:x:11:0:operator:/root:/sbin/nologin
```



# Examples using grep

```
grep -n root /etc/passwd
```

```
1:root:x:0:0:root:/root:/bin/bash
```

```
12:operator:x:11:0:operator:/root:/sbin/nologin
```

```
grep -v bash /etc/passwd | grep -v nologin
```

```
sync:x:5:0:sync:/sbin:/bin/sync
```

```
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
```

```
halt:x:7:0:halt:/sbin:/sbin/halt
```

```
news:x:9:13:news:/var/spool/news:
```

```
mailnull:x:47:47::/var/spool/mqueue:/dev/null
```

```
xfs:x:43:43:X Font Server:/etc/X11/fs:/bin/false
```

Shows which users are not using bash, but accounts with the nologin shell are not displayed



# Examples using grep

- lines starting with the string "root"

```
grep ^root /etc/passwd  
root:x:0:0:root:/root:/bin/bash
```

- which accounts have no shell assigned whatsoever, search for lines ending in ":"

```
grep :$ /etc/passwd  
news:x:9:13:news:/var/spool/news:
```

- find a string that is a separate word (enclosed by spaces), it is better to use the -w

```
grep -w / /etc/fstab  
LABEL=/ / ext3 defaults 1 1
```



# Character classes

- A bracket expression is a list of characters enclosed by "[" and "]".
- It matches any single character in that list
  - if the first character of the list is the caret, "^", then it matches any character NOT in the list.
  - For example, the regular expression "[0123456789]" matches any single digit.
- Within a bracket expression, a range expression consists of two characters separated by a hyphen.
  - It matches any single character that sorts between the two characters.
  - "[a-d]" is equivalent to "[abcd]".
  - NOTE: it might be equivalent to "[aBbCcDd]"



# Example

```
grep [yf] /etc/group
sys:x:3:root,bin,adm
tty:x:5:
mail:x:12:mail,postfix
ftp:x:50:
nobody:x:99:
floppy:x:19:
xfs:x:43:
nfsnobody:x:65534:
postfix:x:89:
```



# Wildcards

- Use the "." for a single character match.
- To get a list of all five-character English dictionary words starting with "c" and ending in "h":

```
~> grep '\<c...h\>' /usr/share/dict/words  
catch  
clash  
cloth  
coach  
couch  
cough  
crash  
crush
```

# Wildcards

- For matching multiple characters, use the asterisk. This example selects all words starting with "c" and ending in "h"

```
~> grep '\<c.*h\>' /usr/share/dict/words
```

```
caliph
```

```
cash
```

```
catch
```

```
cheesecloth
```

```
cheetah
```

```
--output omitted--
```

- to find the literal asterisk character in a file or output, use single quotes

