${Unix_Tools}

Markus Kuhn

Computer Laboratory, University of Cambridge

https://www.cl.cam.ac.uk/teaching/current/UnixTools/

Computer Science Tripos – Part IB

Why do we teach Unix Tools?

- Second most popular OS family (after Microsoft Windows)
- Many elements of Unix have became part of common computer science folklore, terminology & tradition over the past 25 years and influenced many other systems (including DOS/Windows)
- Many Unix tools have been ported and become popular on other platforms: full Unix environment in Apple's macOS, Cygwin, WSL
- Your future project supervisors and employers are likely to expect you to be fluent under Unix as a development environment
- Good examples for high-functionality user interfaces

This short lecture course can only give you a first overview. You need to spend at least 2–3 times as many hours with e.g. MCS Linux to

- explore the tools mentioned
- solve exercises (which often involve reading documentation to understand important details skipped in the lecture)

Prerequisites

The most basic Unix commands and concepts:

- Hierarchical file space: directories (folders), root of a directory tree
- absolute path: `/home/mgk25/wg/teaching/unixtools/Makefile`
- relative path: `f2.txt` or `wg/teaching/` or `../../proj/phd.tex`
- print current working directory: `pwd`
- change the current working directory: `cd` *`path`*
- listing a directory: `ls` or `ls -la` or `ls` *`path`*
- use of a plain-text file editor: e.g. `emacs` *`path`* or `nano` *`path`*
- deleting a file: `rm` *`path`*
- make a new directory: `mkdir` *`path`*
- deleting an (empty) directory: `rmdir` *`path`*
- rename or move a file or directory: `mv` *`oldpath newpath`*
- leave: `exit`

Machines to practice on remotely (via SSH/PuTTY):
`linux.cl.ds.cam.ac.uk`, `slogin.cl.cam.ac.uk`

Brief review of Unix history

- "First Edition" developed at AT&T Bell Labs during 1968–71 by Ken Thompson and Dennis Ritchie for a PDP 11
- Rewritten in C in 1973
- Sixth Edition (1975) first widely available version
- Seventh Edition in 1979, UNIX 32V for VAX
- During 1980s independent continued development at AT&T ("System V Unix") and Berkeley University ("BSD Unix")
- Commercial variants (Solaris, SCO, HP/UX, AIX, IRIX, ...)
- IEEE and ISO standardisation of a *Portable Operating System Interface based on Unix (POSIX)* in 1989, later also *Single Unix Specification* by X/Open, both merged in 2001

  The POSIX standard is freely available online: `http://www.unix.org/` and `https://pubs.opengroup.org/onlinepubs/9699919799/`

A brief history of free Unix

- In 1983, Richard Stallman (MIT) initiates a free reimplementation of Unix called GNU ("GNU's Not Unix") leading to an editor (emacs), compiler (gcc), debugger (gdb), and numerous other tools.
- In 1991, Linus Torvalds (Helsinki CS undergraduate) starts development of a free POSIX-compatible kernel, later nicknamed *Linux*, which was rapidly complemented by existing GNU tools and contributions from volunteers and industry to form a full Unix replacement.
- In 1991, Berkeley University releases a free version of BSD Unix, after removing remaining proprietary AT&T code. Volunteer projects emerge to continue its development (FreeBSD, NetBSD, OpenBSD).
- In 2000, Apple releases *Darwin*, the now open-source core components of their OS X and iOS operating systems. Volunteer projects emerge to port many Unix tools onto Darwin (Homebrew, Fink, MacPorts, GNU Darwin, etc.).

Free software license concepts

- **public domain:** authors waive all copyright
- **"MIT/BSD" licences:** allow you to copy, redistribute and modify the software in any way as long as
  - you respect the identity and rights of the author (preserve copyright notice and licence terms in source code and documentation)
  - you agree not sue the author over software quality (accept exclusion of liability and warranty)
- **GNU General Public Licence (GPL):** requires in addition that any derived work distributed or published
  - must be licensed under the terms of the GPL
  - must have its source code made publicly available

Numerous refinements and variants of these licences have been written. For more information on the various opensource licence types and their philosophies: `https://opensource.org/licenses`

Original Unix user interfaces

The initial I/O devices were teletype terminals . . .

Photo: Bell Labs

VT100 terminals

. . . and later video display terminals such as the DEC VT100, all providing 80 characters-per-line fixed-width ASCII output. Their communications protocol is still used today in graphical windowing environments via "terminal emulators" (e.g., xterm, konsole).

Photo: `http://www.catb.org/esr/writings/taouu/html/`

The VT100 was the first video terminal with microprocessor, and the first to implement the ANSI X3.64 (= ECMA-48) control functions. For instance, "$^{\mathrm{E}}\mathrm{s}_{\mathrm{C}}$`[7m`" activates inverse mode and "$^{\mathrm{E}}\mathrm{s}_{\mathrm{C}}$`[0m`" returns to normal, where $^{\mathrm{E}}\mathrm{s}_{\mathrm{C}}$ is the ASCII "escape" control character ($27 = 0\mathrm{x}1\mathrm{B}$).

```
https://www.vt100.net/
http://www.ecma-international.org/publications/standards/Ecma-048.htm
man console_codes
```

Unix tools design philosophy

- Compact and concise input syntax, making full use of ASCII repertoire to minimise keystrokes
- Output format should be simple and easily usable as input for other programs
- Programs can be joined together in “pipes” and “scripts” to solve more complex problems
- Each tool originally performed a simple single function
- Prefer reusing existing tools with minor extension to rewriting a new tool from scratch
- The main user-interface software (“shell”) is a normal replaceable program without special privileges
- Support for automating routine tasks

Brian W. Kernighan, Rob Pike: The Unix Programming Environment. Prentice-Hall, 1984.

Unix documentation

Most Unix documentation can be read from the command line.
Classic manual sections: user commands (1), system calls (2), library functions (3), devices (4), file formats (5).

- The `man` tool searches for the manual page file ($\rightarrow$ `$MANPATH`) and activates two further tools (`nroff` text formatter and `more` text-file viewer). Add optional section number to disambiguate:

  ```
  $ man 3 printf      # C subroutine, not command
  ```

  Honesty in documentation: Unix manual pages traditionally include a BUGS section.

- `xman`: X11 GUI variant, offers a table of contents

- `info`: alternative GNU hypertext documentation system

  Invoke with `info` from the shell of with `C-h i` from emacs. Use `M`(enu) key to select topic or [Enter] to select hyperlink under cursor, `N`(ext)/`P`(rev)/`U`(p)/`D`(irectory) to navigate document tree, Emacs search function (Ctrl-S), and finally `Q`(uit).

