

UNIX

michael.fink@uibk.ac.at 2017

Overview

History

Practicalities

User Interface

Shell

Toolbox

Processes

Users, Groups

File System

Development System

Cluster Computing

UNIX / Linux History

1969

- UNIX at Bell Labs (antithesis to MULTICS, Thompson, Ritchie, et al.)
goals: simple & general time sharing system which supports
programming, authoring, collaboration
- (Re)written in C (→ portable), internal use only (chance to mature)



Thompson & Ritchie @ PDP-11

UNIX / Linux History

1970s

- Given to universities → BSD
- Commercialization → System V



- **Concepts**
simplicity, “everything is a file”, toolbox (small specialized utilities, shell = glue)

1980s – early 2000s

- Widespread adoption by market, UNIX wars + diversification (DEC, HP, IBM, SGI, SUN, ...) → Standardization (**POSIX**)
- HPC: graphical RISC workstations & servers, Vector Computers



SGI Personal IRIS



Convex C220

UNIX / Linux History

1990 - 2010s Linux

- GNU core utilities & toolchain (reimplementation of UNIX utilities & compilers), but no kernel
- Linus Torvalds develops new kernel and ports GNU utilities → GNU/Linux
- widespread adoption by end users, academia, and industry
- runs on commodity hardware (mobile devices, desktops, large computers)
- diversification + bloat (distributions, uncontrolled development + duplication of features)

Important UNIX versions 2010s: Linux, 4.4BSD, AIX, Solaris, macOS

In this tutorial:

UNIX generically refers to any type of UNIX-like OS
Linux specifically refers to GNU/Linux

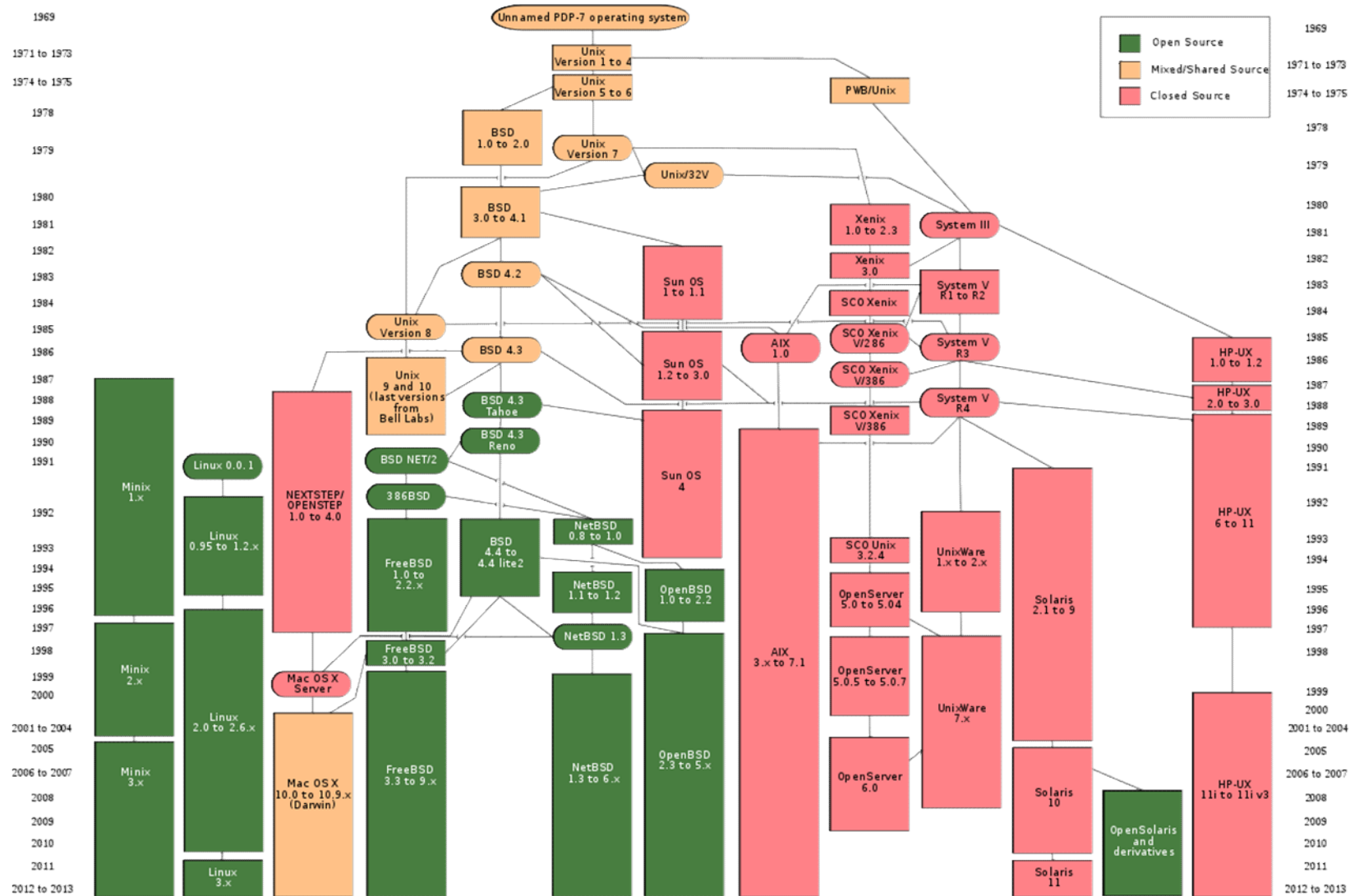


HPC and Linux

- Beowulf clusters (Donald Becker, Thomas Sterling)
networked commodity machines
- HPC Clusters
Small SMP machines coupled by high performance interconnect (IB)
Parallel programming using message passing (MPI)
- Shared Memory Systems (SGI Altix, Ultraviolet)
SMP → ccNUMA
100s-1000s CPUs share large memory (10s TB), single OS instance



UNIX / Linux History



UNIX heritage tree (simplified)

Relevance
many implementations of
the same utilities
(portability issues)

Linux: GNU utilities
have been ported to many
other platforms

UNIX - basic concepts

OS Layer Model

- Kernel
 - Process management (schedule processes)
 - Memory management (for processes and devices)
 - File system (files and directories)
 - Device drivers (disks, terminals, networking etc.)
- All interaction through System Calls
- Library Calls provide abstracted interface

Everything Is A File

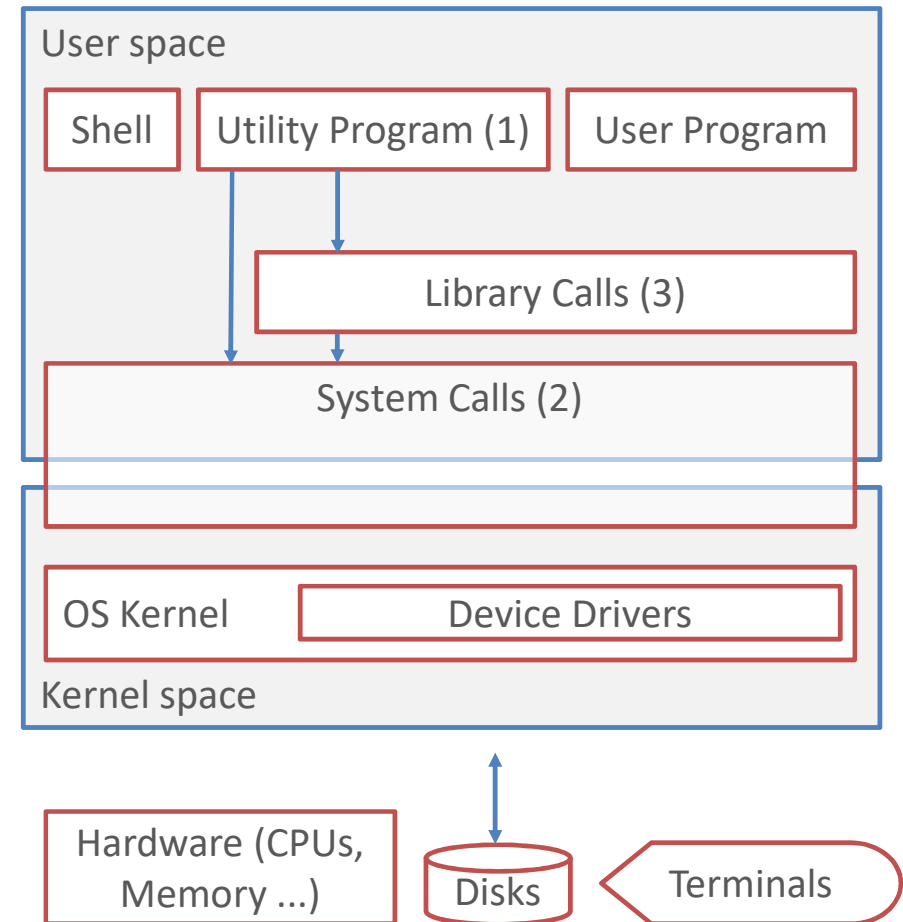
Files, Terminal I/O (/dev/tty), Disks (/dev/sdx) ...

Technically, **shell**, **utility programs**, and **user programs** are **on the same level**

Utility programs: simple, specialized:

- “**just do one thing**” (e.g. copy file, edit text, search text ...)
- do **work silently** → can use utilities to **build programs**
- **brief error messages** identifying **root cause** of error

Shell: provides interactive + programmable interface to OS, used to orchestrate utilities and user programs



Accessing a Server

Server

- get account (workgroup: sysadmin; large sites: apply for account)

Preparing your workstation

Linux

- make sure `X11` + `openssh-client` are installed
- `$HOME/.ssh/config`: `ForwardX11 yes`
- **using:** start terminal emulation (e.g. `xterm`) and issue `ssh hostname.domainname`

Windows

- Install `ssh` client (includes terminal emulation - `putty`) and X11 server (`Xming` + `Xming-fonts`)
- `putty`: enable X11 forwarding (`connection/ssh/X11`)
- **using:** start `putty` (and `Xming` if needed) and enter `hostname.domainname` in Host Name field

Common Commands, Working with Files

- Change password `passwd`
 - Log out `exit`
 - Read documentation `man command` display the “man-page” for *command* (`./..`)
 - Display contents of file `cat file [...]` copy contents of all named files to terminal window
`less file [...]` browse file contents
 - Edit text file `nano file` many other editors available.
`vi file` UNIX standard editor (Linux: replaced by `vim`) (*)
`nedit file` (Bill Joy: An Introduction to Display Editing with Vi)
NCSA editor (graphical, needs X11)
 - List words in command line `echo word [...]` useful for checking and debugging
- (*) not easy to learn, very powerful, very popular,
only editor that works well on slow network connections

Using Less to Browse Files and Man Pages

less [*option* ...] [*file* ...] browse contents of named *files* or *stdin*. **man** uses **less** to display man page

- c repaint screen from top of window
- i ignore case in searches
- s squeeze multiple blank lines

default options may be set in environment variable \$LESS

commands while browsing (single key stroke - similar to vi) - may be prefixed with number *n*

h		help
q		quit
f	^F SPACE	scroll forward <i>n</i> lines (default: full screen)
b	^B	scroll backward - " -
d	u	scroll forward / backward <i>n</i> lines (default: half screen)
j	k	scroll forward / backward <i>n</i> (default: 1) lines
r	^R ^L	repaint screen
g		go to beginning of file
G		go to line <i>n</i> (default: end of file)
F		go to end of file and watch it growing (terminate with ^C)
/	<i>pattern</i> ? <i>pattern</i>	forward / backward search for <i>pattern</i>
n	N	repeat previous search in opposite direction
ESC	u	undo highlight of search results
:n	:p :x	view next / previous / first file
^G		view info about current file (name, size, current position)

Using vi and vim to Edit Files

{**vi**|**view**|**vim**} *file* [...] invoke standard UNIX editor **vi** (edit/readonly) or its improved derivative **vim**

Concept

Editor has three modes: **insert mode** (text entry): start with commands like **i** **a** (insert, append), end with **ESC** key
command mode (cursor movement, text buffer manipulations): shorthand English
many commands can be prefixed with number *n*
command line entry: start from command mode with **:**, end with **ENTER** key

Commands	Mnemonic	Function
h j k l	keys next to each other	cursor movement ← ↓ ↑ → (j think ^J) one (<i>n</i>) characters
0 \$ ENTER	\$ matches EOL in regex	cursor to beginning / end of current line, beginning of text in next line
H M L	home middle last	cursor movement to first, middle, last line in window
w b W B	word back	cursor movement one (<i>n</i>) words forward / backwards; nonblank words
^F ^D ^B ^U	forward down backward up	scroll forward (full page, half page), backward (full page, half page)
^E ^Y	end (?)	scroll forward, backward one (<i>n</i>) lines: cursor stays put
zENTER z.	zap	scroll current line to beginning/center of window, cursor stays on line
i a I A	insert append	start insert mode before/after cursor; “beginning” / end of current line
x dw d<i>n</i>w dd D	scissor, delete...	delete character, word, <i>n</i> words, rest of current line
s <i>n</i>s S	substitute	start insert mode, overwriting 1, <i>n</i> characters, entire line
cw c<i>n</i>w C	change	start insert mode, overwriting 1, <i>n</i> words, rest of current line
:w :w!	write	write current file, force overwrite even if readonly
:q :q!	quit	quit, force quit discarding changes
ZZ :x	eXit	write file if changes were made, then quit.

Recommendation: to avoid clobbering files, do NOT use **wq**, NEVER EVER use **wq!**

Why: **wq** modifies file although no changes were made; **wq!** forces inadvertent changes to go to file

Further vi Commands and Remarks

More cursor movement commands

Commands	Mnemonic	Function
<code>f</code> <i>c</i> <code>t</code> <i>c</i> <code>F</code> <i>c</i> <code>T</code> <i>c</i> <code>;</code> <code>,</code> <code>/</code> <i>string</i> <code>ENTER</code> <code>?</code> <i>string</i> <code>ENTER</code> <code>n</code> <code>N</code>	Find character <i>c</i> , up To <i>c</i> / delimits regex	move cursor to first (n'th) instance of <i>c</i> or to preceding character same to the left repeat last f or t in same / opposite direction forward search <i>string</i> (actually regex) in file backward search <i>string</i> in file repeat last search in same / opposite direction

Deleting, copying and moving text

Structure of commands: single-letter commands `d` `y` `c` take address (cursor movement command) as argument
doubling command `dd` `yy` `cc` refers to entire line, prefix (*n*)

Commands	Mnemonic	Function
<code>c</code> <i>address</i> <code>cc</code> <code>S</code> <i>n</i> <code>cc</code> <code>d</code> <i>address</i> <code>dd</code> <i>n</i> <code>dd</code> <code>y</code> <i>address</i> <code>yy</code> <i>n</i> <code>yy</code> <code>p</code> <code>P</code>	change delete yank put	change text from cursor to <i>address</i> delete data, copy to unnamed <i>buffer</i> copy data to unnamed <i>buffer</i> insert <i>buffer</i> contents after / before cursor

Using named buffers: prefix these commands by "*c*" (single character a-z: 26 named buffers)

Recommended reading

Bill Joy, "An Introduction to Display Editing with Vi"
vim online documentation

Getting Information - Accessing System Documentation

UNIX man pages and GNU documentation

<code>man [section] name</code>	view description of command or utility <i>name</i> in <i>section</i> (default: first found)
<code>man section intro</code>	view introduction to <i>section</i>
<code>man -k string</code>	search man pages for <i>string</i>
<code>info [name]</code>	start GNU info browser
<code>info [name] less</code>	browse GNU documentation for <i>name</i> (if installed by sysadmin) using <code>less</code> as browser

man sections

- 1 user commands
- 2 system calls (interface to operating system kernel)
- 3 library functions
- 4 special files (interface to devices)
- 5 file formats
- 6 games
- 7 miscellany
- 8 administrative commands

Referring to documentation

In texts: `name(section)` e.g.: The `ls(1)` command lists files and directories.
The `fopen(3)` library routine internally uses the `open(2)` system call

Working with Directories

Concepts

Data organized in tree structure, starting at “root” (/), “directory” ≈ “folder”

Each file and directory has full (absolute) path starting with / . example: /home/c102mf/mydir/myfile

Path components separated by / character

File names are case sensitive

CWD (current working directory): starting point for relative path

Example: if CWD = /home/c102mf/mydir then myfile → /home/c102mf/mydir/myfile

Special directories

\$HOME	your HOME directory. CWD after login	
.	this directory	} present in every directory
..	parent directory	

e.g. if CWD = /home/c102mf/mydir then

./myfile	→	/home/c102mf/mydir/myfile
../myfile	→	/home/c102mf/myfile

Commands

- Display working directory `pwd` or `/bin/pwd`
- List directory contents `ls [-l] [-a] [-R]` (-l display attributes -a include „hidden“ -R recursive)
- Create directory `mkdir name`
- Change working directory `cd [name]` (default: \$HOME)
- Remove directory `rmdir directory` *directory must be empty*

Copying, Moving, and Deleting Data

Commands

- Copy files
 - `cp file1 file2` copy contents of *file1* to *file2*
 - `cp file [...] directory` copy *files* to target *directory*
 - `cp -r [-p] [-a] dir1 dir2` recursively copy directory
(-p keep permissions -a keep everything)
- Rename or move files
 - `mv file1 file2` rename *file1* to *file2*
 - `mv name [...] directory` move named *files* or *directories* to target *directory*
- Remove (=delete file)
 - `rm [-i] [-f] file [...]` remove named *files* (-i ask before removing)
 - `rm -r [-f] directory` recursively remove *directory* and all of its contents
(-f force, suppress errors)

Note: removing a file does not free the occupied space if the file is still held open by a running process.