```
~> grep * /etc/profile
```

```
~> grep '*' /etc/profile
```

```
for i in /etc/profile.d/*.sh ; do
```



# Exercises



1. Display a list of all the users on your system who log in with the Bash shell as a default.
2. From the `/etc/group` directory, display all lines starting with the string "daemon".
3. Print all the lines from the same file that don't contain the string.
4. Display localhost information from the `/etc/hosts` file, display the line number(s) matching the search string and count the number of occurrences of the string.
5. Display a list of `/usr/share/doc` subdirectories containing information about shells.
6. How many README files do these subdirectories contain? Don't count anything in the form of "README.a\_string".
7. Make a list of files in your home directory that were changed less than 10 hours ago, using `grep`, but leave out directories.
8. Put these commands in a shell script that will generate comprehensible output.
9. Can you find an alternative for `wc -l`, using `grep`?
10. Using the file system table (`/etc/fstab` for instance), list local disk devices.
11. Make a script that checks whether a user exists in `/etc/passwd`. For now, you can specify the user name in the script, you don't have to work with arguments and conditionals at this stage.
12. Display configuration files in `/etc` that contain numbers in their names.



# Globbing Introduction



- Globs, also called “wildcards”, are special characters to the shell designed to match filenames used for manipulating (listing, copying, moving, etc.) groups of files
- Three types:
  - \* = match zero or more of any character
  - ? = match exactly one character
  - [ ] = match exactly one character from a range of characters
- Learn more: `man 7 glob`





# “\*” Wildcard Examples

- Display all files in the current directory:  
`$echo *` or `$ls *`
- Display all files in the current directory that begin with the letter D:  
`$echo D*` or `$ls D*`
- Display all files in the current directory that begin with "D" and have an "n":  
`$echo D*n*` or `$ls D*n*`



# “?” Wildcard Examples

- Display all files in the current directory that have exactly one character in file name:  
`$echo ?` or `$ls ?`
- Display all files in the current directory that begin with the letter D and have three more characters:  
`$echo D???` or `$ls D???`

# “[ ]” Wildcard Examples

- Display all files in the current directory that begin with "a", "b" or "c":  
`$echo [abc]*` or `$ls [abc]*`  
`$echo [a-c]*` or `$ls [a-c]*`
- When using a range ([a-c]), the range is based on the ASCII text table
- Display all files in the current directory that don't begin with "a", "b" or "c":  
`$echo [^a-c]*` or `$ls [^a-c]*`

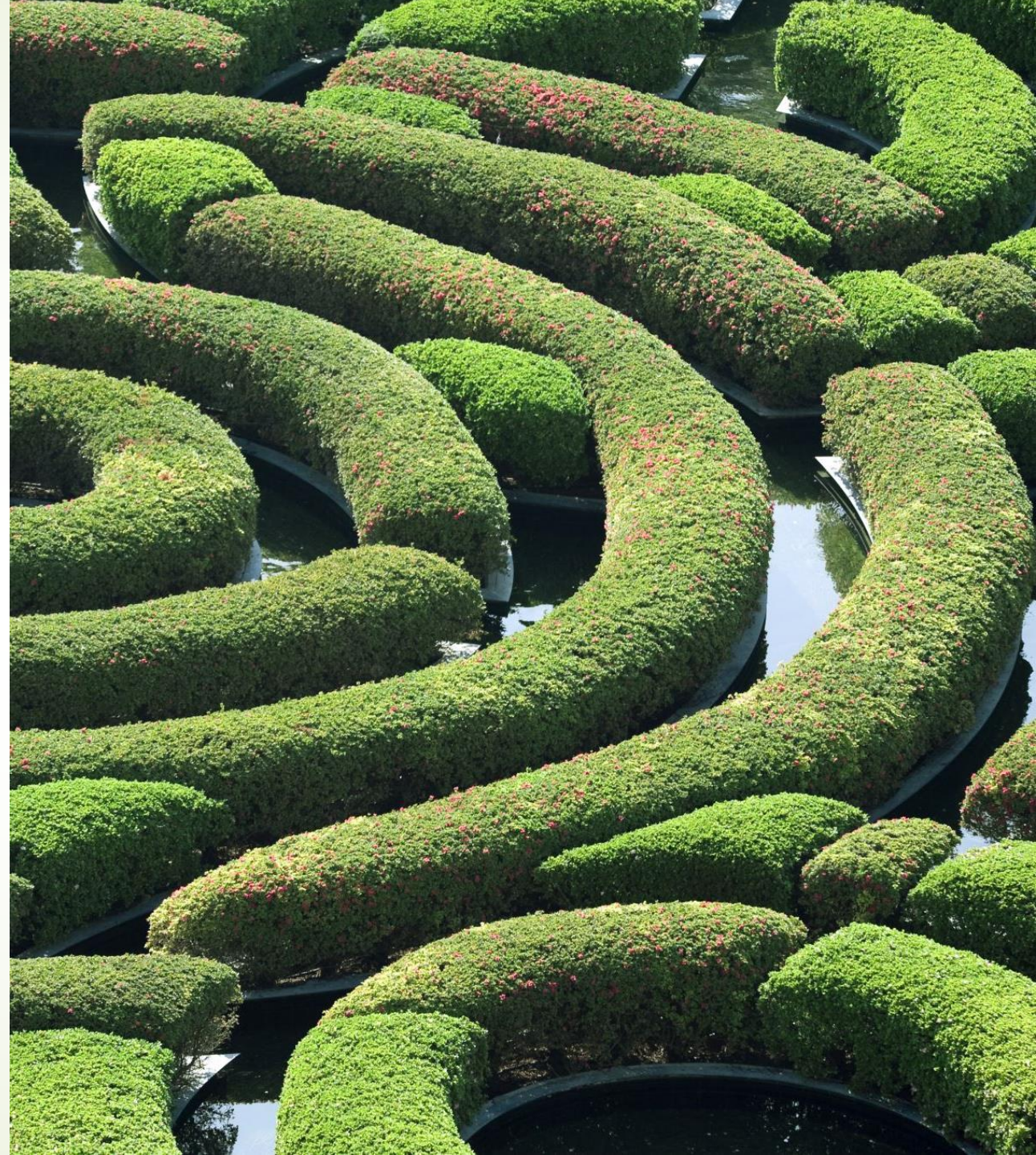
# More complex Examples

- Display all files in the current directory that begin with "a", "b" or "c" and are at least 5 characters long:  
`$echo [abc]????* or $ls [abc]????*`
- Display all files in the current directory that don't end with "x", "y" or "z":  
`$echo*[!xyz] or $ls*[!xyz]`





# Finding Files







# The locate Command

- Useful for fast searches of a filesystem for a file(s)
- Uses a database which is updated once a day
- Database updated by administrator using the `updatedb` command
- Syntax:  
    `$locate filename`

# The locate Command

- The locate command only will return results of files that the current user would normally have access to
- The locate command will display all files that has the search term anywhere in the file name
- The locate command is case sensitive

