

Unix Crash Course



Unix Crash Course

Slide 1

\$ who am i

- Rob Wolfram
- Unix System administrator
- E-mail: r.s.wolfram@amsterdamumc.nl
- PGP Key: **0xF7A0F7A0**

Unix Crash Course

Slide 2

Subjects (before lunch)

- A short history of Unix and Linux
- Structure and philosophy of Unix
- Files and filesystems
- Shell variables and globbing
- Processes and jobs

Unix Crash Course

Slide 3

Subjects (after lunch)

- Networking and the X windowing system
- Shell scripting concepts
- Regular expressions
- **sed** and **awk**
- Miscelaneous (editing and programming)

Unix Crash Course

Slide 4

Mumbo Jumbo?



Unix Crash Course

Slide 5

Unix History

- AT&T abandoned MULTICS
- Ken Thompson & Dennis Ritchie developed UNICS on a PDP7
- UNIX ported to Ritchie's C language
- BSD released, many ports followed
- POSIX effort to unify Unices
- GNU project / Linux kernel

Unix Crash Course

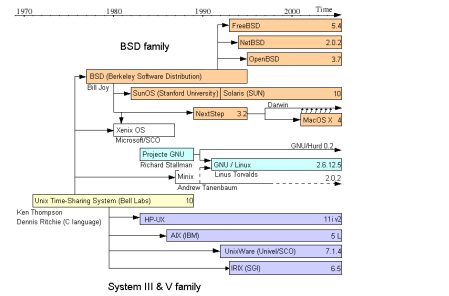
Slide 6

Unix History



Ritchie and Thompson working on a PDP11/20 at Bell Labs

Unix History



(Gnu/) Linux distros

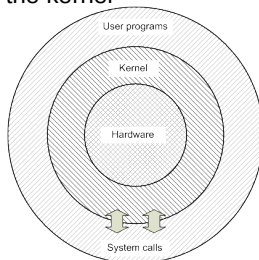


Unix philosophy

- Everything is a file
- Combine small tools to build your requirement

Unix structure

Unix has an onion-like structure, the hardware is handled by the kernel



Kernel provisions

The kernel provides:

- CPU scheduling of processes
- Accessing hardware
- System calls for user-land programs
- The filesystem
- Privilege separation etc.

Multi-user environment

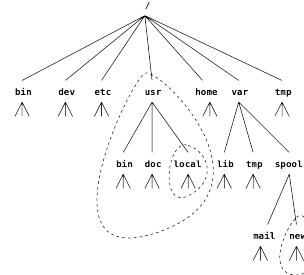
- Unix runs programs from multiple users concurrently, even *interactive* programs
- User identified by numeric id, user name provided for verbosity
- User is member of one or more groups, identified by group id
- Interactive sessions need a TTY for input and output

Unix Crash Course

Slide 13

The filesystem

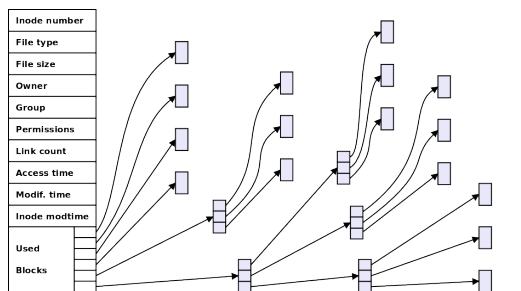
A single unified tree of multiple filesystems



Unix Crash Course

Slide 14

Inodes



Unix Crash Course

Slide 15

File permissions

```
-rwxr-xr-x 1 root wheel 109944 Nov 19 2012 /bin/lS
```

↓ user permissions
↓ group permissions
↓ world permissions
↓ link count
↓ user
↓ group
↓ size
↓ modification time
↓ file name
↓ file type

Unix Crash Course

Slide 16

Lab 1

- (page 10, execute **setup_labs.sh**)
- `ls -l`
- `cat README`
- `chmod a+r README`
- `ls -l`
- `cat README`