Understanding File Systems & Capacity

Concept

File system = collection of files + directories on partition (or disk or disk array)

Capacity of file system (data + metadata) limited by size of partition

All file systems organized in tree (mount points)

Commands

`df [-h]` display mount point and capacity of all file systems (-h human readable)

`df [-h] .` display file system info for working directory

`quota -v` display your file system quota (if defined by sysadmin)

Transferring Files Between Workstation and Server

Windows use WinSCP graphical client

Note Text file format differs between
UNIX (Lines separated by LF) and
Windows (Lines separated by CRLF)

Use TEXT MODE for text files and BINARY MODE for binary files
STANDARD MODE often guesses wrong - will damage binary files

Linux use scp (command line, similar to cp) or sftp (interactive)

```
scp file [...] user@server:[targetdirectory]      upload files
scp -r directory [...] user@server:[targetdirectory]  upload directories
```

```
scp user@server:[directory]/file [...] targetdirectory  download files
scp -r user@server:directory [...] targetdirectory      download directories
```

```
sftp user@server      start interactive file transfer session
{ cd | lcd } directory  remotely / locally change to directory
get file [localfile]    download
put file [remotefile]    upload
```

Motivation: Understanding the UNIX Toolbox

UNIX design principles

Main interface: **Shell** = **command interpreter** (interactive + programming language)
makes functionality of **UNIX kernel** available to user

shells **POSIX shell** (portable), **bash** (standard Linux shell, ported to many UNIXes), many others

Individual **utilities** (**ls**, **cp**, **grep**) do the work

- external programs, **simple** + **specialized**
- standardized **data flow model**:
 - read data from **stdin** or **set of files** given in command line
 - write data to **stdout** - can be used by other programs or shell
 - most utilities avoid side effects: usually **do not modify other files**

Shell = glue. Orchestrates utilities & puts them to work

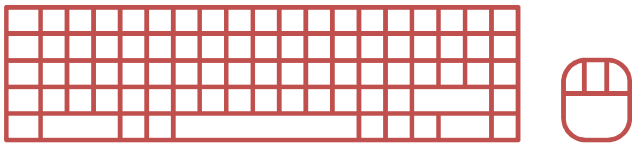
- Controls execution of programs: start, foreground, background
- Assembles command lines:
 - from **command line arguments** typed by user
 - can create **list of files** from **wildcards** (e.g. *, ?)
 - can use output from other commands
 - to create parts of or entire command lines (**command substitution**)
- Establishes data flow between programs: I/O redirection, pipelines
- Full-featured programming language: variables, loops, conditionals

“**Everything is a file**”, strongly **text** based

Understanding Command Entry and Execution

Terminal (-emulation)

```
$ ls -l x.txt  
-rw-r----- 52 c102mf c102 x.txt  
$
```



Server

Shell (/bin/bash) = **command interpreter**
issue prompt (\$)
read and parse command line (**ls -l xxx.txt**)
look up command (**built-in**, **\$PATH**)
execute command (**/bin/ls -l xxx.txt**) and wait

/bin/ls

read directory contents and file attributes
output results

issue prompt

Understanding the UNIX Command Line

Command line is sequence of **words**, separated by **white space**.

command [*argument* ...] → argv[0] argv[1] ...

Conventions

- first word is **name** of command (**built-in** or external **program**)
- arguments are processed **sequentially** by program **avoid positional arguments** in your own programs
- arguments can be **options** or **operands**
- **options** modify **behavior** of programs (**how**), results should be independent of order
varying conventions, sometimes used alternatively in same program. most common:
 - x** [*value*] single letter preceded by minus, directly followed by option value if required by option (**POSIX**)
option letters may be grouped: -**x** -**y** equivalent to -**xy** , -**y** -**x** , -**yx**
 - option** [*value*] option word preceded by one minus character
 - option**[=*value*] option word preceded by two minus characters,
followed by equal sign and argument if required (**GNU**)
 - xyz** [*value* ...] first argument contains single letter options, followed by values in order if required (**BSD**)
and more - see man pages
- remaining arguments are **operands** (often files; **what**)
warning: some programs ignore this convention and allow options after operands

Understanding the UNIX Command Line

Examples

`ls -l -a directory` same as `ls -la directory`

`cp [-p] file1 file2` make copy under different name

`cp [-p] file1 [...] directory` make copies of multiple files in target directory

`cc -O3 myprog.c mysub.c -o myprog` compile two program sources (max optimization) + create executable *myprog*

Same commands may allow mixed conventions.

Most well-known: **GNU-tar** (create and extract files to/from archive) and **ps**

POSIX

`tar -c -v -f my_project.tar.gz -z my_project`

BSD

`tar cvfz my_project.tar.gz my_project`



GNU

`tar --create --verbose --file=my_project.tar.gz --gzip my_project`

Automating Work: Using Wildcards to Supply File Names

Wildcards: automatically create **list of file names** as operands for commands
replacements done **by shell** before program is started

wildcard	meaning
*	0 or more arbitrary characters
?	1 arbitrary character
[abcs-z]	square brackets: character class (enumeration or range): one of a , b , c , s , t , ... z

Examples

<code>rm *.o</code>	remove all object files. warning: what happens with <code>rm * .o ?</code>
<code>ls -l *. [ch]</code>	all C source and header files. almost equivalent: <code>ls -l *.c *.h</code>
<code>mv *.txt *.doc mydir</code>	move all .txt and .doc files into directory <i>mydir</i>

Counterexample

<code>mv *.for *.f</code>	does not work (why?)
---------------------------	----------------------

Understanding Standard Input / Output

Concept

Fundamental for toolbox design of UNIX

Every process has three preconnected data streams

descriptor	name	usage
0	stdin	standard input
1	stdout	standard output
2	stderr	standard error - error messages

Normally: all three connected to terminal (`tty`)

Programs conforming to this convention are called **filters**



Shell Syntax: Redirecting standard input / output to files

redirections done by shell before program starts

<code>command < inputfile</code>	<code>command</code> takes its <code>stdin</code> from <code>inputfile</code>
<code>command > outputfile</code>	<code>command</code> writes its <code>stdout</code> to <code>outputfile</code> (existing file is overwritten)
<code>command >> outputfile</code>	<code>command</code> appends its <code>stdout</code> to <code>outputfile</code>
<code>command 2> errorfile</code>	<code>command</code> writes its <code>stderr</code> to <code>errorfile</code>
<code>command >&2</code>	redirect <code>stdout</code> of <code>command</code> to current <code>stderr</code>

Combining redirections (examples)

<code>command < infile > outfile 2> errfile</code>	connect all three streams to separate files
<code>command < infile > outfile 2>&1</code>	send <code>stderr</code> to same stream as <code>stdout</code> (order matters)

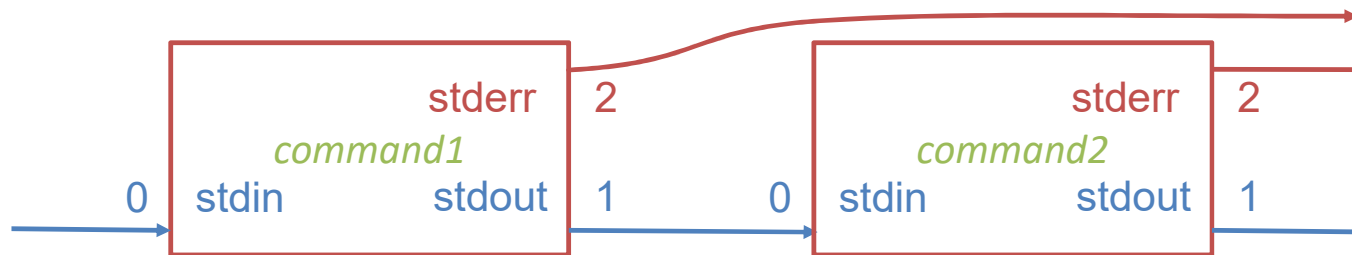
Connecting Programs: Building Pipelines

Concept

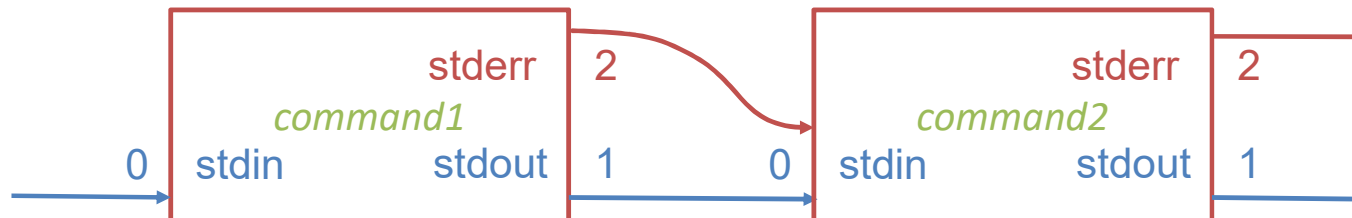
Pipeline: program sends its `stdout` directly to `stdin` of another program

Shell Syntax

`command1 | command2 [| ...]` `stdout` of `command1` is directly connected to `stdin` of `command2`
`command1`'s `stderr` is still connected to original `stderr` (default: `tty`)



`command1 2>&1 | command2` `stdout` and `stderr` of `command1` are sent to `stdin` of `command2`



pipeline set up by shell before commands start

Understanding Shell Variables + Environment Variables

Concepts

Three ways to pass information to program

- Input data (`stdin`; program may open any `files`)
- Command line arguments
- Environment variables

Environment variables

environment is set of `name=value` pairs, many predefined
used to modify (default) behavior of programs

environment copied (`one way`) from `parent` to `child` when program starts `main(int argc, char *argv[], char *envp[])`
accessing environment: `getenv` (C, Matlab), `os.environ['name']`, (Python), `$name` (bash), `$ENV{name}` (perl)...

Shell variable (AKA parameter)

Shell has local variables, can be connected to environment of current shell by `export` command

Convention

Most system- or software-specific variables have UPPERCASE names

Shell syntax and commands

<code>name=value</code>	set shell variable <code>name</code> to <code>value</code> (no whitespace around =)
<code>export name</code>	associate shell variable <code>name</code> with environment
<code>export name=value</code>	set <code>name</code> to <code>value</code> and associate with environment
<code>\$name</code> <code>\${name}</code>	use variable: substitute <code>value</code> in command line (use {...} to separate from adjacent characters)
<code>env</code>	print all environment variables to stdout
<code>name=value [...] command</code>	run <code>command</code> in temporarily modified environment

Important Environment Variables - Examples

\$HOME	User's HOME directory Default for cd command Many programs search initialization files in \$HOME (convention: \$HOME/.xxxx "invisible")
\$USER	Login name of current user
\$PATH	Colon-separated list of directories - searched by shell for executable programs e.g.: PATH=/bin:/usr/bin:/usr/local/bin:/usr/X11/bin
\$SHELL	Which command interpreter to use in shell-escapes e.g.: SHELL=/bin/bash
\$EDITOR	Which editor to use e.g.: EDITOR=/usr/bin/vi
\$TERM	Type of terminal emulator - needed by editors, less, and other screen-oriented software e.g.: TERM=xterm
\$HOSTNAME	Name of host (server) on which command runs
\$PS1 \$PS2	Shell's command prompt strings; bash: many macros
\$DISPLAY	Network address of X Server - needed by all X11 clients (GUI programs) e.g.: DISPLAY=localhost:12.0
\$TEMP	Directory where some programs put their temporary files. Default: /tmp
\$SCRATCH	Set by some installations: location of scratch directory

Using the PATH Variable to Determine Search for Programs

Syntax

PATH=*directory:directory:...* colon-separated list of directories

Examples

PATH=/home/c102mf/bin:/usr/local/bin:/usr/bin:/bin

PATH=/home/c102mf/bin:/usr/local/bin:/usr/bin:/bin: empty entry means . (CWD)

Semantics

when *command* is entered, shell

- tests if *command* is *alias* or *shell-builtin*. If yes, run this. Else...
- if *command* contains / , try to locate executable using *command* as path to file. Else, shell...
- searches executable file *directory/command* for each *directory* component of \$PATH

first match is executed.