- Check `/usr/share/doc/` and Web for further documentation.

Examples of Unix tools

`man`, `apropos`, `xman`, `info`
: help/documentation browser

`more`, `less`
: plaintext file viewer

`ls`, `find`
: list/traverse directories, search

`cp`, `mv`, `rm`, `touch`, `ln`
: copy, move/rename, remove, renew files, link/shortcut files

`mkdir`, `rmdir`
: make/remove directories

`cat`, `dd`, `head`, `tail`
: concatenate/split files

`du`, `df`, `quota`, `rquota`
: examine disk space used and free

`ps`, `top`, `free`, `uptime`, `w`
: process table and system load

`vi`, `emacs`, `nano`
: interactive editors

`cc`, `gcc`
: C compilers

`make`
: project builder

`cmp`, `diff`, `patch`
: compare files, apply patches

`rcs`, `cvs`, `svn`, `git`, `hg`, `bzr`
: revision control systems

`adb`, `gdb`
: debuggers

`awk`, `perl`, `python`, `tcl`, `ruby`
: scripting languages

`m4`, `cpp`
: macro processors

`sed`, `tr`
: edit streams, replace characters

`sort`, `grep`, `cut`
: sort/search lines of text, extract columns

`nroff`, `troff`, `tex`, `latex`
: text formatters

`mail`
: send or process email messages

Examples of Unix tools (cont'd)

`telnet`, `ftp`, `rlogin`, `finger`, `talk`, `ping`, `traceroute`, `wget`, `curl`, `ssh`, `scp`, `rsync`, `hostname`, `host`, `ifconfig`, `route`
: network tools

`xterm`
: VT100 terminal emulator

`tar`, `cpio`, `compress`, `zip`, `gzip`, `bzip2`
: file packaging and compression

`echo`, `cd`, `pushd`, `popd`, `exit`, `ulimit`, `time`, `history`
: builtin shell commands

`fg`, `bg`, `jobs`, `kill`
: builtin shell job control

`date`, `xclock`
: clocks

`crontab`
: schedule jobs to run periodically

`which`, `whereis`
: locate command file

`clear`, `reset`
: clear screen, reset terminal

`stty`
: configure terminal driver

`display`, `ghostview`, `okular`
: graphics file viewers

`xfig`, `tgif`, `gimp`, `inkscape`
: graphics drawing tools

`*topnm`, `pnmto*`, `[cd]jpeg`
: graphics format converters

`bc`
: calculator

`passwd`
: change your password

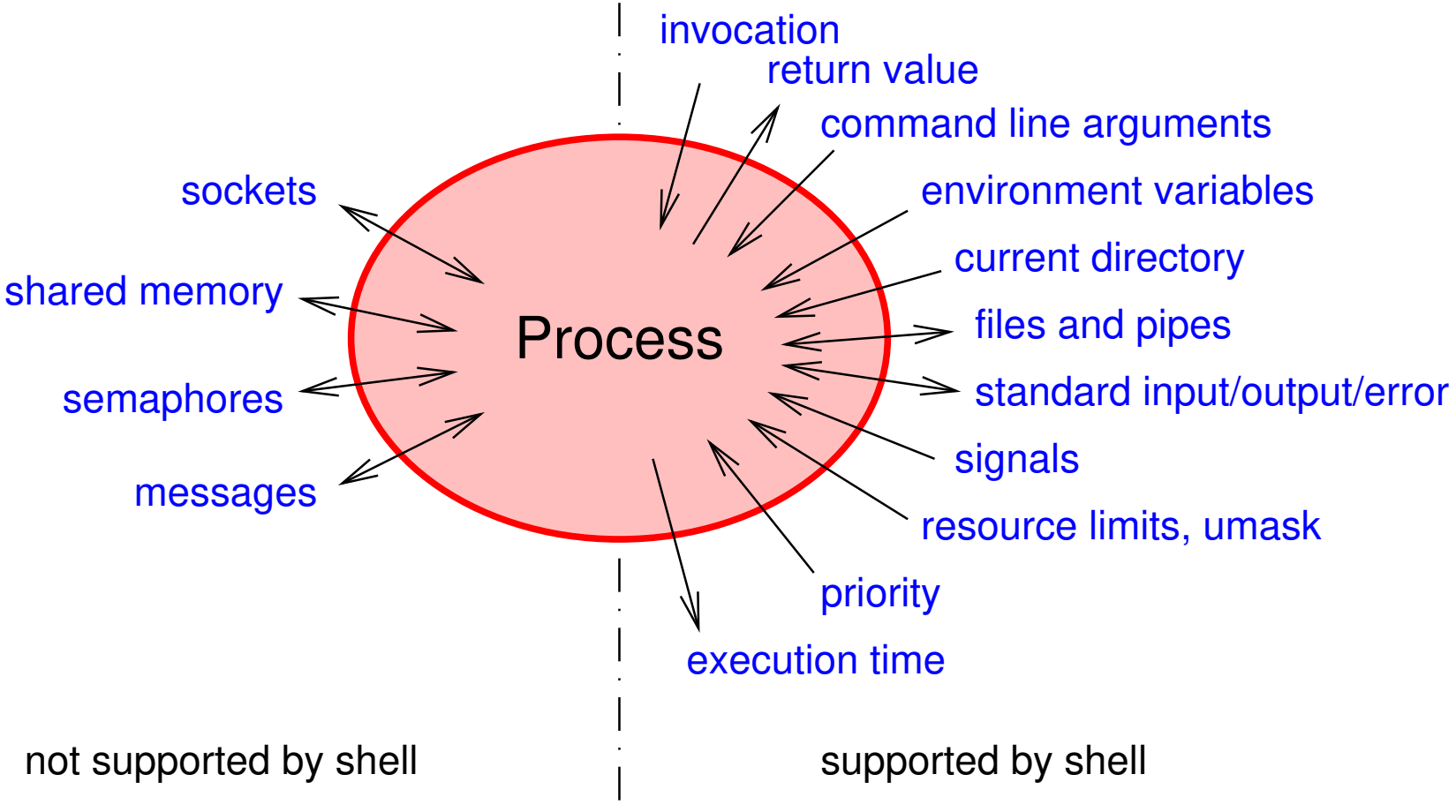
`chmod`
: change file permissions

`lex`, `yacc`, `flex`, `bison`
: scanner/parser generators

The Unix shell

- The user program that Unix starts automatically after a login
- Allows the user to interactively start, stop, suspend, and resume other programs and control the access of programs to the terminal
- Supports automation by executing files of commands ("shell scripts"), provides programming language constructs (variables, string expressions, conditional branches, loops, concurrency)
- Simplifies file selection via keyboard (regular expressions, file name completion)
- Simplifies entry of command arguments with editing and history functions
- Most common shell ("`sh`") developed 1975 by Stephen Bourne, modern GNU replacement is "`bash`" ("Bourne-Again SHell")

Unix inter-process communication mechanisms



Command-line arguments, return value, environment

A Unix C program is invoked by calling its `main()` function with:

- a list of strings `argv` as an argument
- a list of strings `environ` as a predefined global variable

```
#include <stdio.h>
extern char **environ;

int main(int argc, char **argv)
{
  int i;
  printf("Command line arguments:\n");
  for (i = 0; i < argc; i++)
    puts(argv[i]);
  printf("Environment:\n");
  for (i = 0; environ[i] != NULL; i++)
    puts(environ[i]);
  return 0;
}
```

Environment strings have the form

$$name{=}value$$

where *name* is free of "=".