Unix Crash Course

Slide 17

The Shell

- There are various shells available:
 - **sh**: the Bourne shell
 - **csh**: the C shell
 - **ksh**: the Korn shell
 - **tcsh**: the TENEX C shell
 - **bash**: the Bourne-Again shell
 - **zsh**: the Z shell

Unix Crash Course

Slide 18

The Shell

- The shell interprets entered commands
Syntax: `command argument_list`
- Arguments are separated by white space
Certain arguments (usually single characters preceded by a dash change the program's behaviour and are called "options")
- The shell will perform command substitution, variable expansion and globbing and execute the command with modified command line. All these steps depend on quoting.

Unix Crash Course

Slide 19

Frequent commands

```
man      chmod      wc
echo     chown     more
read     chgrp     date
ls        umask     time
cp        cat        tar
mv        head      gzip
ln        tail      compress
rm        cut        xargs
pwd       grep      tee
cd        sed       expr
mkdir    sort      awk
rmdir    uniq      find
```

Unix Crash Course

Slide 20

Shell variables

Using shell variables:

```
$ echo $HW
$ HW="Hello, World."
$ echo $HW
Hello, World.
$
```

Export variables to make them visible in a sub-shell:

```
$ export VARIABLE
```

Use curly braces to disambiguate variables:

```
$ echo ${VARIABLE}more_text
```

Unix Crash Course

Slide 21

Special variables

```
$PATH          $0
$MANPATH       $1 - $9
$LD_LIBRARY_PATH $#
$HOME          $*
$USER          @$
$PWD           $?
$SHELL        $$
$PS1           !$
$PS2
```

Unix Crash Course

Slide 22

Quoting

Variables expand in double quotes, not in single quotes. Backslashes "escape" a single character:

```
$ HW="Hello, World."
$ echo "$HW"
Hello, World.
$ echo '$HW'
$HW
$ echo \$HW
$HW
$ echo \\$HW
\Hello, World.
$
```

Unix Crash Course

Slide 23

Command substitution

Use "backquotes" for command substitution:

```
$ wc log.`date +%Y%m%d`
   1      8      54 log.20131102
$
```

On modern shells, `$()` is allowed. This enables nesting:

```
bash$ wc log.$(expr $(date +%Y%m%d) - 100)
   630    1616   40744 log.20131002
bash$
```

Unix Crash Course

Slide 24

Globbering

- The asterisk (*) expands to *zero* or more characters (e.g. "ls foo*")
- The question mark expands to exactly one character (e.g. "ls /etc/?asswd")
- Characters in square brackets expand to *one* character from the list. Ranges are allowed. ("ls foo. [abc0-9]"). Negate the list with an exclamation mark ("ls foo. [!abc0-9]").

Globbering

- On modern shells, the tilde (~) expands to the users homedirectory and "~foo" to the homedirectory of user "foo"
- Globbering is handled by the shell, the executed command doesn't know if globbering occurred
 - Notice that this can cause an error of an oversized argument list

Lab 2

(page 16 – 17)

- File & directory management
- Variable substitution
- Command substitution
- Globbering

Redirection

- Three file descriptors:
 - FD0 is standard input (stdin)
 - FD1 is standard output (stdout)
 - FD2 is standard error (stderr)
- More are available for advanced use
- Redirect output with > and input with < (e.g. "command 1> file" or "command 0< file" (FD is optional for stdin and stdout)

Redirection

- Connect file descriptors with >& construct: (command > file 2> &1)
- > overwrites the output file or create a new one, >> will append to the file instead
- << is called a "here document" and used in scripts.