Important recommendation

- CWD (.) should be *avoided* in PATH, but if necessary, put it as *empty* (*) entry *at end*
why: other users may plant Trojans in directories with public write access (e.g. /tmp)
example: executable */tmp/ls* may contain malicious code.
`cd /tmp ; ls` triggers this code if CWD comes in PATH before legitimate directory
- (*) why empty: so one can extend PATH: `PATH=${PATH}/usr/site/bin:`

Querying where command is

which command prints path of *command* that would be executed to *stdout*

Example: `which ls` → `/bin/ls`

Protecting Parts of Command Line Against Shell Substitutions

Concept

Shell

reads command line

substitutes **wildcards** and **variables** (special characters: * ? [] \$)

breaks result into **words** at whitespace (**blank**, **tab**) → **arguments** for program

This behavior may be changed by quoting

Syntax

'...'	Text between single quotes: literally preserve all special characters and whitespace, one argument
"..."	Double quotes: expand variables , preserve other special characters and whitespace, one argument
\	Escape character: suppress special meaning of following character

Examples

<code>rm 'my file'</code>	Remove a file with blank character in its name
<code>rm my\ file</code>	Same

<code>a='my file'</code>	
<code>rm \$a</code>	tries to remove two files 'my' and 'file'
<code>rm "\$a"</code>	tries to remove one file named 'my file'

Command Substitution: Feeding Program Output into Command Line

Concept

The output of one command can be inserted into the command line of another command

Syntax

``command`` (old syntax - backticks) or
`$(command)` (new syntax) anywhere in a command line
`$(<file)` shorthand for `$(cat file)`

Semantics

The output of `command` is split into words using the `$IFS` environment variable (default value: `blank`, `tab`, `newline`)

The result is substituted in the embedding command line

Examples

```
which myscript → /home/c102mf/bin/myscript
vi $(which myscript) edit myscript (found via $PATH)
```

```
ls > ls.out
vi ls.out          edit list of files to be removed
rm $(<ls.out)      remove files in list
Beware of blanks in file names (set IFS to newline)
```

`command2 $(command1)`



Example: Using Command Substitution + Variables in Interactive Session

Goal

keeping track of directories for re-use in commands

How to

After `cd`-ing into some directory containing interesting files

```
p=$( /bin/pwd)      remember current working directory in variable p .  
                    /bin/pwd resolves real path to CWD, ignoring redirections by symbolic links (later)
```

Then `cd` to some other directory, and

```
q=$( /bin/pwd)      remember new working directory in variable q .
```

Later you can do things like

```
cd $p  
cp -p $q/*.c .  
tar cvf $q/../project.tar .
```

Here Documents: Feeding Shell Script Text into Stdin

Concept

The input of a command can be taken directly from the shell script

Syntax

```
command [arg ...] <<[-]WORD  
arbitrary lines of text          # this is the here-document  
WORD
```

Semantics

if *WORD* is not quoted, the here document undergoes **variable expansion** and **command substitution**
else no substitutions are made

if <<-*WORD* was used, **leading tabs are stripped** from the here-document
the resulting lines are connected to the `stdin` of `command`

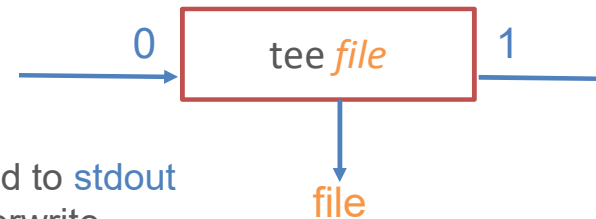
Example: verbose error message

```
cat <<EOF >&2  
Error in $0 on $(date): could not find $1 in \ $PATH = $PATH  
called as $0 "$@"  
Aborting.  
EOF
```

Useful Commands

for more information: see `man command`

duplicate data from pipeline to file(s)



`tee` [*option* ...] [*outfile* ...] copy `stdin` to each *outfile* and to `stdout`
-a append to *outfile*, do not overwrite

create text for **pipelines** and **command substitutions** - output written to `stdout` - often used in shell scripts

`date` [*option* ...] print date to `stdout`
+*format* use format e.g.: +%Y%m%d-%H%M%S

`seq` [*option* ...] [*first*] *last* print sequence of numbers to `stdout`, used e.g. in loops
-f *format* use format (like in printf(3))
-w pad output with zeros to equal width
-s *string* separate output with *string* instead of newline

`basename` *name* [*suffix*] remove directory components (and *suffix*) from *name* and print to `stdout`

`dirname` *name* strip last component from path name and print result to `stdout`

Useful Commands - Examples

Sending output of long running *program* to stdout and file

```
program | tee program.out
```

Saving program output to dated file

```
date                → Don Sep 28 09:40:14 CEST 2017
LANG=en_US date     → Thu Sep 28 09:43:32 CEST 2017
date +%Y%m%d-%H%M%S → 20170928-094027
```

```
program > program.$(date +%Y%m%d-%H%M%S).out
```

Using basename and dirname to dissect path names

```
file=/home/c102mf/project/src/main.c
basename $file      → main.c
basename $file .c   → main
dirname $file        → /home/c102mf/project/src
```

Useful Filter Commands (1)

use these commands to filter text in **pipelines** and **command substitutions**

for all commands: if no files are given, read input from stdin

for more options see **man command**

cat [*option* ...] [*file* ...]

-n

-s

concatenate *file* contents and print to **stdout**

number output lines

suppress repeated blank lines

sort [*option* ...] [*file* ...]

-n

-r

-f

-s

-k *fstart,fstop*

print sorted concatenation of *files* to **stdout**

numeric

reverse

ignore case - fold lower case to upper case

stable sort (do not change order of lines with same key)

key fields (start, stop), origin = 1

uniq [*option* ...] [*infile* [*outfile*]]

-c

-d

-u

eliminate repeated adjacent lines and print results to **stdout** or *outfile*
(**warning**: *file* arguments do **not** follow [*file* ...] convention)

prefix output lines by number of occurrences

print only duplicated lines

print only unique lines

Useful Filter Commands (2)

wc [<i>option</i> ...] [<i>file</i> ...]	(word count) print file name, line, word, and byte counts for each <i>file</i> to <i>stdout</i>
-C	bytes
-m	characters (different from bytes when using unicode)
-w	words
-l	lines
	note: to suppress output of file name, read from stdin
head [<i>option</i> ...] [<i>file</i> ...]	print first <i>num</i> (10) lines of each <i>file</i> to stdout
-n <i>num</i>	number of lines to print
-q	do not print headers giving file names
tail [<i>option</i> ...] [<i>file</i> ...]	print last <i>num</i> (10) lines of each <i>file</i> to stdout
-n <i>num</i>	number of lines to print
-q	do not print headers giving file names
-f	output appended data as file grows (useful for watching output of long running background program)
--pid= <i>pid</i>	(with -f : terminate when process <i>pid</i> dies)
--retry	keep trying until file is accessible
expand [<i>option</i> ...] [<i>file</i> ...]	convert tab characters to spaces
-t [<i>tab</i> [, ...]]	specify tab stops (default: 8)

Useful Filter Commands (grep)

Ethymology: editor command g/regular expression/p

`grep` [*option* ...] [-e] *pattern* [*name* ...] search contents of named *files* for matches of *pattern* and print matching lines to *stdout*

- e *pattern* protect a *pattern* (possibly) beginning with '-' from option processing
- i ignore case
- v invert match: only print **non-matching** lines
- w match only whole words

- c print only counts of matching lines
- l (list files) print only names of files containing at least one match
- s (silent) no error messages about missing or unreadable files
- q (quiet) no output, only exit status (0 .. match(es) found, 1 .. no match, 2 .. error)
- H -h prefix output lines with file name (default if more than one file) ; suppress file name prefix
- n prefix output lines with line number
- r recursively search all files in *directories*
- E use **extended regular expressions** (./.)

pattern search pattern given as **regular expression**: describes set of strings.

Regular Expressions

- used throughout UNIX utilities (`grep`, `sed`, `vi`, `awk`, `perl`; `regex` library functions). several slightly differing varieties
- similar, but **different** from shell **wildcards**
- do **not** create lists of **file names**, but **match strings**; metacharacters and their meanings differ from wildcards

Understanding Regular Expressions (1)

Regular expressions (use quotes to protect metacharacters from shell) (-E) denotes **extended regex**

simple string	<code>abc</code>	matches string <code>abc</code>	<code>grep 'abc' ...</code>	prints all lines containing <code>abc</code>
period	<code>.</code>	matches one arbitrary character. <code>a.c</code> matches <code>aac abc a8c a(c</code> but not <code>abb</code>		
character class	<code>[abc-s-z]</code>	matches one character from those between brackets <code>a[abc-s-z]c</code> matches <code>aac abc acc asc ... azc</code> but not <code>akc</code>		
negation	<code>[^abc]</code>	matches anything but enclosed characters <code>a[^mno]c</code> matches <code>abc akc azc</code> but not <code>amc aaac</code>		
anchoring	<code>^ \$ \< \></code>	match the beginning / end of line / word <code>^abc</code> matches lines beginning with <code>abc</code> , but not lines containing <code>abc</code> somewhere else <code>\<abc\></code> matches word <code>abc</code> but not <code>xabc abcy</code>		
repetition	<code>*</code>	preceding item matches zero or more times <code>ab*c</code> matches <code>ac abc abbc abbbbbbbbc</code>		
repetition (-E)	<code>+</code>	preceding item matches one or more times <code>ab+c</code> matches <code>abc abbc abbbbbbbbc ...</code> but not <code>ac</code>		
optional (-E)	<code>?</code>	preceding item matches zero times or once <code>ab?c</code> matches <code>ac abc</code> but not <code>abbc ...</code>		
count (-E)	<code>{n}</code>	preceding item matches exactly <code>n</code> times <code>ab{3}c</code> matches <code>abbbc</code> but not <code>ac abc abbc abbbbc</code>		
	<code>{n,}</code> <code>{,m}</code> <code>{n,m}</code>	preceding item matches at least <code>n</code> times, at most <code>m</code> times, from <code>n</code> to <code>m</code> times		

Understanding Regular Expressions (2)

Regular expressions (continued)

(-E) denotes **extended regex**

grouping (-E)	()	turn regex between parentheses into new item <code>x(abc)*y</code> matches <code>xy</code> <code>xabcy</code> <code>xabcabcy</code> but not <code>xaby</code> etc.
alternation (-E)		matches regex on either side of pipe character <code>x(abc def)y</code> matches <code>xabcy</code> <code>xdefy</code> but not <code>xaefy</code>
back reference (-E)	<code>\n</code>	matches what previous <i>n</i> -th group matched - counting opening (<code>x(ab cd)y\1z</code> matches <code>xabyabz</code> <code>xcdycdz</code> but not <code>xabycdz</code>

Combined example

`x(ab|cd).*\1y` matches `xababy` `xab___aby` `xcdlllcdy`

Difference between basic and extended regular expressions

In basic regex, metacharacters `? + { } | ()` have no special meaning, but `.` `*` `[]` `\` do

Using prefix `\` (backslash) reverses meaning

Grep examples

```
grep -r 'foo' .  
vi $(grep -rwl 'foo' .)  
ls -l | grep -E '\.(cc|cp|cxx|cpp|CPP|c++|C)$'  
grep -iv '^[c*]' *.f  
grep '#include[<"] [a-z0-9/]+.h[>"]' *. [ch]
```

recursively search for string `foo` in all files
edit all files that contain variable named `foo`
list all C++ source files
find non-comment lines in Fortran 77 source files
find include lines in C source and header files

Useful Filter Commands (sed)

sed [*option* ...] *script* [*file* ...] stream editor - perform text transformations on input, print result to stdout

- n suppress automatic output
- i edit in place
- r use **extended regular expressions**
- s treat files **separately**, reset line number for each file
- e *script* add *script* to sed **commands** to be executed

Commands

s/regex/replacement/[g] try to match *regex* and replace by *replacement* if successful.
suffix *g* : replace all occurrences, not only first match
can use backreferences (*\n*) to insert previously matched text

d delete line, skip to next line

p print line (useful with -n)

; { ... } separate / group commands (many more)

Commands may be prefixed by *addresses*, which select lines

<i>number</i>	match line <i>number</i>
<i>\$</i>	match last line
<i>/regex/</i>	match lines matching <i>regex</i>
<i>addr1,addr2</i>	match from <i>addr1</i> to <i>addr2</i> (can be used as a multiple on / off switch)

Examples (./)

Useful Filter Commands (sed) Examples

Examples

```
sed 's/foo/bar/'
```

replace **first** foo in each input line by bar .
e.g. foo foo → bar foo foot → bart

```
sed 's/foo/bar/g'
```

replace **all** foo by bar
foo foo → bar bar foon foot → barn bart

```
sed 's/\<foo\>/bar/g'
```

replace all entire words foo by bar
foo foo → bar bar but foot → foot

```
sed -E 's/(foo|bar)/X\1/g'
```

replace foo and bar by Xfoo and Xbar

```
sed -E 's/"([a-z]+)"/>>\1<</g'
```

replace quoted lowercase strings by same between >> and <<

```
sed 's/ */g'
```

eliminate all blanks from input

```
sed '/^begin$/,/^end$/s/foo/bar/g'
```

in all sections between pairs of lines begin and end ,
replace foo by bar

Useful Filter Commands (awk)

`awk` [*options*] *program* [*file* ...] invoke the awk pattern scanning and processing language
-F *fs* use field separator *fs* to separate input fields (default: white space)

Concept

program is sequence of *pattern* { *action* } statements

awk reads its *input lines*, splits them into *fields* \$1 \$2 and executes *action* on each line if *pattern* matches

Patterns are logical expressions involving

- regular expressions
- special words e.g. BEGIN END (match exactly once before / after all input)
- relational expressions
- operators ~ () && || ! , for matching, grouping, and, or, negation, range

Actions are statements in C-like programming language

- data types: scalar, associative array (i.e. indexed by string value)
- builtin variables: FS / OFS (input/output field separator), NF (number of fields), NR (number of input records so far)
- i/o statements: print, next, ... control statements: if, while, for....

Example

split input data into fields, lookup line, print selected fields

```
getent passwd | awk -F: ' $1 ~ /^string$/ { print $1, $3, $5 }'
```

- awk often used to extract data from program output
- Recommended reading: `info awk` | `less` (GNU awk implementation)
- see also `perl(1)`

Example from Demo

```
getent passwd | awk -F: 'BEGIN { OFS="," } $1 ~ /^c102mf$/ { print $1, $3, $5 } '
```

```
awk ' BEGIN { n = 0 ; } { n += $1 ; } END { print n ; } ,
```

Understanding Processes

Concept

process = running instance of a program

UNIX is multitasking / multiprocessing system: **many processes at any time** sharing one / many CPUs

Process **hierarchy**: every process has exactly one **parent**, may have **children**

Properties

USER UID	process owner
PID	process ID (number)
PPID	parent's process ID
CMD COMMAND	command line
STAT	process state (running, stopped, I/O wait, defunct ...)

Commands

ps -ef	display all running processes (POSIX)
	output USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
ps aux	display all running processes (BSD)
	output UID PID C STIME TTY TIME CMD
kill [-9] <i>pid</i>	terminate running process with numerical ID <i>pid</i> (-9 force termination)

Foreground + Background Processes

Concepts

Foreground process: Shell waits until process has finished (normal case)

Background process: Shell immediately issues new prompt, process runs **asynchronously**

Syntax and commands

<code>command</code>	start foreground process
<code>command &</code>	start background process
<code>nohup command &</code>	start background process - not terminated at logout
<code>\$!</code>	special parameter (later): PID of last background process
<code>wait pid</code>	wait for background process <code>pid</code> to finish. Example: <code>wait \$!</code>

Hint: compute cluster: use **batch** system to put long-running processes on cluster nodes

Examples

<code>ps -f</code>	<code>ps ux</code>	display processes under current shell
<code>ps -ef</code>	<code>ps aux</code>	display all processes on system
<code>ps -ef grep \$USER</code>		display all my processes
<code>ps -ef sed -n -e '1{p;d}; /'"\$USER"'/p'</code>		display all my processes including header line (→ script)

Manipulating Processes, Signals

Concept

Shell and TTY driver of OS (using special characters) work together to allow state changes to running program

Typically special characters are `<ctrl-x> ≈ ^x` sequences

Some actions send signals to running processes

Processes may catch signals (except -9) to perform cleanup, otherwise most signals terminate

Actions

<code>^C</code>	terminate running program (keyboard interrupt <code>SIGINT = 2</code>)
<code>^Z</code>	suspend running program: stop execution (<code>SIGTSTP</code>), may be continued in foreground or background
<code>jobs</code>	display all jobs (= individual processes or pipelines) under current shell (note: different from batch job) output contains job number <code>[n]</code> , <code>+</code> (current job) or <code>-</code> (previous job)
<code>bg %n</code>	continue job <code>n</code> in background (<code>SIGCONT</code>) <code>bg +</code> <code>bg -</code> same for current and previous jobs
<code>fg %n</code>	continue job <code>n</code> in foreground (<code>SIGCONT</code>) <code>fg +</code> <code>fg -</code> for current / previous
<code>wait %n</code>	wait for background job <code>n</code> to finish
<code>kill [-9] %n</code>	kill (suspended or background) job <code>n</code> (<code>SIGTERM = 15</code> <code>SIGKILL = 9</code>)
<code>kill [-9] pid</code>	kill process with process ID <code>pid</code> (<code>SIGTERM</code> <code>SIGKILL</code>)

`exit` Exiting login shell sends `SIGHUP = 1` to all shell's children (jobs). use `nohup` to ignore

`man 7 signal` documentation for all valid signals

Working With Terminal Sessions

Special characters

Interrupting and suspending running programs

^C **^Z** interrupt / suspend (as described before)

Erasing typos in command line

^H **^W** **^U** erase **single character**, **last word**, **entire line** from typed input

Terminating input

^D End Of File. Terminates input for any command reading from TTY
If shell reads EOF, it will exit.