Argument `argv[0]` is usually the name or path of the program.

Convention: `main() == 0` signals success, other values signal errors to calling process.

File descriptors

Unix processes access files in three steps:

- Provide kernel in `open()` or `creat()` system call a path name and get in return an integer "file descriptor".
- Provide in `read()`, `write()`, and `seek()` system calls an opened file descriptor along with data.
- Finally, call `close()` to release any data structures associated with an opened file (position pointer, buffers, etc.).

The `lsof` tool lists the files currently opened by any process. Under Linux, file descriptor lists and other kernel data can be accessed via the simulated file system mounted under `/proc`.

As a convention, the shell opens three file descriptors for each process:

- 0 = standard input (for reading the data to be processed)
- 1 = standard output (for the resulting output data)
- 2 = standard error (for error messages)

Basic shell syntax – pipes

Start a program and connect the three default file descriptors stdin, stdout, and stderr to the terminal:

```
$ command
```

Connect stdout of `command1` to stdin of `command2` and stdout of `command2` to stdin of `command3` by forming a pipe:

```
$ command1 | command2 | command3
```

Also connects terminal to stdin of `command1`, to stdout of `command3`, and to stderr of all three.

Note how this function concatenation notation makes the addition of command arguments somewhat clearer compared to the mathematical notation `command3(command2(command1(arg1), arg2), arg3)`:

```
$ ls -la | sort -n -k5 | less
```

Basic shell syntax – sequences

Execute several commands (or entire pipes) in sequence:

```
$ command1 ; command2 ; command3
```

For example:

```
$ date ; host grace-1.linux.cl.ds.cam.ac.uk
Thu  5 Nov 16:26:07 GMT 2020
grace-1.linux.cl.ds.cam.ac.uk has address 131.111.9.26
```

Conditional execution depending on success of previous command (as in logic-expression short-cut):

```
$ make ftest && ./ftest
$ ./ftest || echo 'Test failed!'
```

Return value 0 for success is interpreted as Boolean value "true", other return values for problems or failure as "false". The trivial tools `true` and `false` simply return 0 and 1, respectively.

File redirecting – basics

Send stdout to file

```
$ command >filename
```

Append stdout to file

```
$ command >>filename
```

Send both stdout and stderr to the same file. First redirect stdout to `filename`, then redirect stderr (file descriptor 2) to where stdout goes (target of file descriptor 1 = `&1`):

```
$ command >filename 2>&1
```

Feed stdin from file

```
$ command <filename
```

File redirecting – advanced

Open other file descriptors for input, output, or both

```
$ command 0<in 1>out 2>>log 3<auxin 4>auxout 5<>data
```

"Here Documents" allow us to insert data into shell scripts directly such that the shell will feed it into a command via standard input. The << is followed immediately by an end-of-text marker string.

```
$ tr <<THEEND A-MN-Za-mn-z N-ZA-Mn-za-m
> Vs lbh zhfg cbfg n ehqr wbxr ba HFRARG, ebgngr gur
> nycunorg ol 13 punenpgref naq nqq n jneavat.
> THEEND
```

Redirecting to or from `/dev/tcp/`*`hostname`*`/`*`port`* will open a TCP socket connection:

```
{ echo -e "GET /iso-paper.c HTTP/1.0\r" >&3 ; \
  echo -e "Host: mgk25.user.srcf.net\r\n\r" >&3 ; cat <&3 ; \
} 3<>/dev/tcp/mgk25.user.srcf.net/80
```

The above example is a bash implementation of a simple HTTP client (web browser). It downloads and displays the file `http://mgk25.user.srcf.net/iso-paper.c`.

Bash's `/dev/tcp/...` feature may be disabled in some Linux distributions (security concerns). Normally use "`curl http://mgk25.user.srcf.net/iso-paper.c`" instead.

Command-line argument conventions

Each program receives from the caller as a parameter an array of strings (`argv`). The shell places into the `argv` parameters the words entered following the command name, after several preprocessing steps have been applied first.

```
$ cp 'Lecture Timetable.pdf' lecture-timetable.pdf
$ mv *.bak old-files/
```

Command options are by convention single letters prefixed by a hyphen ("`-h`"). Unless followed by option parameters, single character flag options can often be concatenated:

```
$ ls -l -a -t
$ ls -lat
```

GNU tools offer alternatively long option names prefixed by two hyphens ("`--help`"). Arguments not starting with hyphens are typically filenames, hostnames, URLs, etc.

```
$ gcc --version
$ curl --head https://www.cl.cam.ac.uk/
```

Command-line argument conventions (cont'd)

The special option "--" signals in many tools that subsequent words are arguments, not options. This provides one way to access filenames starting with a hyphen:

```
$ rm -- -i
$ rm ./-i
```

The special filename "-" signals often that standard input/output should be used instead of a file.

All these are conventions that most – but not all – tools implement (usually via the `getopt` library function), so check the respective manual first.

The shell remains ignorant of these "-" conventions!

`https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html`

Shell command-line preprocessing

A number of punctuation characters in a command line are part of the shell control syntax

```
| & ; ( ) < >
```

or can trigger special convenience substitutions before `argv` is handed over to the called program:

- brace expansion: `{,}`
- tilde expansion: `~`
- parameter expansion: `$`
- pathname expansion / filename matching: `* ? []`
- quote removal: `\ ' "`

Brace and tilde expansion

Brace expansion

Provides for convenient entry of words with repeated substrings:

```
$ echo a{b,c,d}e
abe ace ade
$ echo {mgk25,fapp2,rja14}@cam.ac.uk
mgk25@cam.ac.uk fapp2@cam.ac.uk rja14@cam.ac.uk
$ rm slides.{bak,aux,dvi,log,ps}
```

This `bash` extension is not required by the POSIX standard; e.g. Ubuntu Linux `/bin/sh` lacks it.