Pipes

- A pipe connects stdout of a command to stdin of the next
This is central to the Unix philosophy, i.e. create small but powerful tools and connect them
Example:
ls /tmp | wc -l
- stderr can't be piped alone, only with stdout

tee and xargs

- Two commands used a lot with pipes: `tee` and `xargs`. Examples:
- Save log output and count entries:

```
grep 10.1.2.3 /var/log/apache/access.log \  
| tee /tmp/rogueclient.txt | wc -l
```
- Search for text in files that are less than 4 days old:

```
find /var/log -mtime -4 -print | xargs \  
grep -l 'kernel error'
```

Grouping

Combine stdout of multiple commands with `()` or `{}`. Parentheses work in sub-shell, braces in current shell:

```
$ echo Foo ; echo Bar | wc  
Foo  
    1      1      4  
$ { echo Foo ; echo Bar ; } | wc  
    2      2      8  
$ (echo Foo ; echo Bar) | wc  
    2      2      8  
$
```

Lab 3

(page 19)

- Redirection and piping

Forking

- The shell will start a “child process”. The command will be executed in this process.
- After the child exits, it signals the “parent”.
- Changed environment in child is not visible in the parent (variables, current directory)
- Parent variables should be “exported” to be visible in the child (`export VARNAME`)
- Executing a shell in current process is called “sourcing” (`. scriptfile`)

Processes

- Every process has a “process id”
- Use `ps` to retrieve information of processes
- Use `kill` to send processes a signal
Some signals (e.g. `SIGINT`) are handled by the program, others (e.g. `SIGKILL`) are handled by the kernel

Jobs

- A process can be started in the background by appending an ampersand (`&`) to the CL
- A program is suspended by sending it a `SIGSTOP` (e.g. by pressing `CTRL-Z`)
- `jobs` gives a list of all suspended and backgrounded processes
- `fg` and `bg` continue running a process in the foreground or background respectively (job id may be appended with `%` sign)

Scheduling

- Run a program unattended later with `at`:

```
echo "find /tmp -mtime +30 | xargs rm -f"\  
| at 20:08 tomorrow
```
- Schedule regularly with `cron`. Syntax:
`min hou dom mon dow command [arguments]`
Example:
`5 * * 3,6 2 echo foo >> /tmp/myfile`

Shell Initialization

- The shell will source files on login or other startup.
 - `sh, ksh`: `/etc/profile, $HOME/.profile` (on login)
 - `bash`: `/etc/profile, $HOME/.bash_profile, $HOME/.profile` (on login)
`/etc/bash.bashrc, $HOME/.bashrc` (interactive)

Lunch

See you in an hour

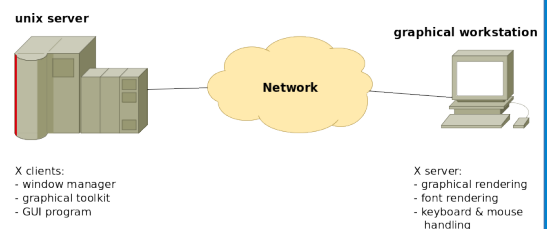
Networking

- Interactive shells and file sharing can be started from networked hosts
- Common tools:
 - `telnet`
 - `ftp`
 - "rsh" tools (`rsh, rlogin, rcp`)
 - secure shell suite (`ssh, scp, sftp`)
- Pseudo-TTY's are assigned to interactive networked shells

The X Windowing System

- The standard Unix GUI (X) is networked based. Consists of an "X server" (which can display graphics and handle keyboard and mouse) and an "X client" (a program requesting graphical output).
- The X server is identified by the `$DISPLAY` variable (e.g. `myscreen.example.com:0.0`)

The X Windowing System



X server access

- Host based access: (dis)allow all users access to the X server. Syntax: `xhost +|- [hostname]`
- Cookie based access. List cookie on X server and add it to the `.xauthority` file from the user running the X client. `xauth` is used for cookie management
- `ssh` can automate the `xauth` process and pass X traffic via encrypted tunnel.