Controlling terminal output

^S **^Q** stop / continue output from running program (stop start)

Suppressing special meaning of character

^V next typed character will be passed verbatim to command's input

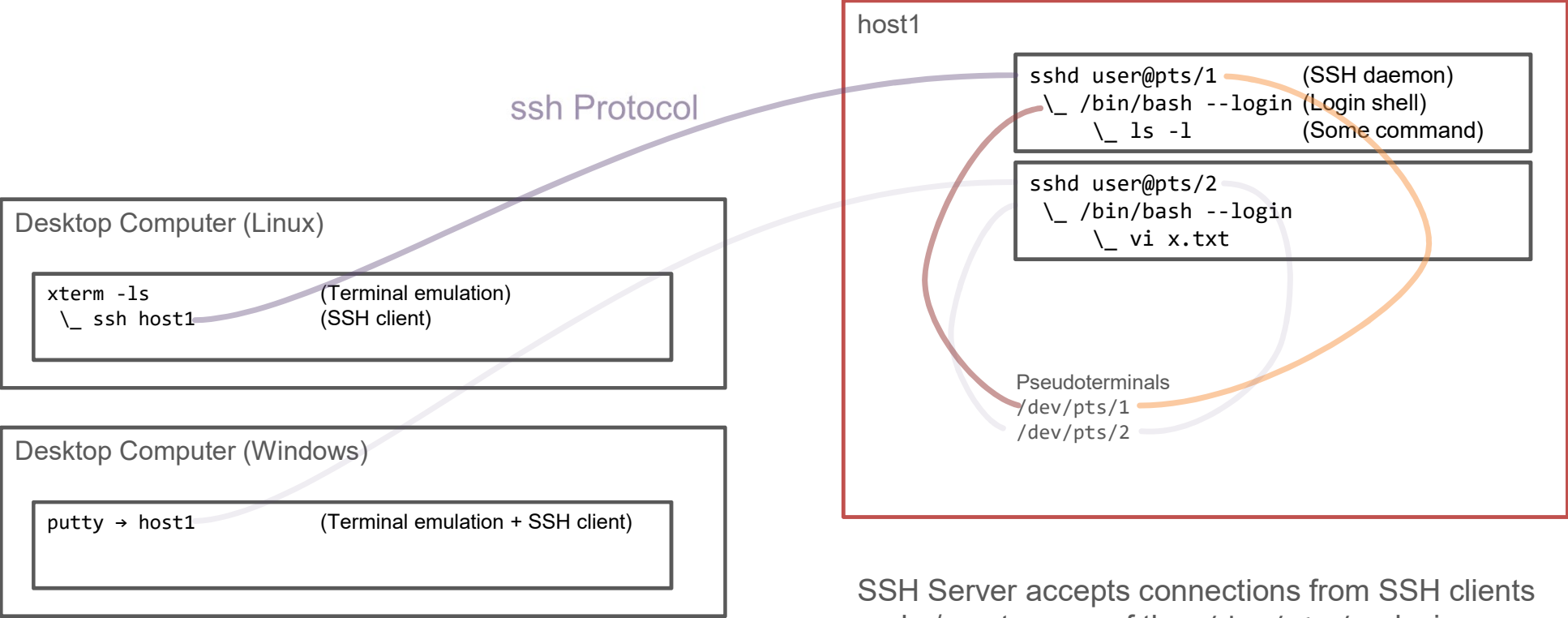
Commands

stty -a display all **tty** parameters, including **special characters**

stty change tty settings (many)

reset re-initialize terminal settings after errors (e.g. abort of editor)
sometimes you need to enter **^Jreset^J**

Understanding Remote (SSH) Terminal Sessions



SSH Server accepts connections from SSH clients
grabs/creates one of the `/dev/pts/n` devices
Starts login shell for user

- SSH Client

connects to host
sends input to host
receives output from host
- Terminal emulation

displays output in window
reads keyboard and mouse input

Getting Information About Current Session and Host

<code>hostname</code>	print system's host name to stdout
<code>uptime</code>	tell how long system has been running and load averages (1, 5, 15 minutes)
<code>uname [-a]</code>	print system information (host name, OS version, hardware name etc.)
<code>tty</code>	print to stdout the file name of the terminal (TTY) connected to stdin
<code>who am i</code>	print current session's user name, TTY, login time and origin

Using the `ps` and `top` Commands to Display Processes in System

POSIX Options

`ps` [*option* ...]

- e every process (default: processes in same session)
 - j **jobs** format (process ID, process group (=job) ID, session ID, CPU time, command)
 - f **full** format (user, process ID, parent process ID, TTY, CPU time, command)
 - l **long** format (UID, process ID, parent process ID, wchan, TTY, CPU time, command)
 - H tree format
- } combine these as needed

BSD Options

`ps` [*optionletters*] text

- a all processes (default: only yourself)
- x include processes without a TTY
- u user oriented format (user, PID, %CPU, %MEM, virtual + resident size, TTY, state, start time, CPU time, cmd)
- f forest (tree format)

top

`top` [-u *user*] [-p *pid*] display real-time view of running system, including load average and running processes.
many options and commands

Example: Identifying Left-over Processes for Killing

Situation

- Session was interrupted. Left-over processes are suspected to run on server. Or
- You have started several nohup processes - need to see what's left over

Workflow

ssh to server

<code>tty</code>	identify my own TTY to prevent suicide later
<code>ps -efH grep \$USER less</code>	list (tree) all my processes running in system, including TTY name.
	note PIDs to be killed (processes running on other or with no terminals), sparing those running on my own TTY
<code>kill PID [...]</code>	terminate PIDs in question
<code>ps -efH less</code>	check for success. If processes cannot be terminated try hard kill
<code>kill -9 PID [...]</code>	kill PIDs in question

Alternate kill command

`killall [-g pgid] [-s signal] [-u user] [name ...]` kill all processes matching criteria.

Remarks on Running Processes While You are Logged Off

Processes running after logging off or broken session

- accidental disconnect in middle of some activity
→ log on, check + kill remaining processes
- use **nohup** then log off
use on machines with no batch system
→ occasionally check for running processes
check output
kill processes not performing as expected
note: check with sysadmin if nohup is welcome
UIBK HPC systems: **do NOT run production jobs with nohup**, use **batch system** instead
- submit batch jobs (**qsub** or **sbatch**) if present
preferred method if there is a batch system
→ occasionally check for running jobs (**qstat** or **squeue** commands)
check output
cancel jobs not performing as expected (**qdel** or **scancel** commands)
- use screen(1) / VNC to protect interactive tty / X11 sessions against diconnects

Automating Work: Using the Shell as a Programming Language

Concepts

- Shell command interpreter - used interactively or for writing programs (scripts)
- Shell Script Text file with **execute permission**, containing **shell commands** and **programming constructs**
Well-written shell script can behave like executable program in every respect

Workflow: Writing and using a shell script

- `vi script` use editor to create / modify shell script
- `chmod +x script` make text file `script` executable
- `script [arg ...]` can be called like normal command if in \$PATH
- `./script [arg ...]` CWD normally not in \$PATH

Syntax

- `#!/bin/bash` In first line tell system which interpreter to use (magic (5) / “shebang”; default: /bin/sh)
- `# comment` All text after hash sign is ignored (comment)
- all features (wildcards, variables, special characters) may be used in scripts

Recommendation

- use Bourne Shell compatible shell for interactive use and programming. bash is OK, wide-spread and powerful
- definitely do NOT use “C-Shell” or derivative. Why?
- Google “**csH programming considered harmful**” (<https://www-uxsup.csx.cam.ac.uk/misc/csh.html>) - still valid)

Using and Setting Shell Arguments

Processing arguments

<code>\$0</code>	Name of script	<code>argv[0]</code>
<code>\$1, \$2, ...</code>	Positional arguments	<code>argv[1], argv[2], ...</code>
<code>\$* "\$*" "\$@"</code>	All arguments: broken into words at whitespace, as one word, preserve original arguments	
<code>\$#</code>	Number of arguments	
<code>shift [n]</code>	drop first <code>n</code> (default: 1) arguments - useful in sequential processing of arguments	

Example: `shift 2` discards values of `$1, $2`, copies values of `$3, $4 ...` to `$1, $2 ...`

Some special variables, set automatically by shell

<code>\$?</code>	Exit status of last command
<code>\$\$</code>	Process ID (PID) of current shell
<code>\$_</code>	Process ID (PID) of most recent background process

Setting arguments

`set [word ...]` current arguments (`$1, $2,...` - if any) are discarded and replaced by *words*

Using Compound Commands and Functions

Compound commands

- (*commands*) *commands* are executed in a **subshell** environment.
changes to environment or shell have no effect in the calling context
exit status is that of last command executed
- { *commands* ; } *commands* are executed in **current shell** environment.
group command can be used in many places where simple command is expected (e.g. in && ||)
{ and } must be separated by blanks, commands must be terminated with ; or newline
exit status is that of last command executed

Defining a function

name () *compound-command* defines *name* as a shell function

Calling a function

name [*argument* ...] commands in *compound-command* of function definition *name* are executed ...
... in the context of current shell (if { ... ; } was used - usual case) or
... in a subshell environment (if (...) was used)
positional parameters \$1, ... are set to **arguments** while function is executed
variables etc. in the function definition are expanded when function is executed
exit status is that of last command executed

Running Scripts in Current Shell, Initializing Sessions and Shell Scripts

Concept

Normally **shell scripts** are started in a **separate process** (new shell) → changes to variables etc. have **no effect**

If script shall be run in the **same shell**, use **source** command

Syntax

<code>. file [argument ...]</code>	portable syntax (first word is a period)
<code>source file [argument ...]</code>	read and execute commands in named <i>file</i> in current shell . arguments become \$1 \$2 ... only while <i>file</i> is executed

Initialization files

Some files are automatically **sourced** when shell starts or exits

Files in \$HOME automatically created when account is created

initialization file	scope	executed when
<code>/etc/profile</code>	system wide	begin of login shells
<code>\$HOME/.bash_profile</code>	personal	begin of login shells
<code>/etc/bashrc</code>	system wide	invoked by <code>\$HOME/.bashrc</code> (if not deleted by user)
<code>\$HOME/.bashrc</code>	personal	begin of interactive shells, also invoked by <code>\$HOME/.bash_profile</code>

Recommendation: change these files with caution

e.g. set shell options, set environment variables, add \$HOME/bin to PATH

Understanding the Exit Status of Programs

Concept

Exit status: small integer number returned by process to parent on exit

0 success, true

nonzero failure, false

Meaning of exit status depends on program. (e.g. 1 ... could not open input file, 2 ... incorrect syntax etc.)

Used in conditionals and loops

Shell syntax

`$?` special parameter: exit status of last command

`command1 && command2` run *command1*. if success, run *command2*

`command1 || command2` run *command1*. if fail, run *command2*

`command1 ; command2` run *command1*, then run *command2*

Example remove original after succesful copy:
`cp file1 file2 && rm file1`

`exit n` shell-builtin: exit this shell with status *n*

Example (precedence):
`command && echo success || echo failure`

Commands used to return exit status

`true` /bin/true exits immediately with success (0)

`false` /bin/false exits immediately with failure (1)

`test expression` exit with 0 if expression (./.) is true, else 1

Testing and Computing Expressions (portable - replaced by bash builtins)

`test expression`

`[expression]`

e.g:

- `-f name`
- `-d name`
- `-s file`
- `-r|-w|-x name`
- `-t fildes`
- `[-n] word`
- `-z word`

`word1 = word2`
`word1 -eq word2`
`\(expression \)`
`expr1 -a expr2`
`expr1 -o expr2`

bash builtin: `[[expression]]`

`expr expression`

e.g. `word1 op word2`

bash builtin: `((expression))`

return 0 if *expression* is true, 1 else

alternate form, `[` is really `/usr/bin/`

name exists and is a regular file

name exists and is a directory

file exists and has size > 0

name exists and has read / write / execute permission for current user

file descriptor *fildes* is connected to tty (0=stdin, 1=stdout, 2=stderr)

length of *word* is nonzero - warning: protect variables with " ... " (quotes)

length of *word* is zero (quotes!)

string comparison: equal (!= not equal - use whitespace)

compare numerical value (-eq -ne -gt -lt -ge -le)

group expressions - use quotes or \ to remove special meaning of parentheses

true if both are true (and)

true if any are true (or)

different syntax, more functionality

print value of *expression* to stdout

op may be one of + - * / arithmetic operations

different syntax, more functionality

Shell Programming: Conditionals

Syntax

Note for C programmers: *commands* in conditionals play same role as *logical expressions* in C
exit status 0 \approx true 1 \approx false is slightly counterintuitive

```
if cmd1
then
  commands
fi
```

```
if cmd1
then
  commands
else
  commands
fi
```

```
if cmd1
then
  commands
elif cmd2
then
  commands
else
  commands
fi
```

Semantics

Run *cmd1* - if success (exit status 0) run *commands* in *then* clause

Otherwise (if present) run *cmd2* - if success run *commands* after corresponding *then* clause ... and so on

Otherwise (if present) run *commands* after *else* clause

Shell Programming: Taking Branches on Patterns

Syntax

```
case word in
[ pattern [ | pattern ] ... )
  commands
;;
[... ]
esac
```

Semantics

word (typically variable) is expanded and compared against *patterns*
at first match, *commands* after matching pattern until `;;` are executed
execution continues after `esac`
use `*` pattern as a match-all (default)

Example:

```
case "$a" in
cow|dog|frog) echo "animal" ;;
daisy|violet) echo "flower" ;;
b..)         echo "a three letter word starting with b" ;;
*)           echo "unknown species" ;;
esac
```

Shell Programming: While Loops

Syntax

```
while cmd
do
    commands
done
```

Semantics

Run *cmd* - if success (exit status 0) run *commands* after *do* clause

Repeat until *cmd* exits with nonzero status

break [*n*] exits from innermost loop (or from *n* levels)

Example: watch files appearing and growing while other programs create and write to them

```
while true
do
    clear
    date
    ls -l *.out
    sleep 2
done
```

Better version:

```
watch 'ls -l *.out'
```

Q: why the quotes?

Example: Option Processing

Example:

Sequential option processing for a script with usage

script [-d] [-f file] [arg ...]

caution: need to add error checking

```
debug=0
file=script.in } set defaults

while test $# -gt 0
do
    case "$1" in
    -d)  debug=1 ; shift ;;
    -f)  file="$2" ; shift 2 ;;
    -*)  echo >&2 "$0: error: $1: invalid option" ; exit 2 ;;
    *)   break ;;      # remaining arguments are non-option
    esac
done

for arg
do
    ... } process operands
done
```

Shell Programming: For Loops

Syntax

```
for variable [in word [...]]  
do  
    commands  
done
```

Semantics

Set shell *variable* to successive values in list of *words* (default "\$@" - useful in scripts) and execute *commands* for each value of *variable*

break [*n*] exits from innermost loop (or from *n* levels)

Example

```
for i in $(ls *.c)  
do  
    cp $i $i.backup  
done
```

Putting it Together - Examples - Renaming Many Files

Task

rename many files from xxx.for to xxx.f

```
for i in *.for
do
    mv -i $i $(basename $i .for).f
done
```

Demo

touch a.for b.for	create example files
set -x	set shell option: display commands before they are executed (good for debugging and analyzing)

running above loop yields

```
+ for i in '*.for'
++ basename a.for .for
+ mv -i a.for a.f
+ for i in '*.for'
++ basename b.for .for
+ mv -i b.for b.f
```

note: this example fails if file names contain blanks

Examples - Displaying Arguments

Writing a simple shell argument checker

```
$HOME/bin/pargs
```

```
#!/bin/bash
for i
do
    echo ">>$i<<"
done
```

Usage examples: file name containing blank

```
$ echo "$IFS" | od -bc
00000000 040 011 012 012
          \t  \n  \n

$ ls -l
-rw-rw-r-- 1 c102mf c102mf  0 Sep 27 11:13 a b
-rw-rw-r-- 1 c102mf c102mf 82 Sep 27 14:58 c
```

```
$ pargs *
>>a b<<
>>c<<
$ pargs $(ls)
>>a<<
>>b<<
>>c<<
```

```
$ oldifs="$IFS"; IFS='^J\'
$ pargs $(ls)
>>a b<<
>>c<<
$ IFS="oldifs"
```


Shell Programming: Opening and Reading Files

Shell-builtins

read [-u *fd*] [*name* ...] read one line from stdin, split into words (IFS) and put each word in variable \$*name*.
remaining data goes to last variable. returns 0 (true) unless end of file.