Tilde expansion

Provides convenient entry of home directory pathname:

```
$ echo ~pb22 ~/Mail/inbox
/home/pb22 /home/mgk25/Mail/inbox
```

The builtin `echo` command simply outputs `argv` to stdout and is useful for demonstrating command-line expansion and for single-line text output in scripts.

Parameter and command expansion

Substituted with the values of shell variables

```
$ OBJFILE=skipjack.o
$ echo ${OBJFILE} ${OBJFILE%.o}.c
skipjack.o skipjack.c
$ echo ${HOME} ${PATH} ${LOGNAME}
/homes/mgk25 /bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin mgk25
```

or the standard output lines of commands

```
$ which emacs
/usr/bin/emacs
$ echo $(which emacs)
/usr/bin/emacs
$ ls -l $(which emacs)
-rwxr-xr-x    2 root     system    3471896 Mar 16  2001 /usr/bin/emacs
```

Shorter alternatives: variables without braces and command substitution with grave accent (`) or, with older fonts, back quote (‘)

```
$ echo $OBJFILE
skipjack.o
$ echo `which emacs`
/usr/bin/emacs
```

Pathname expansion

Command-line arguments containing `?`, `*`, or `[...]` are interpreted as regular expression patterns and will be substituted with a list of all matching filenames.

- `?` stands for an arbitrary single character
- `*` stands for an arbitrary sequence of zero or more characters
- `[...]` stands for one character out of a specified set. Use "–" to specify range of characters and "!" to complement set. Certain character classes can be named within `[:...:]`.

None of the above will match a dot at the start of a filename, which is the naming convention for hidden files.

Examples:

```
*.bak [A-Za-z]*.??? [[:alpha:]]* [!A-Z] .??* files/*/*.o
```

Quote removal

Three quotation mechanisms are available to enter the special characters in command-line arguments without triggering the corresponding shell substitution:

- `'...'` suppresses all special character meanings
- `"..."` suppresses all special character meanings, except for

  `$ \ ``
- `\` suppresses all special character meanings for the immediately following character

Example:

```
$ echo '$$$' "* * * $HOME * * *" \$HOME
$$$ * * * /homes/mgk25 * * * $HOME
```

The bash extension `$'...'` provides access to the full C string quoting syntax. For example `$'\x1b'` is the ASCII ESC character.

Job control

Start command or entire pipe as a background job, without connecting stdin to terminal:

```
$ command &
[1] 4739
$ ./testrun 2>&1 | gzip -9c >results.gz &
[2] 4741
$ ./testrun1 & ./testrun2 & ./testrun3 &
[3] 5106
[4] 5107
[5] 5108
```

Shell prints both a job number (identifying all processes in pipe) as well as process ID of last process in pipe. Shell will list all its jobs with the `jobs` command, where a + sign marks the last stopped (default) job.

Job control (cont'd)

Foreground job: Stdin connected to terminal, shell prompt delayed until process exits, keyboard signals delivered to this single job.

Background job: Stdin disconnected (read attempt will suspend job), next shell prompt appears immediately, keyboard signals not delivered, shell prints notification when job terminates.

Keyboard signals: (keys can be changed with `stty` tool)

- Ctrl-C "intr" (SIGINT=2) by default aborts process
- Ctrl-\ "quit" (SIGQUIT=3) aborts process with core dump
- Ctrl-Z "susp" (SIGSTOP=19) suspends process

Another important signal (not available via keyboard):

- SIGKILL=9 destroys process immediately

Job control (cont'd)

Job control commands:

- `fg` resumes suspended job in foreground
- `bg` resumes suspended job in background
- `kill` sends signal to job or process

Job control commands accept as arguments

- process ID
- % + job number
- % + command name

Examples:

```
$ ghostview                          # press Ctrl-Z
[6]+  Stopped                 ghostview
$ bg
$ kill %6
```

Job control (cont'd)

A few more job control hints:

- `kill -9 ...` sends SIGKILL to process. Should only be used as a last resort, if a normal kill (which sends SIGINT) failed, otherwise program has no chance to clean up resources before it terminates.
- The `jobs` command shows only jobs of the current shell, while `ps` and `top` list entire process table. Options for `ps` differ significantly between System V and BSD derivatives, check man pages.
- `fg %-` or just `%-` runs previously stopped job in foreground, which allows you to switch between several programs conveniently.

Shell variables

Serve both as variables (of type string) in shell programming as well as environment variables for communication with programs.

Set *variable* to *value*:

```
variable=value
```

Note: No whitespace before or after "=" allowed.

Make variable visible to called programs:

```
export variable
export variable=value
```

Modify environment variables for one command only:

```
variable1=value variable2=value command
```

"`set`" shows all shell variables
"`printenv`" shows all (exported) environment variables.

Standard environment variables

- `$PATH` — Colon-separated list of directories in which shell looks for commands (e.g., "`/usr/local/bin:/bin:/usr/bin`").
  Should never contain ".", at least not at beginning. Why?
- `$LD_LIBRARY_PATH` — Colon-separated list of directories where the loader looks for shared libraries (see `man ld.so`)
- `$LANG`, `$LC_*` — Your "locale", the name of a system-wide configuration file with information about your character set and language/country conventions (e.g., "`en_GB.UTF-8`").

  `$LC_*` sets locale only for one category, e.g. `$LC_CTYPE` for character set and `$LC_COLLATE` for sorting order; `$LANG` sets the default and `$LC_ALL` overrides all. "`locale -a`" lists all available locales.

  The local-dependent sorting order, implemented by the C library function `strcoll()`, is a complex multi-pass algorithm; for the much simpler ASCII-byte order, as implemented by `strcmp()`, use `export LC_COLLATE=C`

  ```
  $ LC_TIME=en_US.UTF-8 date
  ```

- `$TZ` — Specification of your timezone (mainly for remote users)

  ```
  $ TZ=GMT0BST-1,M3.5.0/01:00,M10.5.0/02:00 date    # POSIX syntax
  $ TZ=America/New_York date                        # Olson tz database
  ```

Standard environment variables (cont'd)

- `$HOME` — Your home directory, also available as "~".
- `$USER`, `$LOGNAME` — Your login name.
- `$OLDPWD` — Previous working directory, also available as "~-".
- `$PS1` — The normal command prompt, e.g.

```
$ PS1='\[\033[7m\]\u@\h:\W \!\$\[\033[m\] '
mgk25@ely:unixtools 12$
```

- `$PRINTER` — The default printer for `lpr`, `lpq` and `lprm`.
- `$TERM` — The terminal type (usually `xterm` or `vt100`).
- `$PAGER`/`$EDITOR` — The default pager/editor (usually `less` and `emacs`, respectively).
- `$DISPLAY` — The X server that X clients shall use.

man 7 environ

Executable files and scripts

Many files signal their format in the first few "magic" bytes of the file content (e.g., `0x7f,'E','L','F'` signals the System V *Executable and Linkable Format*, which is also used by Linux and Solaris).