Shell scripting

- A “shebang” is needed to tell the OS what script language is used. Syntax: `#!/bin/sh`
- Functions are “named groupings” and are not executed at time of declaration. Syntax: `shfunc() { commandList ; }`
- *Here document* redirects stdin from the script:
`command << WORD`
 first line of stdin
 last line of stdin
`WORD`

Shell flow: if

- Syntax:
`if command`
 `then`
 `command list`
 `elif command`
 `then`
 `command list`
 `else`
 `command list`
 `fi`
- Alternative:
`command1 && command2 || command3`

Shell flow: case

- Syntax:
`case string in`
 `valuelist1)`
 `command list`
 `;;`
 `valuelist2)`
 `command list`
 `;;`
 `...`
 `valuelistn)`
 `command list`
 `;;`
`esac`

Shell flow: case

- Valuelists consist of one or more patterns to match against the string, separated by pipes (|)
- Shell globbing syntax is allowed when matching the string
- Only the first matching entry is executed

Shell flow: while

- Repeat a block of commands as long as the constraint is valid. Syntax:
`while command`
 `do`
 `command list`
 `done`
 or
 `until command`
 `do`
 `command list`
 `done`
- Exit or restart the loop with `break` or `continue`

Shell flow: **for**

- Repeat a loop a number of times while assigning a value to a variable. Syntax:

```
for VAR in value-list
do
  command list
done
```
- The value-list consists of whitespace-separated values.
- **break** and **continue** are valid in **for** loops.
- Bash allows a C-like syntax:

```
for ((expr1;expr2;expr3)) ; do list ; done
```

test

- The **test** command is used very often for flow control. The syntax is:

```
test expression OR
[ expression ]
```
- Expressions can be tested for strings, numbers or files.

test examples

- ["\$VAR" = foo] - Test string equality
- [-z "\$VAR"] - Test \$VAR as empty string
- ["\$VAR" -lt 12] - Numeric comparison
- [-d foo] - Is foo a directory
- [expression1 -a expression2] - Logical AND
- [expression1 -o expression2] - Logical OR
- [! expression] - Logical NOT

Lab 4

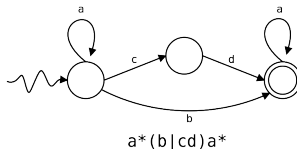
(page 28)

Create a small script with multiple names.
Alter the behaviour depending on the name.

Remember: the name of the script is stored in the variable **\$0**

Regular expressions

- Recognise the language of strings that can be expressed with a *state transition diagram*



- Used extensively in Unix, e.g. **ed**, **grep**, **vi**, **awk**, **perl**, **python**, etc.

Regular expressions

- **abc** – Concatenated characters are recognized as such
- **.** – The dot is a placeholder for any character
- ***** – The asterisk represents zero or more repetitions of the previous character
- **[abc0-9]** – A single character in the brackets is recognized, **0-9** is a range
- **[^abc0-9]** – A caret as first bracketed character negates the list.

Regular expressions

- `^` and `$` bind to the empty string at the beginning and end of a line respectively
- `\<` and `\>` bind to the empty string at the beginning and end of a line respectively
- `\|` is the logical OR between two regexps
- `\(` (and `\)`) can be used to group part of a regexp that can be referenced as `\n`, where `n` is the number of the `nth` grouping.

Extended regexps

- The `?` recognizes zero or one repetitions of the previous character or group
- The `+` recognizes one or more repetitions
- `{n,m}` recognizes at least `n` and at most `m` repetitions. Either `n` or `m` is optional. A single `n` recognizes exactly `n` repetitions.
- The characters `(`, `)` and `|` are not escaped in extended regexps

sed

- `sed` is a *stream editor*. It will change the text of stdin or the file(s) in the arguments and send the result to stdout.
- A `sed` command can be preceded by a range definition. If the range is omitted, all lines are submitted to the command.
- Lines that are unaffected by either the range or the command are printed verbatim to stdout.