 -u *fd* read from file descriptor *fd* instead of 0 = *stdin*

exec [*command* [*args* ...]] [*redirections*] if *command* is given, it replaces current shell.
 otherwise, *redirections* are performed for current shell.

use **exec** with no command to open descriptors in **current shell**

Example

fragment from shell script

<code>infile=myfile.in</code>	in real application, would take file name e.g. from command line
<code>exec 3<\$infile</code>	open \$infile in current shell's file descriptor 3 for reading
<code>while read -u 3 line</code>	each iteration reads one line from file fd=3 and puts entire line into variable named "line"
<code>do</code>	
<code>pargs \$line</code>	\$line not quoted: split into words here. (do useful stuff instead of calling pargs)
<code>done</code>	
<code>echo "finished"</code>	loop ends when all lines have been read (read returns non-zero exit status)

Examples - Grep Command to Include Header Lines

Defining new grep-like command that always displays first line of input (head grep)

`$HOME/bin/hgrep`

```
#!/bin/bash
# usage: hgrep PATTERN [FILE ...]
test $# -ge 1 || { echo >&2 "$0: error: no pattern given" ; exit 2 ; }
pattern="$1"; shift
sed -s -n -e '1{p;d}; /'"$pattern"'/p' "$@"
```

Usage example:

```
09:26:57 c102mf@login.leo3e:~ $ ps -efH | hgrep $USER
UID      PID  PPID  C  STIME TTY      TIME CMD
root      23002  2055  0  08:45 ?        00:00:00 sshd: c102mf [priv]
c102mf    23005  23002  0  08:45 ?        00:00:00 sshd: c102mf@pts/4
c102mf    23006  23005  0  08:45 pts/4    00:00:00 -bash
c102mf    32350  23006  0  09:27 pts/4    00:00:00 ps -efH
c102mf    32351  23006  0  09:27 pts/4    00:00:00 /bin/bash /home/c102/c102mf/bin/hgrep c102mf
c102mf    32352  32351  0  09:27 pts/4    00:00:00 sed -n -e 1{p;d}; /c102mf/p
c102mf    26023      1  0 Sep19 ?        00:00:00 SCREEN -D -R
c102mf    26024  26023  0 Sep19 pts/17   00:00:00 /bin/bash
```

Exercise:

make this more general: add “-n lines” option

Examples - Listing all Includes in a Programming Project

Goal: create a complete list of included files in a source file hierarchy
demonstration of a slightly non-trivial sed replacement construct

Steps: (use source of some arbitrary sourceforge/gitlab project)

```
$ wget https://gitlab.com/procps-ng/procps/repository/master/archive.tar.gz -O procps-ng.tar.gz
$ tar xf procps-ng.tar.gz ; mv procps-master-* procps-ng
$ cd procps-ng
```

first sighting

```
$ grep -r '#include' . | less
```

lines of interest look like `#include <getopt.h>` or `#include "libiberty.h"`, sometimes with leading and trailing stuff

complete construct

from output, get rid of everything outside `<...>` and `"..."`, then sort trimmed output and remove duplicate lines
`grep -h` would get rid of file names output, do not need because we eliminate this with other leading text

```
$ grep -r '#include' . | sed 's/^\.*\([<"]\([^>"]*\>"]\)\).*$/\1/' | sort | uniq | less
```

grouping construct `\(... \)` in search expression permits back-reference `\1` in replacement

within, we first look for opening quote (character class `[<"]`),

then all characters except closing quote `[^>"]*`, then closing quote `[^>"]`

outside parentheses, we use match-all `.*` and anchor this to the beginning (^) respectively end (\$) of line

in the replacement, only the found text between parentheses is inserted

Examples - Processing Options and Arguments

Example: famous / silly pingpong program as script

```
#!/bin/bash
```

```
# pingpong: count replacing multiples of 3 with ping, 5 with pong
```

```
usage () {  
    cat <<-STOP >&2  
        usage: $0 [-n pingmod] [-m pongmod] [-d] [number ...]  
        defaults: pingmod = 3, pongmod = 5  
        -d print some debugging output  
STOP  
}
```

```
debug=no  
pingmod=3  
pongmod=5
```

```
while test $# -gt 0  
do
```

```
    case "$1" in  
        -n) pingmod="$2"; shift 2 ;;  
        -m) pongmod="$2"; shift 2 ;;  
        -d) debug=yes; shift ;;  
        -*) echo >&2 "$0: unknown option $1"; usage ; exit 2 ;;  
        *) break ;;          # no more options  
    esac
```

```
done
```

```
if test "$debug" = yes  
then
```

```
    cat <<-STOP  
        parameters in effect:  
        -n $pingmod -m $pongmod debug: $debug numbers: $@
```

```
STOP  
fi
```

```
for number  
do
```

```
    echo "=====  
    for i in $(seq $number)  
    do
```

```
        unset pp  
        test $(expr $i % $pingmod) = 0 && { echo -n ping ; pp=1 ; }  
        test $(expr $i % $pongmod) = 0 && { echo -n pong ; pp=1 ; }  
        test -z "$pp" && echo -n $i  
        echo
```

```
    done  
done  
echo "=====“
```

Examples - Parameter Study (shell loop) - no error checking

Parameter study: run program “mean” with command line taken from *n*-th line of file containing parameters

Idea: later, we get value of *n* from batch system (job array)

Example program: `mean -t {a|g|h} -f file` -t type (arithmetic, geometric, harmonic) -f file containing numbers

Input files: `ari.txt` `geo.txt` `hrm.txt` each containing test data with “simple” arithmetic, geometric, and harmonic means

Parameter file `params.in`

```
-t a -f ari.txt
-t g -f ari.txt
-t h -f ari.txt
-t a -f geo.txt
[...]
```

`ari.txt`

```
11 12 13 14
```

`geo.txt`

```
1 10 100 1000 10000
```

`hrm.txt`

```
1 .5 .3333333333333333 .25 .2
```

Driver script `runall.sh`

```
#!/bin/bash
file="$1"

for i in $(seq $(wc -l < "$file" ))
do
    ./mean $(sed -n "$i p" "$file")
done
```

doubly nested command substitution
inner counts input lines
outer makes list of line numbers

extracts *i*-th line of file
and substitutes it as
command arguments

Usage

```
$ ./runall.sh params.in
ari.txt: 4 a      12.500000
ari.txt: 4 g      12.449770
ari.txt: 4 h      12.399484
geo.txt: 5 a     2222.200000
geo.txt: 5 g     100.000000
geo.txt: 5 h       4.500045
hrm.txt: 5 a       0.456667
hrm.txt: 5 g       0.383852
hrm.txt: 5 h       0.333333
```

Examples - Parameter Study (GNU parallel) - no error checking

GNU parallel:

build and execute shell command lines from stdin in parallel on same host - similar to xargs

Execution script `runone.sh`

```
#!/bin/bash
```

```
file="$1"; line="$2"  
length=$(wc -l < "$file" )
```

```
./mean $(sed -n "$line p" "$file")
```

Driver script `runparallel.sh`

```
#!/bin/bash
```

```
file="$1"
```

```
seq $(wc -l < "$file" ) |  
parallel -k ./runone.sh "$file"
```

Usage

```
$ ./runparallel.sh params.in
```

```
ari.txt: 4 a      12.500000  
ari.txt: 4 g      12.449770  
ari.txt: 4 h      12.399484  
geo.txt: 5 a    2222.200000  
geo.txt: 5 g     100.000000  
geo.txt: 5 h       4.500045  
hrm.txt: 5 a       0.456667  
hrm.txt: 5 g       0.383852  
hrm.txt: 5 h       0.333333
```

Examples - Parameter Study (batch system) - no error checking

Job array:

run multiple instances across hosts of identical job with unique value of `$SGE_TASK_ID` between 1 and *n*

Job script `runonejob.sge`

```
#!/bin/bash
#$ -q short.q
#$ -N onejob
#$ -cwd

file="$1"; line="$SGE_TASK_ID"

length=$(wc -l < "$file" )

./mean $(sed -n "$line p" "$file") > oneout.$line
```

Usage

```
$ ./submitparallel.sh params.in
output will go to individual files
```

collect output in correct order after jobs have run

```
for i in $(seq $(wc -l < params.in ))
do
    cat oneout.$i
done
```

Driver script `submitparallel.sh`

```
#!/bin/bash

file="$1"

length=$(wc -l < "$file" )
qsub -t 1-$length ./runone.sh "$file"
```

Examples - Parameter Study (shell loop)

Parameter study: run program “mean” with command line taken from n -th line of file containing parameters

Idea: later, we get value of n from batch system (job array)

Example program: `mean -t {a|g|h} -f file` -t type (arithmetic, geometric, harmonic) -f file containing numbers

Input files: `ari.txt` `geo.txt` `hrm.txt` each containing test data with “simple” arithmetic, geometric, and harmonic means

Parameter file `params.in`

```
-t a -f ari.txt
-t g -f ari.txt
-t h -f ari.txt
-t a -f geo.txt
[...]
```

`ari.txt`

```
11 12 13 14
```

`geo.txt`

```
1 10 100 1000 10000
```

`hrm.txt`

```
1 .5 .3333333333333333 .25 .2
```

Driver script `runall.sh`

```
#!/bin/bash
usage () { echo >&2 "usage: $0 PARAMETERFILE"; exit 2 ; }
# set -x
file="$1"
test -n "$file" || usage
test -f "$file" -a -r "$file" ||
{ echo >&2 "$0: cannot read file $file" ; exit 1 ; }
for i in $(seq $(wc -l < "$file" ))
do
    ./mean $(sed -n "$i p" "$file")
done
```

Usage

```
$ ./runall.sh params.in
ari.txt: 4 a      12.500000
ari.txt: 4 g      12.449770
ari.txt: 4 h      12.399484
geo.txt: 5 a     2222.200000
geo.txt: 5 g     100.000000
geo.txt: 5 h       4.500045
hrm.txt: 5 a       0.456667
hrm.txt: 5 g       0.383852
hrm.txt: 5 h       0.333333
```


Examples - Parameter Study (GNU parallel)

GNU parallel:

build and execute shell command lines from stdin in parallel on same host - similar to xargs

Execution script `runone.sh`

```
#!/bin/bash
usage () { echo >&2 "usage: $0 PARAMETERFILE LINE"; exit 2 ; }
test $# = 2 || usage
file="$1"; line="$2"
test -n "$file" || usage
test -f "$file" -a -r "$file" ||
{ echo >&2 "$0: cannot read file $file" ; exit 1 ; }
length=$(wc -l < "$file" )
test "$line" -gt 0 -a "$line" -le $length ||
{ echo >&2 "$0: line $line not in (1 .. length $file = $length)" ; exit 1 ; }
}
./mean $(sed -n "$line p" "$file")
```

Driver script `runparallel.sh`

```
#!/bin/bash
usage () { echo >&2 "usage: $0 PARAMETERFILE "; exit 2 ; }
test $# = 1 || usage
file="$1"
test -n "$file" || usage
test -f "$file" -a -r "$file" ||
{ echo >&2 "$0: cannot read file $file" ; exit 1 ; }
seq $(wc -l < "$file" ) |
parallel -k ./runone.sh "$file"
```

Usage

```
$ ./runparallel.sh params.in
ari.txt: 4 a      12.500000
ari.txt: 4 g      12.449770
ari.txt: 4 h      12.399484
geo.txt: 5 a    2222.200000
geo.txt: 5 g     100.000000
geo.txt: 5 h       4.500045
hrm.txt: 5 a       0.456667
hrm.txt: 5 g       0.383852
hrm.txt: 5 h       0.333333
```

Examples - Parameter Study (batch system)

Job array:

run multiple instances across hosts of identical job with unique value of `$SGE_TASK_ID` between 1 and *n*

Job script `runonejob.sge`

```
#!/bin/bash
#$ -q std.q
#$ -N onejob
#$ -l h_rt=10
#$ -cwd
usage () { echo >&2 "usage: $0 PARAMETERFILE"; exit 2 ; }
test $# = 1 || usage
file="$1"; line="$SGE_TASK_ID"
test -n "$file" || usage
test -f "$file" -a -r "$file" ||
{ echo >&2 "$0: cannot read file $file" ; exit 1 ; }
length=$(wc -l < "$file" )
test "$line" -gt 0 -a "$line" -le $length ||
{ echo >&2 "$0: line $line not in (1 .. length $file = $length)" ; exit 1 ; }

./mean $(sed -n "$line p" "$file") > oneout.$line
```

Driver script `submitparallel.sh`

```
#!/bin/bash
usage () { echo >&2 "usage: $0 PARAMETERFILE "; exit 2 ; }
test $# = 1 || usage
file="$1"
test -n "$file" || usage
test -f "$file" -a -r "$file" ||
{ echo >&2 "$0: cannot read file $file" ; exit 1 ; }

length=$(wc -l < "$file" )
qsub -t 1-$length ./runonejob.sge "$file"
```

Usage

\$ `./submitparallel.sh params.in`
output will go to individual files

collect output after jobs have run

```
for i in $(seq $(wc -l < "$file" ))
do
    cat oneout.$i
done
```

Understanding Users and Groups

Concept: UNIX is a multiuser system

User = entity to identify multiple **persons** using a computer as well as special **system accounts**

Group = logical collection of users

every user is a member of one **default group** and may be member of **additional groups**

Users and groups are used to

- identify and validate persons trying to access system (**login** procedure)
- manage **permissions** for
 - files and directories
 - control of processes

Properties of user

User name (login name: lower case alphanumeric), **password**, numerical **UID**, **default group**, **login shell**, **home directory**

Properties of group

Group name, numerical **GID**, list of **members** (beyond default members)

Process attributes: **effective UID and GID** (used for access checking), real UID and GID (original IDs)

Superuser (root) has UID = 0 special privileges

Using Commands to Identify Users

Commands

<code>whoami</code>	<code>logname</code>	print effective user ID, login name
<code>groups</code>	<code>[user ...]</code>	print list of group memberships of named (default: current) user
<code>id</code>	<code>[option ...] [user ...]</code> <code>-u -g -G -r</code>	print user and group information on named (default: current) user print only: user, group, list of memberships, real instead of effective IDs
<code>who</code>	<code>[option ...]</code> <code>-a</code> <code>-m</code>	list information on users who are currently logged in more information only user using this command ("who am i")
<code>finger</code>	<code>[option ...] [user ...]</code> <code>-l</code>	display information on users (default: all currently logged in) long format (default if <code>user</code> is given)

Understanding File Access Permissions

\$ **ls -l**

[...]