The "`file`" tool identifies hundreds of file formats and some parameters based on a database of these "magic" bytes:

```
$ file $(which ls)
/bin/ls: ELF 32-bit LSB executable, Intel 80386
```

The kernel recognizes files starting with the magic bytes "`#!`" as "scripts" that are intended for processing by the interpreter named in the rest of the line, e.g. a `bash` script starts with

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

If the kernel does not recognize a command file format, the shell will interpret each line of it, therefore, the "`#!`" is optional for shell scripts.

Use "`chmod +x file`" and "`./file`", or "`bash file`".

Plain-text files

- File is a sequence of lines (trad. each < 80 characters long).
- Characters ASCII encoded (or extension: ISO 8859-1 "Latin 1", Microsoft's CP1252, EUC, Unicode UTF-8, etc.)
- Each line ends with a special "line feed" control character (LF, Ctrl-J, byte value: $10_{10} = 0A_{16}$).

```
$ echo hello | hd
00000000  68 65 6c 6c 6f 0a                                 |hello.|
00000006
```

- "Horizontal tabulator" (HT, TAB, Ctrl-I, byte value: 9) advances the position of the next character to the next multiple-of-8 column.

Some systems (e.g., DOS, Windows, some Internet protocols) end each line instead with a **two** control-character sequence: "carriage return" (CR, Ctrl-M, $13_{10} = 0D_{16}$) plus "line feed".

Different end-of-line conventions and different ASCII extensions make conversion of plain-text files necessary (dos2unix, iconv). Very annoying!

Alternative "flowed" plain-text format: no LF is stored inside a paragraph, line wrapping of paragraph happens only while a file is being displayed, LF terminates paragraph.

Some plain-text editors (e.g., Windows Notepad) start each UTF-8 plain-text file with a Unicode "Byte Order Mark" (BOM, U+FEFF, EF_{16} BB_{16} BF_{16}), which is not normally used under Unix.

Shell compound commands

A *list* is a sequence of one or more pipelines separated by ";", "&", "&&" or "||", and optionally terminated by one of ";", "&" or end-of-line. The return value of a list is that of the last command executed in it.

- `(` ⟨*list*⟩ `)` executes list in a subshell (e.g., to contain state changes)
- `{` ⟨*list*⟩ `;` `}` groups a list (to override operator priorities)
- `for` ⟨*variable*⟩ `in` ⟨*words*⟩ `;` `do` ⟨*list*⟩ `;` `done`

  Expands *words* like command-line arguments, assigns one at a time to the ⟨*variable*⟩, and executes ⟨*list*⟩ for each. Example:

  ```
  for f in *.txt ; do cp $f $f.bak ; done
  ```

- `if` ⟨*list*⟩ `;` `then` ⟨*list*⟩ `;` `elif` ⟨*list*⟩ `then` ⟨*list*⟩ `;` `else` ⟨*list*⟩ `;` `fi`
- `while` ⟨*list*⟩ `;` `do` ⟨*list*⟩ `;` `done`
  `until` ⟨*list*⟩ `;` `do` ⟨*list*⟩ `;` `done`

Any of the above semicolons can instead also be a line feed
(no \ required there before the line feed to continue the command on the next line)

Shell compound commands (cont'd)

- `case` ⟨*word*⟩ `in`
  ⟨*pattern*⟩`|`⟨*pattern*⟩`|...)`
  ⟨*list*⟩ `;;`
  `...`
  `esac`

  Matches expanded ⟨*word*⟩ against each ⟨*pattern*⟩ in turn (same matching rules as pathname expansion) and executes the corresponding ⟨*list*⟩ when first match is found. Example:

```
case "$command" in
  start)
    app_server &
    processid=$! ;;
  stop)
    kill $processid ;;
  *)
    echo 'unknown command' ;;
esac
```

Boolean tests

The first ⟨*list*⟩ in the `if`, `while` and `until` commands is interpreted as a Boolean condition. The `true` and `false` commands return 0 and 1 respectively (note the inverse logic compared to Boolean values in C!).

The builtin command "`test` ⟨*expr*⟩", which can also be written as "`[` ⟨*expr*⟩ `]`" evaluates simple Boolean expressions on files, such as

`-e` ⟨*file*⟩	is true if ⟨*file*⟩ exists
`-d` ⟨*file*⟩	is true if ⟨*file*⟩ exists and is a directory
`-f` ⟨*file*⟩	is true if ⟨*file*⟩ exists and is a normal file
`-r` ⟨*file*⟩	is true if ⟨*file*⟩ exists and is readable
`-w` ⟨*file*⟩	is true if ⟨*file*⟩ exists and is writable
`-x` ⟨*file*⟩	is true if ⟨*file*⟩ exists and is executable

or strings, such as

⟨*string1*⟩ `==` ⟨*string2*⟩ ⟨*string1*⟩ `<` ⟨*string2*⟩
⟨*string1*⟩ `!=` ⟨*string2*⟩ ⟨*string1*⟩ `>` ⟨*string2*⟩

Boolean tests – examples

```
if [ -e $HOME/.rhosts ] ; then
  echo 'Found ~/.rhosts!' | \
  mail $LOGNAME -s 'Hacker backdoor?'
fi
```

Note: A backslash at the end of a command line causes end-of-line to be ignored.

```
if [ "`hostname`" == python.cl.cam.ac.uk ] ; then
   ( sleep 10 ; play ~/sounds/greeting.wav ) &
else
   xmessage 'Good Morning, Dave!' &
fi
[ "`arch`" != ix86 ] || { clear ; echo "I'm a PC" ; }
```

Aliases and functions

Aliases allow a string to be substituted for the first word of a command:

```
$ alias dir='ls -la'
$ dir
```

Shell functions are defined with "⟨*name*⟩ `()` `{` ⟨*list*⟩ `;` `}`". In the function body, the command-line arguments are available as `$1`, `$2`, `$3`, etc. The variable `$*` contains all arguments and `$#` their number.

```
$ unalias dir
$ dir () { ls -la $* ; }
```

Outside the body of a function definition, the variables `$*`, `$#`, `$1`, `$2`, `$3`, ... can be used to access the command-line arguments passed to a shell script.