```
cs@csl123456789:~$ locate "Test File"  
/home/ftpuser/ftp/files/Test File.txt  
cs@csl123456789:~$
```





# The locate Command

- Advantages

- Fast. Searches a database of all files on the computer verses the live filesystem the find command uses

- Disadvantages:

- The database is only updated once per day so new files are not in the database
  - You can only search for files by name verses other search criteria the find command supports



# The Find Command

- Searches a live filesystem for specified file(s)
- Supports different search criteria options
- Searches the live filesystem which can take a large amount of time
- Slower than the locate command
- Example: `find / -name file_name(s)`

# The find Command Options

Example	Meaning
-iname LOSTFILE	Case insensitive search by name
-mtime -3	Files modified less than three days ago
-mmin -10	Files modified less than ten minutes ago
-size +1M	Files larger than one megabyte
-user joe	Files owned by the user joe
-nouser	Files not owned by any user
-empty	Files that are empty
-type d	Files that are directory files
-maxdepth 1	Do not use recursion to enter subdirectories; only search primary directory

# The whereis Command

- Displays the directory location and man page for the specified command
- Searches only the directories defined by the \$PATH variable
- Example:

```
A0004358:~ 210799e$ whereis grep  
/usr/bin/grep
```

```
cs@cs1123456789:~$ whereis locate  
locate: /usr/bin/locate /usr/share/man/man1/locate.1.gz  
cs@cs1123456789:~$ whereis find  
find: /usr/bin/find /usr/share/man/man1/find.1.gz /usr/share/info/find.info.gz  
cs@cs1123456789:~$ whereis whereis  
whereis: /usr/bin/whereis /usr/share/man/man1/whereis.1.gz  
cs@cs1123456789:~$
```

# The which Command

- Displays the directory location(s) of the specified command or script
- Returns the location of the real command
- Searches only the directories defined by the \$PATH variable
- The -a option is used to locate multiple executable files. Useful to know if an executable script was inserted maliciously to override an existing command.