-rw-r-----.	1	c102mf	c102	116	May 18 12:56	mynotes.txt
-rw-r--r--.	1	c102mf	c102	3157	May 18 19:23	public.txt
-rwxr-x---	1	c102mf	c102	917	Sep 18 13:00	testscript
drwxr-xr-x.	2	c102mf	c102	4096	May 18 12:55	utilities

special access mode (. SELINUX + ACL)

number of links (directories: number of subdirs + 2)

permissions for user
group
others

access permissions for

file

directory

file type

owner

group

size (bytes)

modification date

name

r read
w write
x execute

s suid/sgid
t sticky bit

read file contents
write to file
run as program

sgid: new files inherit GID
only owner may delete files

obsolete

Setting File Access Permissions

`chmod` [*option* ...] *mode* [,*mode*...] *file* [...] set or clear file access permission bits

-R recursive

-v -c verbose, report changes only

mode is one or more of the following (comma separated)

[*u**g**o**a*...][+|=][*perms*] *u* user who owns the file
 g users in the file's group
 o other users not in the file's group
 a (default) all users, do not affect bits set in umask
 + add permissions
 - remove permissions
 = set or copy permissions
or *numerical mode* (. / .)

perms is zero or more of the following

r read
w write
x execute (directory: search)
X execute (set only if execute set for some user)
s set user ID or group ID on execution (directory: inherit group)
t sticky bit (directories: restrict deletion to file owner. e.g. /tmp)
[*u**g**o*] copy permission from user, group or others

Understanding Access Bits, Using umask

Numerical *mode*

1-4 octal digits: (*special*) (*user*) (*group*) (*others*)

value = 0-7 by adding values 1, 2 and 4)

omitted digits = leading zeros

Values

special

4 set user ID on execution

2 set group ID on execution (files) inherit group ID (directories)

1 sticky bit

user group others

4 read

2 write

1 execute

Examples

`chmod u=rwx,g=rx,o= file` same as `chmod 0750 file`

`chmod u=rx,g=r,o= file` same as `chmod 0640 file`

Umask and umask command (shell-builtin)

The *umask* (user file creation mask) is a property of every process

When a new file is created, bits *set in umask* are *cleared in file's permissions*

`umask [-S] [-p] [mode]` set or report umask (-S symbolic form, -p print umask in command format)

Examples

`umask → 0027` same as `umask -S → u=rwx,g=rx,o=`

`umask 0077` set strict umask (new files have no permissions for group and others)

Understanding and Setting File Times

File times (time + date)

time attribute	meaning	displaying	setting
modification time	when file contents last changed	<code>ls -l <i>file</i></code>	<code>touch -m -d <i>string file</i></code>
access time	last file access (e.g. read)	<code>ls -lu <i>file</i></code>	<code>touch -a -d <i>string file</i></code>
change time	last modification of attributes	<code>ls -lc <i>file</i></code>	

Changing a file's attributes (times or access permission)

will always set its change time (AKA inode modification time) to current time

Understanding Links and Symbolic Links

Link count, inode

every file has a unique index number (inode number) in file system

any file may have one or more directory entries (hard link = pair `name inode`)

create additional hard links with

```
ln existingname newname
```

every directory has at least two directory entries: `named entry in parent` and `.` in current directory
plus one `..` in each subdirectory

Symbolic link (AKA symlink or soft link)

named reference to other file or directory with relative or absolute path

create with

```
ln -s target newame
```

displayed in `ls -l` output as `l name -> target`

Example:

```
$ ln -s /scratch/c102mf Scratch
```

```
$ ls -ld Scratch
```

```
lrwxrwxrwx. 1 c102mf c102 15 Sep 28 14:42 Scratch -> /scratch/c102mf
```

```
$ ls -Lld Scratch
```

```
drwx--x---. 51 c102mf c102 32768 Oct 3 18:50 Scratch
```

Using the `ls` command

`ls` [*option* ...] [*name* ...] list information about named *files* or *directories* (default: current working directory)

- a all entries (do not ignore entries starting with `.`)
- d list directories themselves, not their contents
- l long listing (default: compact multicolumn listing)
- L resolve symbolic link, list target instead of link
- F compact listing, mark directories with `/` and executables with `*`
- i print inode number
- s print effectively occupied space (*) on disk (blocks; use `-k` to force kB)
- h print size human-readable
- R recursively list named *directories*
- 1 single column even when output goes to tty

- t sort by modification time, newest first
- u sort by (and show if -l) access time
- c sort by (and show if -l) inode modification time
- r reverse sort order

(*) may differ from logical size:

includes disk addressing metadata for large files; files may have zero-filled holes occupying no space

Getting More Information About Files

file [*option* ...] [*name* ...] try to classify each file and print results to stdout

- b brief. no filenames in output
- i output mime type
- p preserve access date
- z try look inside compressed files
- (many)

For each file name given in command line, file heuristically guesses the file type from its permissions and contents (see magic(5))

Output format

name: description

Example

```
file *
a.out:      ELF 64-bit LSB executable [...] not stripped
bin:        directory
myscript:   Bourne-Again shell script, ASCII text executable
README:     ASCII text
```

Using find to Search for Files in Directory Hierarchy (1)

find [*path* ...] [*expression*] recursively traverses named *directories* (default: current) and evaluate expression for each entry found (default: -print)

expression is made up of

- options** (affect overall operation, always true)
- tests** (return true or false)
- actions** (have side effects, return true or false)
- operators** (default: **and**)

options

- depth process directory contents before directory
- xdev do not cross mount points (same as -mount)

tests numeric arguments may be *n* (exactly) *+n* (greater than) *-n* (less than)

- name *pattern* name matches *pattern* (wildcards - use quotes)
- iname *pattern* name matches *pattern* - ignore case
- type **c** **d** directory **f** regular file **l** symbolic link
- mtime *n* -mmin *n* modified *n* days or minutes ago
- atime *n* -amin *n* accessed *n* days or minutes ago
- newer *file* found item was modified more recently than named *file*
- size *n*[kMG] size is *n* bytes, kilo- mega- gigabytes (binary)

actions

(./.)

Using find (2)

actions

<code>-print</code>	(default) print path name of found object to stdout
<code>-print0</code>	print path name of found object, terminated by NULL character instead of newline
<code>-ls</code>	print <code>ls -dils</code> output for each found object
<code>-exec command ;</code>	for each found object execute shell command. { } inserts path name (use quotes), ; terminates command (quotes)
<code>-execdir command ;</code>	for each found object, cd into directory containing object and execute shell command.
<code>-delete</code>	delete file (dangerous), true if success
<code>-prune</code>	if file is directory, do not descend into it (incompatible with <code>-depth</code>)

operators

<code>(expr)</code>	grouping of expressions (use quotes)
<code>! expr</code>	logical negation
<code>expr1 [-a] expr2</code>	logical and : <code>expr2</code> is not evaluated if <code>expr1</code> is false
<code>expr1 -o expr2</code>	logical or : <code>expr2</code> is not evaluated if <code>expr1</code> is true
<code>expr1 , expr2</code>	list: <code>expr1</code> and <code>expr2</code> are always evaluated, return truth value of <code>expr2</code>

Using xargs and parallel to Process Large Number of Objects

Concept

Command lines are limited in number of arguments and total length.

Use **xargs** to split list of arguments into suitable portions and call command for each portion
useful e.g. for commands that have no *recursive* option

GNU **parallel** is similar to **xargs** but allows parallel execution on same and remote hosts

xargs [*option* ...] *command* [*initial-arguments*] build and execute command lines from standard input

- a *file* read arguments from *file* instead of *stdin*
- d *delim* use *delim* to separate arguments instead of whitespace; e.g. -d '\n'
- n *max-args* use at most *max-args* arguments per command line
- S *max-size* use at most *max-size* characters per command line
- t verbose mode
- 0 input items terminated by NULL character instead of *delim* (corresponds to -print0 of find)

parallel [*option* ...] *command* [*initial-arguments*] build and execute command lines
from standard input in parallel (many options)

Example

set all files in directory *my-project* to fixed modification time

```
touch -t 201709100000 ref-file      then
```

```
find my-project -type f -print0 | xargs -0 touch -r ref-file
```

Compressing and Uncompressing Files (gzip, bzip2)

<code>{gzip bzip2} [option ...] [file ...]</code>	compress files using { Lempel-Ziv coding block-sorting compressor }, replace originals by compressed files appending { .gz .bz2 } to file name
<code>{gunzip bunzip2} [option ...] [file ...]</code>	uncompress files, replace compressed files by originals
<code>{zcat bzcat} [option ...] [file ...]</code>	uncompress files to stdout
<code>-c</code>	compress / uncompress to stdout
<code>-1 ... -9</code>	fast ... best (default: 6 - ignored by bzip2)
other options, some specific to gzip / bzip2 family	

Note

multiple concatenated compressed files can be correctly uncompressed

Example

```
gzip -c file1 > foo.gz
gzip -c file2 >> foo.gz
gunzip -c foo
```

is equivalent to

```
cat file1 file2
```

Packing and Unpacking Collections of Files (tar)

Concept

Pack entire directory hierarchy into single archive file, suitable for archive and distribution purposes

Most software packages are distributed this way

```
tar {c|x|t}...f... archive.tar[.suffix] [name ...]
```

c	Create. Pack named <i>files</i> and (recursively) <i>directories</i> into named <i>archive</i> file or write to <i>stdout</i>
t	Table of contents of named <i>archive</i> file or from <i>stdin</i>
x	eXtract data from named <i>archive</i> file or from <i>stdin</i> , recreating files and directories
v	verbose mode
z	use gzip compression, <i>suffix</i> should be <i>.gz</i>
j	use bzip2 compression, <i>suffix</i> should be <i>.bz2</i>
p	preserve permissions when extracting

Note

Tar is traditionally used with BSD-style single letter options.

GNU tar also supports standard POSIX (-x value) and GNU (--option=value) options; using BSD style helps portability

Examples

```
tar cvfz my_project.tar.gz my_project    recursively pack directory my_project into my_project.tar.gz
tar tvfz my_project.tar.gz                lists table of contents
tar xvfz my_project.tar.gz                unpacks data, recreating the directory my_project
```

Note: for *t* and *x*, the compression option (z or j) may be omitted when using GNU tar

Understanding UNIX File Systems

Concepts

File system = collection of files and directories on one disk, partition, disk array or file server

All file systems organized in **one tree**

Mount point = directory at which another file system starts

Root directory / is starting point for all absolute paths

Facts

Each file system has its own **capacity** (total file size, number of files) and **quota** (if defined)

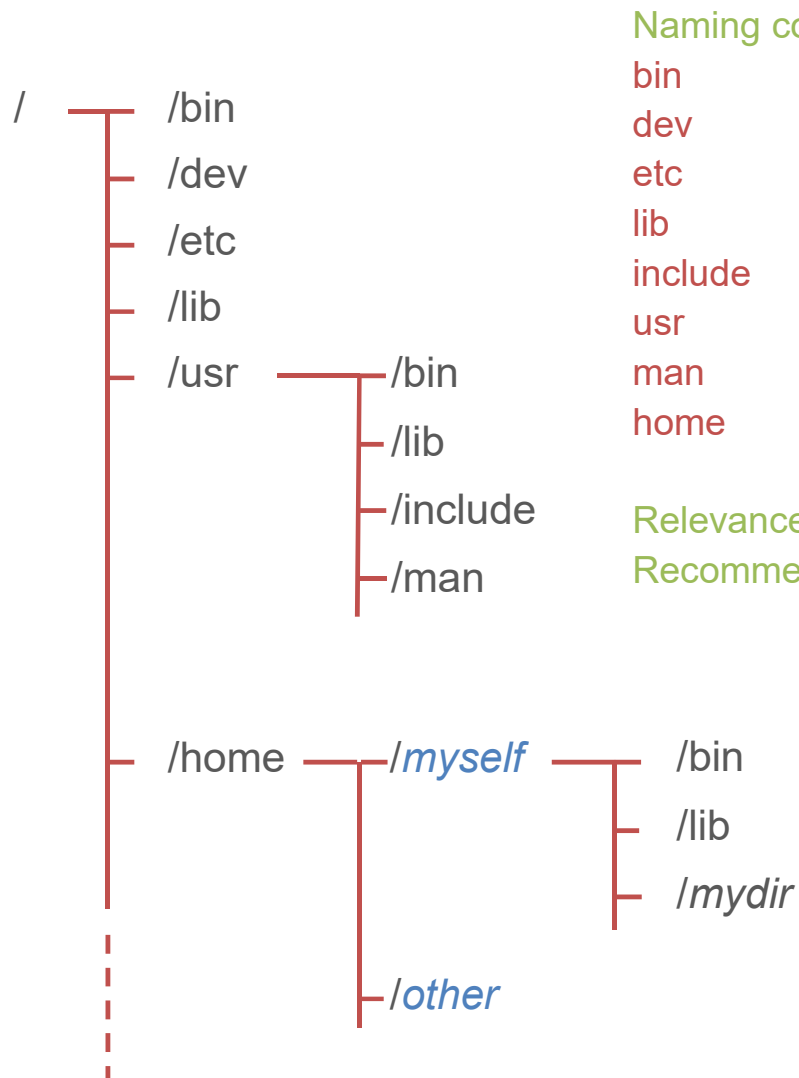
Hard links work only within file system: use symlinks to link across file systems

Moving a file to another file system involves copying all data (move within file system: will create new hard link)

Data loss typically affects an entire file system: backup your files

UNIX file system organization follows certain conventions `./.`

Understanding the UNIX File System Hierarchy



Naming conventions

bin	executable programs (utilities)
dev	device files - interface to system hardware
etc	various configuration files
lib	libraries - collection of object files (components of executables)
include	header files - collection of interface definitions (#include <xxxx.h>)
usr	secondary hierarchy - files and utilities used in multiuser operation
man	starting point of documentation (often share/man)
home	parent of all users' home directories

Relevance: 3rd party and self written software should follow same conventions

Recommended: add \$HOME/bin to your \$PATH variable

Using Special Device Files

Some special device files are useful with shell scripts and programs

<code>/dev/null</code>	reading from <code>/dev/null</code> gives immediate EOF data written to <code>/dev/null</code> is discarded
<code>/dev/zero</code>	reading from <code>/dev/zero</code> returns bytes containing “zero” characters
<code>/dev/tty</code>	process’s controlling terminal. reading and writing from/to <code>/dev/tty</code> will read / write to terminal independent of current stdin or stdout
<code>/dev/random</code>	reading returns random bytes, waiting for entropy pool to supply enough bits (slow!)
<code>/dev/urandom</code>	random bytes, using pseudorandom generator if enough entropy is not available

Understanding Text File Structure + Conversions

Concepts

- File** sequence of bytes
 - Text file** sequence of lines, each terminated by **Newline** (**Line Feed**) character (**NL** , **LF** , **^J** , **\n**)
 - Line** sequence of characters consisting of single bytes ASCII, Latin1 or varying numbers of bytes (UTF-8)
- Character set determined by **environment variable \$LANG** (e.g. **C** (ASCII) **en-US** (Latin1) **en_US.UTF-8** (UTF))
- Note for C programmers: same structure as expected by C programs

Windows: differences to UNIX

- In Windows text files, lines are separated by **Carriage Return + Newline sequence** (**CR NL** , **^M^J** , **\r\n**)
- C programs must open text files in text mode to effect conversion
- Character set encoding determined by **invisible bytes** at beginning of file
- Various nonstandard encodings (code pages) used

Analyzing file contents

- od** [**option** ...] [**file** ...] dump file contents in octal and other formats, output to stdout.
Useful options: **-t o1** **-t x1** **-t c** (octal, hexadecimal, character bytes)

Converting file formats

- [**dos2unix** | **unix2dos**] [**option** ...] [**file** ...] [**-n** **infile outfile**] convert file formats & windows encodings
- iconv** [**option** ...] [**-f** **from-encoding**] [**-t** **to-encoding**] [**file** ...] convert standard encodings

Understanding the X Window System (X11)