Shell history

The shell records commands entered. These can be accessed in various ways to save keystrokes:

- "`history`" outputs all recently entered commands.
- "`!`$\langle n \rangle$" is substituted by the $\langle n \rangle$-th history entry.
- "`!!`" and "`!-1`" are equivalent to the previous command.
- "`!*`" is the previous command line minus the first word.
- Use cursor up/down keys to access history list, modify a previous command and reissue it by pressing Return.
- Type Ctrl-O instead of Return to issue command from history and edit its successor, which allows convenient repetition of entire command sequences.
- Type Ctrl-R to search string in history.

Most others probably only useful for teletype writers without cursor.

Readline

Interactive bash reads commands via the `readline` line-editor library. Many Emacs-like control key sequences are supported, such as:

- Ctrl-A/Ctrl-E moves cursor to start/end of line
- Ctrl-K deletes (kills) the rest of the line
- Ctrl-D deletes the character under the cursor
- Ctrl-W deletes a word (first letter to cursor)
- Ctrl-Y inserts deleted strings
- ESC ^ performs history expansion on current line
- ESC # turns current line into a comment

Automatic word completion: Type the "Tab" key, and bash will complete the word you started when it is an existing $variable, ~user, hostname, command or filename, depending on the context. If there is an ambiguity, pressing "Tab" a second time will show list of choices.

Startup files for terminal access

When you log in via a terminal line or telnet/rlogin/ssh:

- After verifying your password, the `login` command checks `/etc/passwd` to find out what shell to start for you.
- As a login shell, bash will execute the scripts

  ```
  /etc/profile
  ~/.profile
  ```

  The second one is where you can define your own environment. Use it to set exported variable values and trigger any activity that you want to happen at each login.
- Any subsequently started bash will read `~/.bashrc` instead, which is where you can define functions and aliases, which – unlike environment variables – are not exported to subshells.

Desktop environment startup

Login process on the console of a desktop computer:

- The "X server" provides access to display, keyboard and mouse for "X client" applications via the "X11 protocol".
- After boot, the `/sbin/init` system starts the "X display manager" (e.g., `xdm`, `gdm`, `lightdm`, `ssdm`), which starts the X server.
- The xdm then acts as the first X client and provides the login screen.
- After successful login, the xdm starts a shell script Xsession (as the user).
- Xsession then starts all the components of the desktop environment. This includes in particular a "window manager", often also a separate session and/or desktop manager, and several auxiliary servers for inter-application communication (dbus, ssh-agent, etc.). The window, session or desktop managers finally allow the user to start application processes (`xterm`, etc.).

Desktop environment (cont'd)

Logout:

- Most X clients are started as background processes. An exception is the session or window manager, which Xsession starts as the last foreground process.
- If the session or window manager terminates, the Xsession script ends as well. This causes the xdm to reset the X server, which closes the connection to and thereby terminates all the X clients.
- Finally, xdm presents the login screen again.

The detailed behaviour of Xsession and the session manager differs significantly across desktop environments and Unix/Linux distributions, and usually offer many hooks and options for configuration (auto-start of applications).

X servers also manage a database of keyboard mappings and client configuration options (see `man xmodmap`, `man xrdb`).

`man X`, `man xdm`

sed – a stream editor

Designed to modify files in one pass and particularly suited for doing automated on-the-fly edits of text in pipes. sed scripts can be provided on the command line

```
sed [-e] 'command' files
```

or in a separate file

```
sed -f scriptfile files
```

General form of a sed command:

```
[address[,address]][!]command[arguments]
```

Addresses can be line numbers or regular expressions. Last line is "$". One address selects a line, two addresses a line range (specifying start and end line). All commands are applied in sequence to each line. After this, the line is printed, unless option -n is used, in which case only the p command will print a line. The ! negates address match. {...} can group commands per address.

sed – regular expressions

Regular expressions enclosed in /.../. Some regular expression meta characters:

- "." matches any character (except new-line)
- "*" matches the preceding item zero or more times
- "+" matches the preceding item one or more times
- "?" matches the preceding item optionally (0–1 times)
- "^" matches start of line
- "$" matches end of line
- "[...]" matches one of listed characters
  (use in character list "^" to negate and "–" for ranges)
- "\(...\)" grouping, "\{n,m\}" match $n, \ldots, m$ times
- "\" escape following meta character

sed – some examples

Substitute all occurrences of "Windows" with "Linux" (command: s = substitute, option: g = "global" = all occurrences in line):

```
sed 's/Windows/Linux/g'
```

Delete all lines that do not end with "OK" (command: d = delete):

```
sed '/OK$/!d'
```

Print only lines between those starting with BEGIN and END, inclusive:

```
sed -n '/^BEGIN/,/^END/p'
```

Substitute in lines 40–60 the first word starting with a capital letter with "X":

```
sed '40,60s/[A-Z][a-zA-Z]*/X/'
```

grep, head, tail, sort, cut, wc, aspell

- Print only lines that contain pattern:

  `grep` *`pattern files`*

  Option `-v` negates match and `-i` makes match case insensitive.

- Print the first and the last 25 lines of a file:

```
head -n 25 file
tail -n 25 file
```

  `tail -f` outputs growing file.

- Print the lines of a text file in alphabetical order: `sort file`
  Options: `-k` select column, `-t` field separator, `-n` sort numbers, `-u` eliminate duplicate lines, `-r` reverse order, `-R` random order.

- Split input into colon-separated fields and show only fields 1 and 3:

```
getent passwd | cut -d: -f1,3
```

- Count words, lines or characters: `wc`, `wc -l`, `wc -c`

- Print only words from stdin that fail spell test: `aspell list`

```
$ aspell list --mode=tex <dissertation.tex | LC_COLLATE=C sort -u | less
```

`chmod` – set file permissions

- Unix file permissions: $3 \times 3 + 2 + 1 = 12$ bit information.
- Read/write/execute right for user/group/other.
- $+$ set-user-id and set-group-id (elevated execution rights)
- $+$ "sticky bit" (only owner can delete file from directory)
- `chmod ugoa[+-=]rwxst files`

Examples: Make file unreadable for anyone but the user/owner.

```
$ ls -l message.txt
-rw-r--r--    1 mgk25    private      1527 Oct  8 01:05 message.txt
$ chmod go-rwx message.txt
$ ls -l message.txt
-rw-------    1 mgk25    private      1527 Oct  8 01:05 message.txt
```

For directories, "execution" right means right to traverse. Directories can be made traversable without being readable, such that only those who know the filenames inside can access them.