sed

- The range takes the form of `a,b` where both `a` and `b` can be either a line number or a regexp indicating the first line where the regexp matches.
- Example: the command `1,/^$/d` will delete all text from the first line to the first empty line.
- Multiple commands are grouped in braces (`{}`) with each command on a separate line.

sed

Some common sed commands:

- Substitute: `s/regexp/newtext/flags`
`\n` and `&` references are available in RHS
- Delete: `/regexp/d`
- Append: `atext` or insert `itext`.
A single range token is mandatory. Newlines must be escaped with a backslash (`\`)
- Transliterate: `y/fromchars/tochars/`
Replace all occurrences from LHS with corresponding character from RHS

awk

- All commands consist of an optional pattern followed by a block of statements in braces:
`pattern { statements }`
`pattern { statements }`
...
- All lines that pass the `pattern` constraint are subjected to the statements
- The `BEGIN` and `END` patterns indicate statements that are executed before and after reading the input respectively

awk

Patterns can be:

- A regexp (`/pattern/`)
- A relational expression (`$4 < 15`)
- A boolean construct of patterns (`&&`, `||` and `!`)
- Alternate pattern evaluation (C syntax):
`pattern ? pattern : pattern`
- A range (`pattern1, pattern2`)
- Special pattern `BEGIN` or `END`

awk

- The input line is divided in "fields" (`$1`, `$2`, etc) separated by whitespace. `$0` is the whole line.
- Variables can be string or numeric, or an array of variables. Array indexes are associative and placed in square brackets (`[]`).
- Statements in a block are separated by newlines or semicolons (`;`). A statement can be an *action statement* (like `print`) or a *flow statement* (`if`, `for`, `do while`, etc.) with statement blocks of their own.

awk

- Example: Fibonacci numbers

```
awk 'BEGIN { cnt=0
        a=0; b=1
        while (cnt < 10)
        { cnt++
          c=a+b; a=b; b=c
          print "Fib(" cnt ") is " c
        }
    }'
```

Text editing

Editing text is a frequent task in Unix systems.

Some text editors are:

- `vi` (present on about every Unix system)
- `emacs`
- `pico` / `nano`
- `ed` (if all else fails)

Notice the difference in line endings between Unix and other OS-es

Programming

- Most Unix systems come with a C compiler preinstalled.
The GNU project has development environments for many other languages (C++, Fortran, Java, Pascal etc.)
- Use `make` to automate compile and link tasks
- Many scripting languages are available, often not by default (perl, PHP, Python, etc).

Screen

If you have a long running job, start a shell inside `screen`

- `screen -r` to reconnect a disconnected session
- `Ctrl+A D` to disconnect
- `Ctrl+A C` to create a new shell
- `Ctrl+A N` or `Ctrl+A P` to cycle through shells
- `Ctrl+A ?` for help

Mumbo Jumbo Revisited



```
find * -name "CV*" -group jewerkisjehobby \  
-exec egrep -il '(creatief|innovatief)' \  
{ } \; | xargs awk ' $1 == "email" { \  
print $2 }' | xargs Mail -s \  
"Je baan is in Hilversum" \  
-body http://www.omroep.nl/gurus/
```

Mumbo Jumbo Corrected

```
find * -name "CV*" -group jewerkisjehobby \  
| xargs egrep -il '(creatief|innovatief)' \  
| xargs awk '$1 == "email" { print $2 }' \  
| while read addr ; do \  
echo http://www.omroep.nl/gurus | Mail -s \  
"Je baan is in Hilversum" $addr ; done
```

Try it yourself



Download an Ubuntu DVD from:
<https://www.ubuntu.com/download/desktop>
Burn it to a USB stick with Rufus (<https://rufus.ie/>)
This is a live CD that you can try on a PC or Mac
without overwriting an existing OS
... and it's FREE! (both types)

^D

Thank you for your interest.

This presentation will be available online at
<https://unix.hamal.nl/>