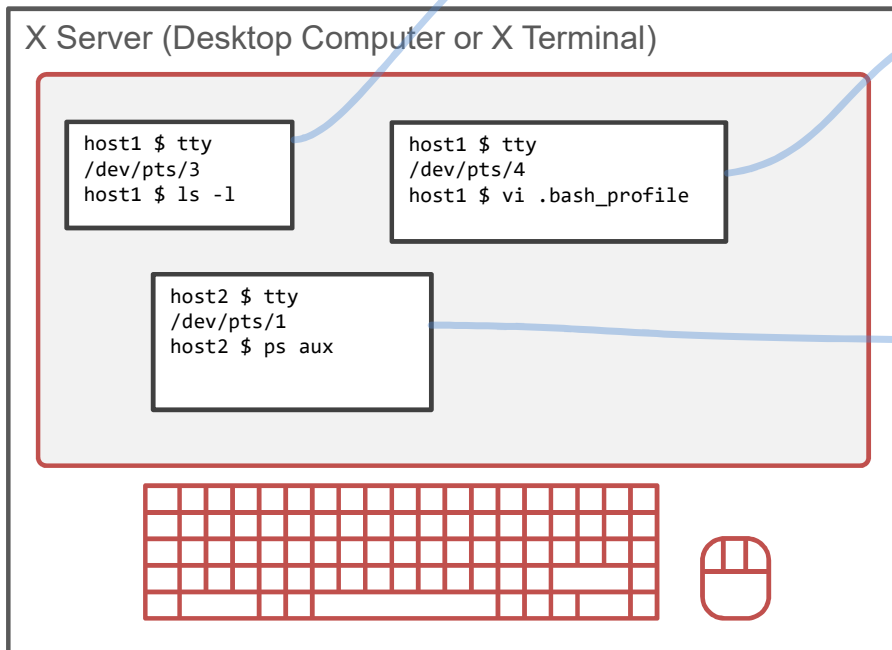
X11 is a **Client-Server** system

Roles of Client and Server counter-intuitive at first

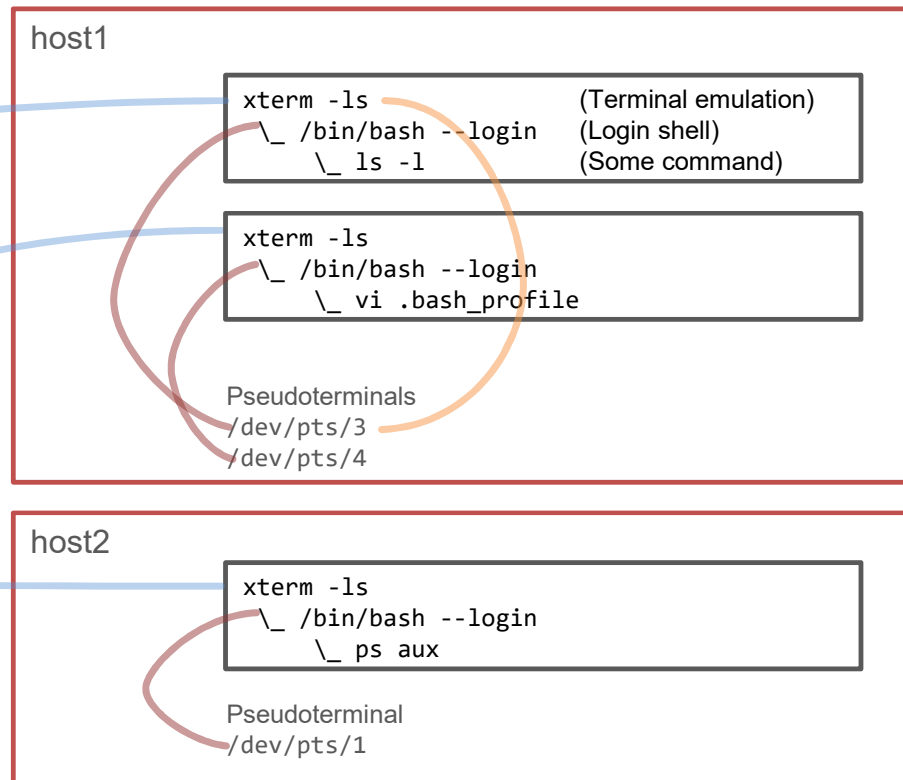
DISPLAY=server:0

port = 6000 + **display-ID**

X11 Protocol



X Server controls **screen**, **keyboard**, **mouse**
accepts connections from X Clients (e.g. xterm)
sends events (**keyboard**, **mouse**) to client that has **focus**



X Client (e.g. xterm) runs on local or remote host
connects to **X Server** named in **\$DISPLAY** variable
uses X server to open window and display text and graphics
does useful work (e.g. editor or terminal emulation)
may start other programs (xterm: default: login shell)

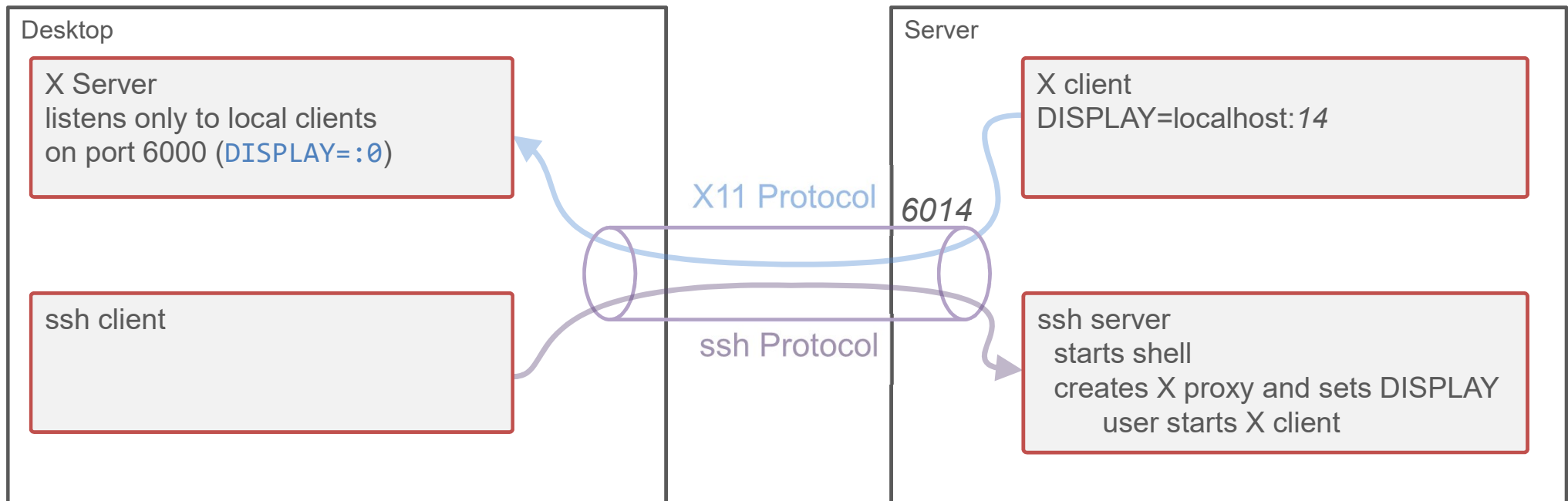
Using the X11 Tunnel

Security: X Server should only accept local connections

Q: how to connect remote clients?

A: X11 tunneling through SSH: client connects to server's localhost
ssh forwards connection to desktop
X server accepts local connection

Usage: `ssh -X server` enables X11 forwarding, ssh server automatically sets `$DISPLAY`
add line `ForwardX11 yes` to `$HOME/.ssh/config` to enable by default



Using Xterm

<code>xterm</code>	[<i>option</i> ...] &	start the xterm terminal emulator in the background (as typically with all X clients)
<code>-ls</code>		run the shell as a login shell
<code>-e</code>	<i>program</i> [<i>args</i> ...]	run program instead of shell. Must be last argument
<code>-display</code>	<i>display</i>	use named display instead of \$DISPLAY
<code>-geometry</code>	<i>WIDTH</i> x <i>HEIGHT</i> + <i>XOFF</i> + <i>YOFF</i>	width and height in characters, offsets in pixels from left and top edge of screen. negative offsets are from right and bottom
<code>-title</code>	<i>title</i>	window title string

Recommended resource definitions in `$HOME/.Xresources` or `$HOME/.Xdefaults` (*)

<code>XTerm*selectToClipboard:</code>	<code>True</code>	allows cut/paste integration with newer X clients and non-X programs
<code>XTerm*faceName:</code>	<code>Mono</code>	use scalable fonts, recommended for high resolution displays
<code>XTerm*faceSize:</code>	<code>8</code>	change value for convenience
<code>XTerm*saveLines:</code>	<code>10000</code>	size of scrollback buffer
<code>XTerm*scrollBar:</code>	<code>True</code>	display scroll bar

(*) `.Xresources` is automatically loaded into X server when an X display manager session is started.

load manually with `xrdb -load $HOME/.Xdefaults`

`.Xdefaults` supplies these settings on the client side when no resources have been loaded.

Use this when not using X display manager (e.g. X server on non-UNIX workstation)

Using Xterm Mouse Actions

xterm uses MIT Athena widgets - non-intuitive but powerful

size of scroll bar shows visible fraction of total buffer

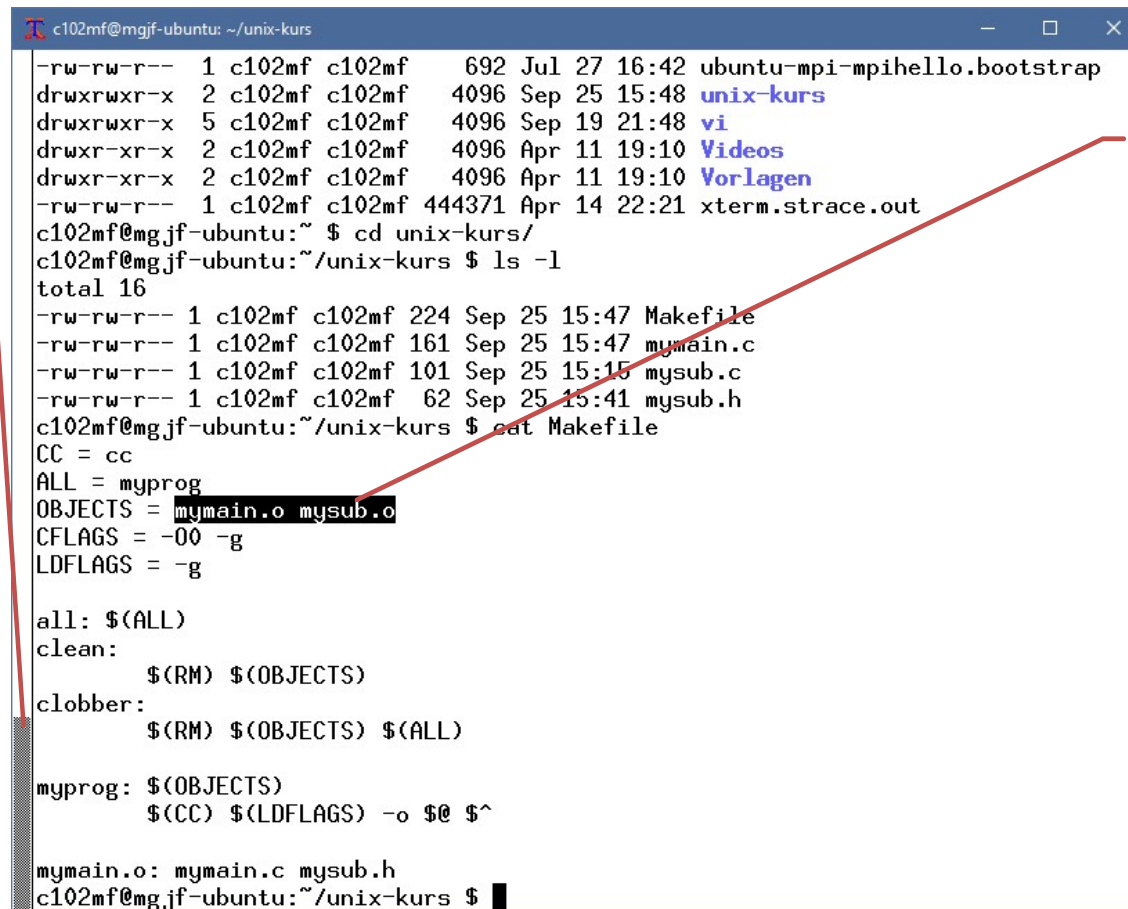
scroll bar mouse actions

right click: scroll back

left click: scroll forward

distance of mouse pointer from top = scroll amount

middle click or drag: scroll absolute



```
c102mf@mgjf-ubuntu: ~/unix-kurs
-rw-rw-r-- 1 c102mf c102mf 692 Jul 27 16:42 ubuntu-mpi-mpihello.bootstrap
drwxrwxr-x 2 c102mf c102mf 4096 Sep 25 15:48 unix-kurs
drwxrwxr-x 5 c102mf c102mf 4096 Sep 19 21:48 vi
drwxr-xr-x 2 c102mf c102mf 4096 Apr 11 19:10 Videos
drwxr-xr-x 2 c102mf c102mf 4096 Apr 11 19:10 Vorlagen
-rw-rw-r-- 1 c102mf c102mf 444371 Apr 14 22:21 xterm.strace.out
c102mf@mgjf-ubuntu:~$ cd unix-kurs/
c102mf@mgjf-ubuntu:~/unix-kurs$ ls -l
total 16
-rw-rw-r-- 1 c102mf c102mf 224 Sep 25 15:47 Makefile
-rw-rw-r-- 1 c102mf c102mf 161 Sep 25 15:47 mymain.c
-rw-rw-r-- 1 c102mf c102mf 101 Sep 25 15:15 mysub.c
-rw-rw-r-- 1 c102mf c102mf 62 Sep 25 15:41 mysub.h
c102mf@mgjf-ubuntu:~/unix-kurs$ cat Makefile
CC = cc
ALL = myprog
OBJECTS = mymain.o mysub.o
CFLAGS = -O0 -g
LDFLAGS = -g

all: $(ALL)
clean:
    $(RM) $(OBJECTS)
clobber:
    $(RM) $(OBJECTS) $(ALL)

myprog: $(OBJECTS)
    $(CC) $(LDFLAGS) -o $@ $^

mymain.o: mymain.c mysub.h
c102mf@mgjf-ubuntu:~/unix-kurs$
```

text area mouse actions

left click + drag: select text

left double click: select word

left triple click: select line

right click or drag: extend or reduce selection (both ends)

middle click: insert selection (or clipboard if enabled) a cursor position

Using Xterm Popup Menus

ctrl + left / middle / right mouse button gives popup menus

Main Options
Full Screen
Secure Keyboard
Allow SendEvents
Redraw Window
Log to File
Print-All Immediately
Print-All on Error
8-Bit Controls
Backarrow Key (BS/DEL)
✓Alt/NumLock Modifiers
Meta Sends Escape
Delete is DEL
Old Function-Keys
Termcap Function-Keys
Sun Function-Keys
VT220 Keyboard
Send STOP Signal
Send CONT Signal
Send INT Signal
Send HUP Signal
Send TERM Signal
Send KILL Signal
Quit

VT Options
✓Enable Scrollbar
✓Enable Jump Scroll
Enable Reverse Video
✓Enable Auto Wraparound
Enable Reverse Wraparound
Enable Auto Linefeed
Enable Application Cursor Keys
Enable Application Keypad
Scroll to Bottom on Key Press
✓Scroll to Bottom on Tty Output
Allow 80/132 Column Switching
✓Keep Selection
✓Select to Clipboard
Enable Visual Bell
Enable Bell Urgency
Enable Pop on Bell
Enable Blinking Cursor
✓Enable Alternate Screen Switching
Do Soft Reset
Do Full Reset
Reset and Clear Saved Lines
Show Tek Window
Switch to Tek Mode
Hide VT Window
Show Alternate Screen
✓Sixel Scrolling
✓Private Color Registers

VT Fonts
Default
Unreadable
Tiny
Small
Medium
Large
✓Huge
Escape Sequence
Selection
✓Bold Fonts
Line-Drawing Characters
✓Packed Font
✓Doublesized Characters
TrueType Fonts
✓UTF-8 Encoding
✓UTF-8 Fonts
UTF-8 Titles
✓Allow Color Ops
Allow Font Ops
Allow Termcap Ops
✓Allow Title Ops
Allow Window Ops

Setting Xterm Character Classes

Concept

double click selects **word** - **meaning** of **word** is configuration dependent
many new distributions define classes optimized for web users
programmers prefer words to be syntactic units