```
cs@cs123456789:~$ which locate
/usr/bin/locate
cs@cs123456789:~$ which find
/usr/bin/find
cs@cs123456789:~$ which which
/usr/bin/which
cs@cs123456789:~$
```



# Which example

```
$ which grep
```

```
grep: /bin/grep /usr/share/man/man1/grep.1.gz
```



# The sed Command

- The **s**tream **e**ditor command (sed) can be used to modify text
- Sed is a non-interactive stream editor. It receives text input, (stdin or from a file), performs certain operations on specified lines of the input, one line at a time, then outputs the result to stdout or to a file.
- Sed determines which lines of its input that it will operate on from the address range passed to it.
- Specify this address range either by line number or by a pattern to match.
  - For example, `3d` signals sed to delete line 3 of the input, and `/Windows/d` tells sed that you want every line of the input containing a match to "Windows" deleted.



# sed

- The sed program can perform text pattern substitutions and deletions using regular expressions
- The editing commands are similar to the ones used in the vi editor:



# Basic sed Options & Operators

Option	Effect
-e SCRIPT	Add the commands in SCRIPT to the set of commands to be run while processing the input.
-f	Add the commands contained in the file SCRIPT-FILE to the set of commands to be run while processing the input.
-n	Silent mode.
-V	Print version information and exit.

Command	Result
a\	Append text below current line.
c\	Change text in the current line with new text.
d	Delete text.
i\	Insert text above current line.
p	Print text.
r	Read a file.
s	Search and replace text.
w	Write to a file.

# Basic sed Operator Examples

Operator	Name	Effect
[address-range]/p	print	Print [specified address range]
[address-range]/d	delete	Delete [specified address range]
s/pattern1/pattern2/	substitute	Substitute pattern2 for first instance of pattern1 in a line
[address-range]/s/pattern1/pattern2/	substitute	Substitute pattern2 for first instance of pattern1 in a line, over address-range
[address-range]/y/pattern1/pattern2/	transform	replace any character in pattern1 with the corresponding character in pattern2, over address-range (equivalent of tr)

# More Complex Examples

Notation	Effect
8d	Delete 8th line of input.
/^\$/d	Delete all blank lines.
1,/^\$/d	Delete from beginning of input up to, and including first blank line.
/Jones/p	Print only lines containing "Jones" (with -n option).
s/Windows/Linux/	Substitute "Linux" for first instance of "Windows" found in each input line.
s/BSOD/stability/g	Substitute "stability" for every instance of "BSOD" found in each input line.
s/ *\$//	Delete all spaces at the end of every line.



# Sed Examples

~> cat -n example

1 This is the first line of an example text.

2 It is a text with errors.

3 Lots of errors.

4 So much errors, all these errors are making me sick.

5 This is a line not containing any errors.

6 This is the last line.

~>

- We want sed to find all the lines containing our search pattern, in this case "errors". We use the p option to obtain the result:

# Sed Examples

```
~> sed '/errors/p' example
This is the first line of an example text.
It is a text with errors.
It is a text with errors.
Lots of errors.
Lots of errors.
So much errors, all these errors are making me sick.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the last line.
```

~>

- As you notice, sed prints the entire file, but the lines containing the search string are printed twice. In order to only print those lines matching our pattern, use the -n option:

```
~> sed -n '/errors/p' example
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
```

~>

# Sed Examples

- In the example file, we will now search and replace the errors instead of only (de)selecting the lines containing the search string.

```
~> sed 's/erors/errors/' example
```

```
This is the first line of an example text.
```

```
It is a text with errors.
```

```
Lots of errors.
```

```
So much errors, all these erors are making me sick.
```

```
This is a line not containing any errors.
```

```
This is the last line.
```

```
~>
```

- this is not exactly the desired effect: in line 4, only the first occurrence of the search string has been replaced, and there is still an 'eror' left.

- Use the g command to indicate to sed that it should examine the entire line instead of stopping at the first occurrence of your string:

```
~> sed 's/erors/errors/g' example
```

```
This is the first line of an example text.
```

```
It is a text with errors.
```

```
Lots of errors.
```

```
So much errors, all these errors are making me sick.
```

```
This is a line not containing any errors.
```

```
This is the last line.
```

```
~>
```



# Sed Examples

- To insert a string at the beginning of each line of a file, for instance for quoting:

```
y~> sed 's/^/> /' example
> This is the first line of an example text.
> It is a text with errors.
> Lots of errors.
> So much errors, all these errors are making me sick.
> This is a line not containing any errors.
> This is the last line.
~>
```


- Insert some string at the end of each line:

```
~> sed 's/$/EOL/' example
This is the first line of an example text.EOL
It is a text with errors.EOL
Lots of errors.EOL
So much errors, all these errors are making me sick.EOL
This is a line not containing any errors.EOL
This is the last line.EOL
~>
```



# Non-interactive editing

- Reading sed commands from a file
- Multiple sed commands can be put in a file and executed using the -f option.
  - No trailing white spaces exist at the end of lines.
  - No quotes are used.
  - When entering text to add or replace, all except the last line end in a backslash.
- Writing output files
  - Writing output is done using the output redirection operator >. This following is an example script used to create very simple HTML files from plain text files.



```
~> cat script.sed
1i\
<html>\
<head><title>sed generated html</title></head>\
<body bgcolor="#ffffff">\
<pre>
$a\
</pre>\
</body>\
</html>

~> cat txt2html.sh
#!/bin/bash
# This is a simple script that you can use for converting text into HTML.
# First we take out all newline characters, so that the appending only happens
# once, then we replace the newlines.
echo "converting $1..."
SCRIPT="/home/sandy/scripts/script.sed"
NAME="$1"
TEMPFILE="/var/tmp/sed.$PID.tmp"
sed "s/\n/^M/" $1 | sed -f $SCRIPT | sed "s/^M/\n/" > $TEMPFILE
mv $TEMPFILE $NAME
echo "done."
~>
```



# Result

```
~> txt2html.sh test
converting test...
done.

~> cat test
<html>
<head><title>sed generated html</title></head>
<body bgcolor="#ffffff">
<pre>
line1
line2
line3
</pre>
</body>
</html>
~>
```



# Sed Exercises



1. Print a list of files in your scripts directory, ending in ".sh".. Put the result in a temporary file.
2. Make a list of files in /usr/bin that have the letter "a" as the second character. Put the result in a temporary file.
3. Delete the first 3 lines of each temporary file.
4. Print to standard output only the lines containing the pattern "an".
5. Create a file holding sed commands to perform the previous two tasks. Add an extra command to this file that adds a string like "\*\*\* This might have something to do with man and man pages \*\*\*" in the line preceding every occurrence of the string "man". Check the results.
6. A long listing of the root directory, /, is used for input. Create a file holding sed commands that check for symbolic links and plain files. If a file is a symbolic link, precede it with a line like "--This is a symlink--". If the file is a plain file, add a string on the same line, adding a comment like "<--- this is a plain file".
7. Create a script that shows lines containing trailing white spaces from a file. This script should use a sed script and show sensible information to the user.



# awk



- Awk is a full-featured text processing language with a syntax reminiscent of C.
- Awk breaks each line of input passed to it into fields. By default, a field is a string of consecutive characters delimited by whitespace.
  - Awk parses and operates on each separate field.
  - Making it ideal for handling structured text files, especially tables
  - Strong quoting and curly brackets enclose blocks of awk code within a shell script.

# awk Examples