`find` – traverse directory trees

`find directories expression` — recursively traverse the file trees rooted at the listed directories. Evaluate the Boolean expression for each file found. Examples:

Print relative pathname of each file below current directory:

```
$ find . -print
```

Erase each file named "`core`" below home directory if it was not modified in the last 10 days:

```
$ find ~ -name core -mtime +10 -exec rm -i {} \;
```

The test "`-mtime +10`" is true for files older than 10 days, concatenation of tests means "logical and", so "`-exec`" will only be executed if all earlier terms were true. The "`{}`" is substituted with the current filename, and "`\;`" terminates the list of arguments of the shell command provided to "`-exec`".

`tar` – converting file trees into byte streams (and back)

Originally: “tape archiver”

Create archive (recurses into subdirectories):

```
$ tar cvf archive.tar files
```

Show archive content:

```
$ tar tvf archive.tar
```

Extract archive:

```
$ tar xvf archive.tar [files]
```

gzip & friends – compressing byte streams

- `gzip` *`file`* — convert "*`file`*" into a compressed "*`file`*`.gz`" (using a Lempel-Ziv/Huffman algorithm).
- `gunzip` *`file`* — decompress "`*.gz`" files.
- `[un]compress` *`file`* — [de]compress "`*.Z`" files (older tool using less efficient and patented LZW algorithm).
- `b[un]zip2` *`file`* — [de]compress "`*.bz2`" files (newer tool using Burrows-Wheeler blocktransform).
- `zcat` `[`*`file`*`]` — decompress `*.Z`/`*.gz` to stdout for use in pipes.
- Extract compressed `tar` archive

```
$ zcat archive.tar.gz | tar xvf -
$ tar xvzf archive.tgz       # GNU tar only!
```

Some networking tools

- `curl` *`url`* — Transfer a URL via HTTP, FTP, IMAP, etc.

  Defaults to stdout, specify destination filename with `-o` *`filename`* or use -O (remote name).

- `wget` *`url`* — Download files over the Internet via HTTP or FTP.

  Option "`-r`" fetches HTML files recursively, option "`-l`" limits recursion depth.

- `lynx [-dump]` *`url`* — text-mode web browser

- `ssh [`*`user`*`@]`*`hostname`* `[`*`command`*`]` — Log in via compressed and encrypted link to remote machine. If "*`command`*" is provided, execute it in remote shell, otherwise go interactive.

  Preserves stdout/stderr distinction. Can also forward X11 requests (option "`-X`") or arbitrary TCP/IP ports (options "`-L`" and "`-R`") over secure link.

- `ssh-keygen -t rsa` — Generate RSA public/private key pair for password-free ssh authentication in "`~/.ssh/id_rsa.pub`" and "`~/.ssh/id_rsa`". Protect "`id_rsa`" like a password!

  Remote machine will not ask for password with ssh, if your private key "`~/.ssh/id_rsa`" fits one of the public keys ("locks") listed on the remote machine in "`~/.ssh/authorized_keys`".

  On MCS Linux, your Windows-server home directory with `~/.ssh/authorized_keys` is mounted only **after** login, and therefore no password-free login for first session.

rsync

`rsync [`*`options`*`] `*`source destination`* — An improved `cp`.

- The source and/or destination file/directory names can be prefixed with `[`*`user`*`@]`*`hostname`*`:` if they are on a remote host.
- Uses `ssh` as a secure transport channel (may require `-e ssh`).
- Options to copy recursively entire subtrees (`-r`), preserve symbolic links (`-l`), permission bits (`-p`), and timestamps (`-t`).
- Will not transfer files (or parts of files) that are already present at the destination. An efficient algorithm determines, which bytes actually need to be transmitted only ⇒ very useful to keep huge file trees synchronised over slow links.

Application example: Very careful backup

```
rsync -e ssh -v -rlpt --delete --backup \
  --backup-dir OLD/`date -Im` \
  me@myhost.org:. mycopy/
```

Removes files at the destination that are no longer at the source, but keeps a timestamped copy of each changed or removed file in `mycopy/OLD/`*`yyyy-mm-dd...`*`/`, so nothing gets ever lost.

`diff`, `patch` – managing file differences

- `diff` *`oldfile newfile`* — Show difference between two text files, as lines that have to be inserted/deleted to change "*`oldfile`*" into "*`newfile`*". Option "`-u`" gives better readable "unified" format with context lines. Option "`-r`" compares entire directory trees.

```
$ diff -u example.bak example.txt
--- example.bak
+++ example.txt
@@ -1,2 +1,3 @@
 an unmodified line
-this sentence no verb
+this sentence has a verb
+a newly added line
```

```
$ diff example.bak example.txt
2c2,3
< this sentence no verb
---
> this sentence has a verb
> a newly added line
```

- `patch <`*`diff-file`* — Apply the changes listed in the provided `diff` output file to the old files named in it. The diff file should contain relative pathnames. If not, use option "`-p`n" to strip the first n directory names from pathnames in "*`diff-file`*".

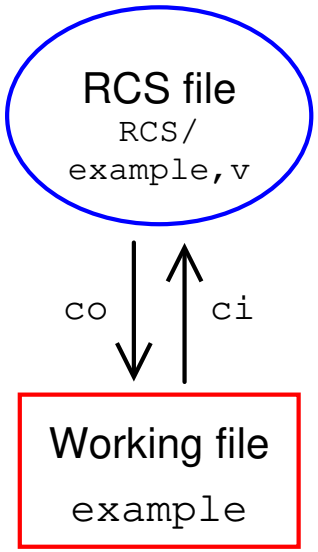
  If the old files found by `patch` do not match exactly the removed lines in a "`-u`" diff output, `patch` will search whether the context lines can be located nearby and will report which line offset was necessary to apply them.

- `diff3` *`myfile oldfile yourfile`* — Compare three files and merge the edits from different revision branches.

RCS – Revision Control System

Operates on individual files only. For every working file "`example`", an associated RCS file "`example,v`" keeps a revision history database.

RCS files can be kept next to the working files, or in a subdirectory `RCS/`.



- `ci example` — Move a file (back) into the RCS file as the new latest revision ("check in").
- `ci -u example` — Keep a read-only unlocked copy.
  "`ci -l ...`" is equivalent to "`ci ...`" followed by "`co ...`".
- `ci -l example` — Keep a writable locked copy (only one user can have the lock for a file at a time).
  "`ci -l ...`" is equivalent to "`ci ...`" followed by "`co -l ...`".
- `co example` — Fetches the latest revision from "`example,v`" as a read-only file ("check out"). Use option "`-r`$n.m$" to retrieve earlier revisions. There must not be a writable working file already.
- `co -l example` — Fetches the latest revision as a locked writable file if the lock is available.