Recommendation

Use the original default character class definition in **.Xresources** or **.Xdefaults**:

```
XTerm*charClass: 0:32,1-8:1,9:32,10-  
31:1,32:32,33:33,34:34,35:35,36:36,37:37,38:38,39:39,40:40,41:41,42:42,43:43,44:44,45:45,46:46,4  
7:47,48-57:48,58:58,59:59,60:60,61:61,62:62,63:63,64:64,65-  
90:48,91:91,92:92,93:93,94:94,95:48,96:96,97-122:48,123:123,124:124,125:125,126:126,127-  
159:1,160:160,161:161,162:162,163:163,164:164,165:165,166:166,167:167,168:168,169:169,170:170,17  
1:171,172:172,173:173,174:174,175:175,176:176,177:177,178:178,179:179,180:180,181:181,182:182,18  
3:183,184:184,185:185,186:186,187:187,188:188,189:189,190:190,191:191,192-214:48,215:215,216-  
246:48,247:247,248-255:48
```

Note

Windows registry definition to give putty an xterm-like behavior is available

UNIX as a Programming Environment

Programming language C originated with UNIX

Default compilers for Linux: **GCC = the GNU Compiler Collection**

Supported languages, standards

- C (1990, 1999, 2011 + GNU extensions)
- C++ (1998, 2003, 2011, 2014, 2017), Objective-C
- GNU Fortran (supports Fortran 95, Fortran 90, Fortran 77)

“UNIX is an IDE”

Automate creation of programs and libraries from source using `make(1)`

UNIX utilities designed to support program development and file management

Program Example

mymain.c

```
#include <stdio.h>
#include "mysub.h"

int main(int argc, char **argv) {
    int i, j, k;
    sub1(&i, &j);
    sub2(i, j, &k);
    printf("%d + %d = %d\n", i, j, k);
}
```

mysub.h

```
void sub1(int *i, int *j) ;
void sub2(int i, int j, int *k) ;
```

mysub.c

```
void sub1(int *i, int *j) {
    *i = 2;
    *j = 3;
}

void sub2(int i, int j, int *k) {
    *k = i + j;
}
```

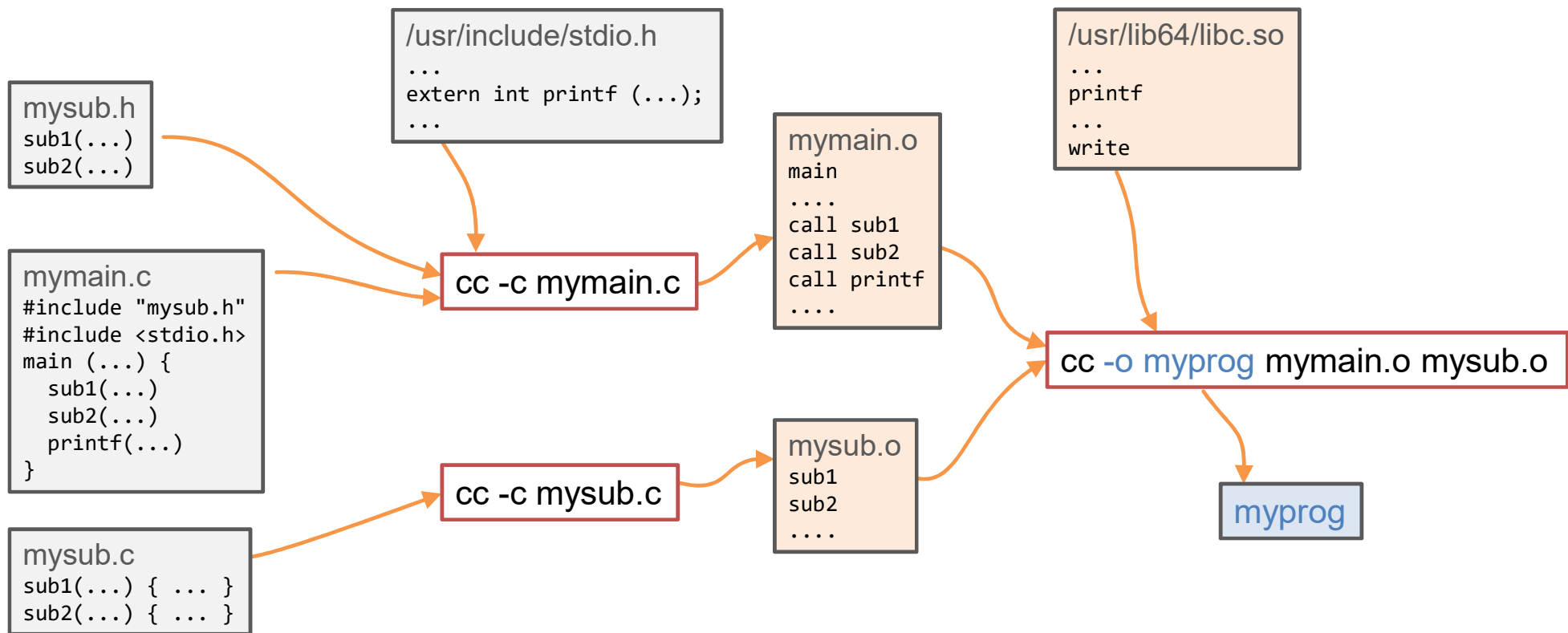
Understanding the Compilation Dataflow

`cc -c mymain.c` compile source of main program into object file (relocatable machine program)

`cc -c mysub.c` same for subroutines

`cc -o myprog mymain.o mysub.o` link object files and create executable program `myprog`

Note: certain files (`mysub.h`, `stdio.h`, `libc.so`, `*.o`) are opened / created implicitly (i.e. not on command line)



Using the Compiler

`compiler` [`option` ...] `file` ... Invoke the `compiler` driver: preprocess, compile, assemble, link program files

`cc gcc` C Language / GNU C

`c++ g++` C++ / GNU C++

`f95 gfortran` Fortran / GNU Fortran

Other vendors use different driver names, e.g. Intel: `icc icpc ifort` and different options

File naming conventions: `file suffix` indicates file language and type

Suffix

Type

`.c` C source program

`.C .cc .cxx .cpp` etc. C++ source program

`.f .f90 .f95 .f03 .f08` Fortran source program conforming to Fortran 77, 90, 95, 2003, 2008 standard
`.o` translated object file

`.a .so` library (static or shared object) to search for function definitions

(none) executable. default: `a.out`

Options

`-c` only compile `file.suffix`, do not link, write output to file with `.suffix` replaced by `.o`

`-o name` compile: use `name` instead of `file.o` (compile) or `a.out` (link)

`-g -pg` compile and link: create symbol tables for debugging (`-g`) or extra code for execution profiling (`-pg`)

`-O -O0 -O1 -O2 -O3` compile: optimization level. `-O0` is default, `-O1` and `-O` are the same
optimization **increases execution speed** and **reduces code size**, may **change program semantics**

`-I dir` compile: add `dir` to search path for `#include <....>` header files

`-l name` link: search function definitions in files `libname.a` and `libname.so` in standard library search path

`-L dir` link: add `dir` to library search path for `-l` argument

(many more - optimization, target architecture, language standard / extensions, warning+debugging,)

Using make to Automate the Compiler Workflow

Concept

Large programming project may consist of many source files, complex dependencies

Compile and link steps must be executed in correct order

After changes, only those parts affected by change need recompiling / linking

Make

Uses **built in** and **user defined rules** and **dependencies** to automate and optimize compilation workflow

User analyzes dependencies and codes these in **Makefile** (or **makefile**)

The **make** utility reads **Makefile** and runs compile / link steps necessary to (re)create target

Makefile Syntax

Makefile consists of **rules**.

Each rule is

one dependency line	target: <i>prerequisite</i> ...
zero or more recipe lines	→ <i>command</i>

where → is the **TAB** (^I \t) character (invisible)

make [<i>target</i>]	rebuilds the named <i>target</i> by...
	1. making all prerequisites (recursive)
	2. executing recipe lines for current rule
	(if empty: use builtin rule if existent)
<i>target</i>	default: first target in Makefile

Why the tab in column 1? Yacc was new, Lex was brand new. I hadn't tried either, so I figured this would be a good excuse to learn. After getting myself snarled up with my first stab at Lex, I just did something simple with the pattern newline-tab. It worked, it stayed. And then a few weeks later I had a user population of about a dozen, most of them friends, and I didn't want to screw up my embedded base. The rest, sadly, is history.
— Stuart Feldman

Example Makefile

Example Makefile (version 1: all rules explicit)

corresponds to example in dataflow graph

note: you must explicitly declare dependencies (e.g. [mymain.o](#) also depends on [myprog.h](#))
the `makedepend` utility can automate this

```
myprog: mymain.o mysub.o
→      cc -o myprog mymain.o mysub.o
mymain.o: mymain.c mysub.h
→      cc -c mymain.c
mysub.o: mysub.c
→      cc -c mysub.c
```

make

first run: compiles `mymain.c`, `mysub.c`, links

after changing `mymain.c` or `mysub.h`: compiles only `mymain.c`, links

Invoking make, Variables

`make` [*option* ...] [*target* ...]

- B unconditionally make all targets
- C *dir* change to *dir* before reading Makefile (used in recursive make)
- f *file* use *file* instead of Makefile
- j *njobs* run *njobs* (commands) simultaneously (may be unreliable)
- n dry run
- p print all rules and macros / variables to stdout

`info make` | `less` complete Make documentation

Variables

- defining variable: all environment variables are copied to Make variables
`NAME = definition` in Makefile - creates variable, overrides environment (except with -e)
- using variable: `$(NAME)` in Makefile is replaced by *definition* (dependency and command lines)
- special variables:
- `$@` name of current target
 - `$<` name of first requisite
 - `$$` list of all requisites
- standard variables:
- `CC CFLAGS` Name of C Compiler and list of compiler options (flags)
 - `CXX CXXFLAGS` C++ Compiler and flags
 - `FC FFLAGS` Fortran Compiler and flags
 - `LDLAGS` Flags used when linking programs
 - `RM` Command used to remove files

Example Makefile: Using Variables and Macros

Example Makefile (version 2: using variables and macros, cleanup)

Goal: reduce redundancy

Note: using a different C compiler now only involves changing CC

```
CC = cc
ALL = myprog          # targets to make. add more targets here
OBJECTS = mymain.o mysub.o
CFLAGS = -O0 -g       # debug
LDFLAGS = -g          # debug

all: $(ALL)           # first rule: catch-all for make called with no arguments. add more targets here
clean:
→      - $(RM) $(OBJECTS)
clobber:
→      - $(RM) $(ALL) $(OBJECTS)

myprog: $(OBJECTS)
→      $(CC) $(LDFLAGS) -o $@ $^

mymain.o: mymain.c mysub.h
→      $(CC) $(CFLAGS) -c $<
mysub.o: mysub.c
→      $(CC) $(CFLAGS) -c $<
```

Using Predefined Rules to Further Simplify Makefiles

Make has many predefined rules → may omit many trivial recipes

`make -p` in directory w/ no Makefile: print out predefined rules

Example Makefile (version 3: using predefined rules)

```
CC = cc
ALL = myprog
OBJECTS = mymain.o mysub.o
CFLAGS = -O0 -g
LDFLAGS = -g

all: $(ALL)
clean:
→ - $(RM) $(OBJECTS)
clobber:
→ - $(RM) $(ALL) $(OBJECTS)

myprog: $(OBJECTS)
→ $(CC) $(LDFLAGS) -o $@ $^

mymain.o: mymain.c mysub.h
```

Installing Third Party Software into Your HOME or SCRATCH

Goal

Install third party software as non-root user from sources

Workflow

- Read the instructions provided by the program authors
- If source distribution release is in a tar archive (“tarball”)
 - Download and extract tarball, cd into source directory
- If Github is used
 - get latest development version with git clone, cd into source directory, git checkout
 - typically it is necessary to do a `./autogen.sh` or similar to create the Configure script
- Read README, INSTALL and other files that could contain instructions
- `./Configure --prefix=$HOME` (or `$SCRATCH`) discover facts about your system and create Makefile
- `make`
- `make test` if provided
- `make install` copy executable, libraries, man pages into `$PREFIX/bin`, `$PREFIX/lib`, `$PREFIX/man`

Debugging Programs

Goal: Diagnosing and correcting program errors:

- insert print statements, compile, and run. Repeat over and over until problem found. NOT RECOMMENDED
- execute program under **debugger**.
 - controlled execution (stepwise, up to breakpoint, when in function, when condition is met)
 - displaying values of variables, stack trace, etc.

Workflow

- Compile program with options **-g** and **-O0**, link with **-g**
-g creates symbol tables, allowing debugger to identify source lines and variables.
debugging optimized programs is possible, but is “fuzzy”, and variables may be optimized away
- Run executable under debugger. Most popular: **GNU debugger**. Commercial debugger for || programs: TotalView
gdb name [core] invokes GNU debugger for program **name** and issue command prompt
- Issue debugging commands. Most used commands:

b [<i>file</i> :]{ <i>func</i> <i>line</i> }	break	set breakpoint in function <i>func</i> or line number <i>line</i>
r [<i>arg</i> ...]	run	run program, supplying arguments
wh bt	where, backtrace	display current location, call stack
p <i>expr</i>	print	display value of <i>expr</i> (use syntax of debugged program)
c	continue	continue running program
s n	step next	stepwise execution: step (into subprogram), next (source line)
l [<i>file</i> :]{ <i>func</i> <i>line</i> }	list	display source lines
h	help	
q	quit	
- Fix problem and recompile

Profiling Programs

Concept

Optimizing programs: need knowledge where program spends its time, what happens where
Create execution profile showing execution times of routines, including call graph

Workflow (unverified)

Compile with `-pg`

Run representative test case - creates execution profile in `gmon.out`

Run `gprof [options] executable [gmon.out]`

... displays call graph profile.

Note

Numerous tools, some by compiler vendors

Modern CPUs have **performance counters**, allowing simultaneous timing of multiple specific events, allowing line-sharp **hot spot analysis**, e.g. cache misses vs. instructions executed

Multiple profiling methods (statistic sampling vs. code instrumentation, event counting, timing, ...)

Look e.g for Open|SpeedShop, HPC Toolkit, TAU, ...

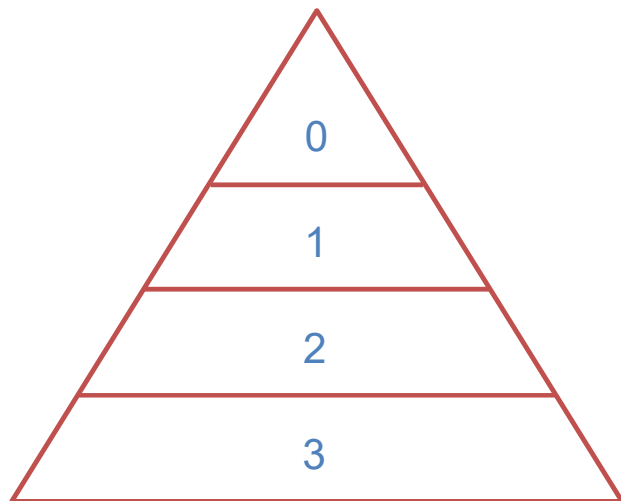
Understanding the HPC Ecosystem

What it is

HPC Cluster = set of interconnected independent **compute servers** (**nodes**)

- **consistent setup** & **software** installation, **modular software** environment
- shared **HOME** and high performance / high capacity **Scratch directories**
- high bandwidth low latency **interconnect** (Infiniband) + software supporting **parallel computation** (MPI)
- load management (**batch**) **system** for placement of sequential and parallel jobs on nodes

HPC tier model



Supranational HPC installations
e.g. PRACE

National HPC installations
e.g. VSC

Local HPC clusters
e.g. LEO, MACH(*)

Workgroup clusters

HPC Enabling Research

Capability Computing

Big machine enables
large scale computations that
cannot be solved on smaller system

Capacity Computing

Big number of CPUs + memory
used to solve many instances of
simple problems in much shorter time

(*) MACH actually not a cluster

Understanding the Architecture of an HPC Cluster

Note: names may vary