```
# $1 is field #1, $2 is field #2, etc.
```

```
echo one two | awk '{print $1}'
```

```
# one
```

```
echo one two | awk '{print $2}'
```

```
# two
```

```
# But what is field #0 ($0)?
```

```
echo one two | awk '{print $0}'
```

```
# one two
```

```
# All the fields!
```

```
awk '{print $3}' $filename
```

```
# Prints field #3 of file $filename to stdout.
```

```
awk '{print $1 $5 $6}' $filename
```

```
# Prints fields #1, #5, and #6 of file $filename.
```

```
awk '{print $0}' $filename
```

```
# Prints the entire file!
```

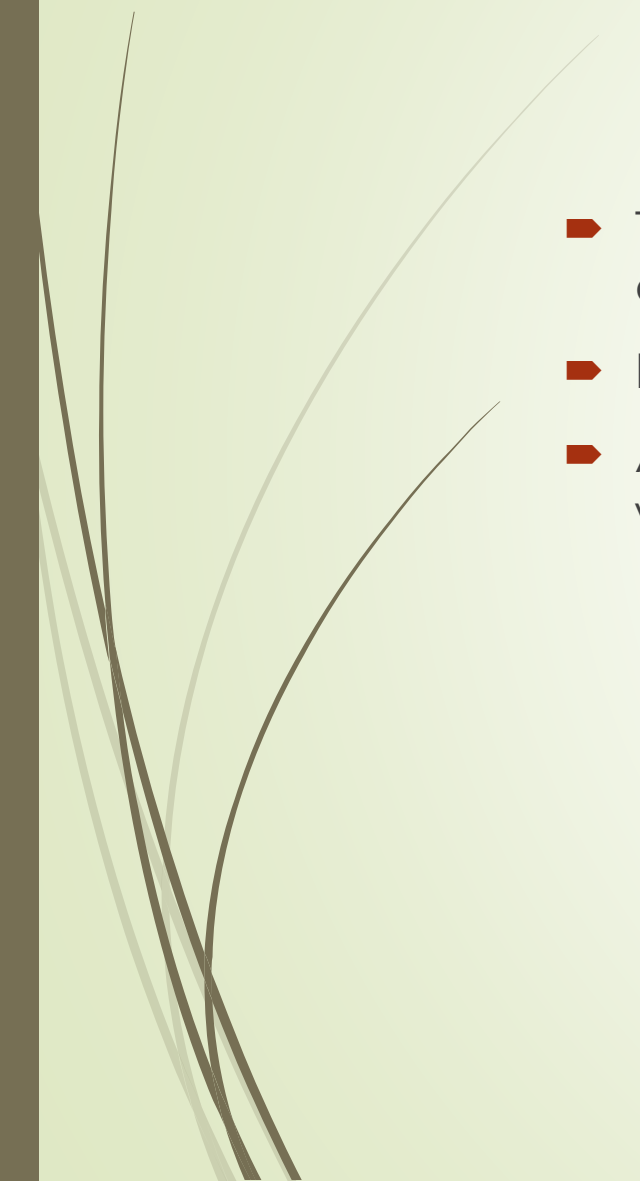


# gawk GNU version of awk

- gawk is used for arithmetic and string operation using structured programming concepts such as loop and if to generate formatted :
    - \$0 entire line
    - \$1 first field
    - \$n nth fields
    - NR line number
    - NF Fields number
    - -F -indicates separator
- echo "my name is john" | gawk '{\$print \$4="Mary";print \$0}': my name is Mary



# Introduction to FHS

- The Filesystem Hierarchy Standard (FHS) is standard that specifies standard directories and their content for use with a filesystem
  - Helps to know what directories expect to find and what to find in them
  - Allows programmers to write programs that will be able to work across a wide variety of systems that conform to this standard
- 



# History of FHS

- First known as known as the Filesystem Standard (FSSTND)
- Renamed in 1997 with series 2
- Final series 2 (2.3) published in 2004
- Draft version of series 3 published in 2001
  - Still under review
  - Unofficial changes ongoing



# Recent unofficial changes

- New directory additions:

- `/run` to contain volatile data that changes at runtime

- `/sys` to hold files related to the kernel

- Directory merging:

- `/bin` merged into `/usr/bin`

- `/sbin` merged into `/usr/sbin`

- `/lib` merged into `/usr/lib`

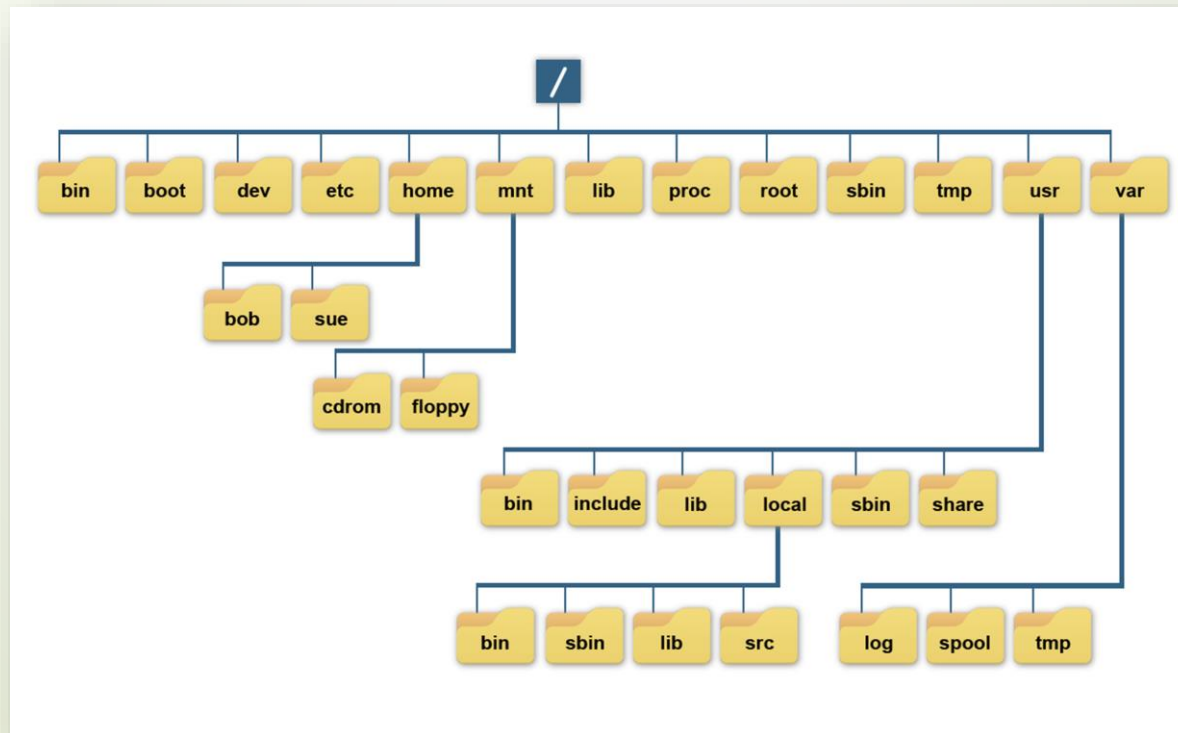


# Directory classifications

- If a directory structure is classified **as shareable**, then it typically does not contain anything that would be unique to a particular system like a configuration file
- If a directory structure is classified **as static** it means that it usually doesn't change and may suggest that it might be mounted read-only
- If a directory structure is classified **as variable**, it is likely to change and would have to be available for both read and writes

# Important directories

- FHS standard details many directories
- Administrators should know those on the next three slides







# Important directories



Directory	Purpose
<b>/</b>	The root of the primary filesystem hierarchy
<b>/bin</b>	Contain essential user executables
<b>/boot</b>	Contain the kernel and bootloader files
<b>/dev</b>	Populated with files representing attached devices
<b>/etc</b>	Configuration files specific to the host
<b>/home</b>	Common location for user home directories
<b>/lib</b>	Essential libraries to support /bin and /sbin executables
<b>/mnt</b>	Mount point for temporarily mounting a filesystem
<b>/opt</b>	Optional third party add-on software
<b>/root</b>	Home directory for the root user
<b>/sbin</b>	Contains system or administrative executables

# Important directories (cont.)

Directory	Purpose
<b>/srv</b>	May contain data provided by services of the system
<b>/tmp</b>	Location for creating temporary files
<b>/usr</b>	The root of the secondary filesystem hierarchy
<b>/usr/bin</b>	Contains the majority of the user commands
<b>/usr/include</b>	Header files for compiling C-based software
<b>/usr/lib</b>	Shared libraries to support /usr/bin and /usr/sbin
<b>/usr/local</b>	The root of the third filesystem hierarchy for local software
<b>/usr/sbin</b>	Non-vital system or administrative executables
<b>/usr/share</b>	Location for architecturally-independent data files
<b>/usr/share/dict</b>	Word lists



# Important directories (cont.)

Directory	Purpose
<b>/usr/share/doc</b>	Documentation for software packages
<b>/usr/share/info</b>	Information pages for software packages
<b>/usr/share/locale</b>	Locale information
<b>/usr/share/man</b>	Location for man pages
<b>/usr/share/nls</b>	Native language support files