RCS – Revision Control System (cont'd)

- `rcsdiff example` — Show differences between working file and latest version in repository (use option "`-r`$n.m$" to compare older revisions). Normal `diff` options like `-u` can be applied.
- `rlog example` — Show who made changes when on this file and left what change comments.

If you want to use RCS in a team, keep all the "`*,v`" files in a shared repository directory writable for everyone. Team members have their own respective working directory with a symbolic link named `RCS` to the shared directory.

As long as nobody touches the "`*,v`" files or manually changes the write permissions on working files, only one team member at a time can edit a file and old versions are never lost. The `rcs` command can be used by a team leader to bypass this policy and break locks or delete old revisions.

RCS remains useful for quickly maintaining history of single files, as an alternative to manually making backup copies of such files.

RCS is no longer commonly used for joint software development.

If you work in a distributed team on a project with subdirectories, need remote access, want to rename files easily, or simply hate locks, use `svn` or `git` instead.

`svn` – Subversion

Subversion is a popular centralized version control system (2001).

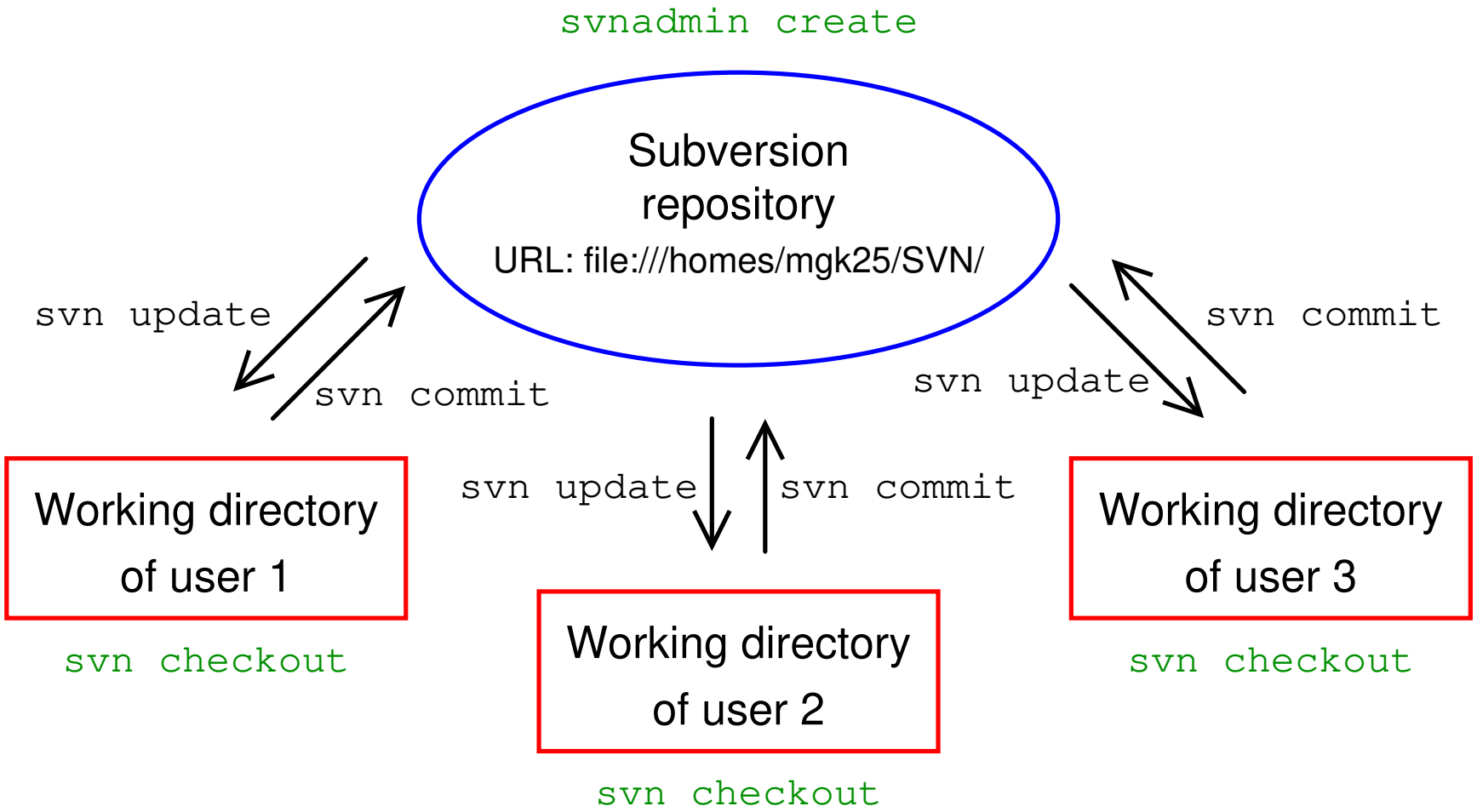
Main advantages over RCS:

- Supports a copy-modify-merge workflow (RCS: lock-modify-unlock). This allows team members to edit the same files **concurrently**.
  - Concurrent edits in different lines
    $\Longrightarrow$ merged automatically
  - Concurrent edits in the same lines
    $\Longrightarrow$ requires manual resolution of conflicts
- Manages entire directory trees, not just single files
- Understand tree edits (file copy, delete, rename, move)
- Supports several remote-access protocols (WebDAV, ssh, etc.)

Full documentation: `http://svnbook.red-bean.com/` and `http://subversion.apache.org/`

Microsoft Windows Subversion GUI: `http://tortoisesvn.tigris.org/`

svn – repository and working directories



Team administrator first creates repository: `svnadmin create`
Team members create personal working directories: `svn checkout`
Team members repeatedly fetch latest version: `svn update`
and return their changes: `svn commit`

svn – Subversion vs CVS

Subversion was specifically written to replace an older system, CVS, which in turn started out as a layer on top of RCS for managing entire directory trees. Its command-line interface closely follows that of CVS, but improves and simplifies the latter in many ways. In particular, Subversion

- understands renaming, moving, copying and replacing of both files and entire directory trees, no per-file version numbers
- understands symbolic links
- performs atomic commits
- versioned metadata (MIME types, EOL semantics, etc.)
- is easy to learn and understand for current CVS users
- simpler branching and tagging (through efficient copying)
- more efficient transactions, more disconnected operations
- wider choice of remote-access protocols (WebDAV, ssh, etc.)

Old RCS/CVS repositories can easily be converted: `http://cvs2svn.tigris.org/`

svn – setting up

Create new repository (e.g. `~/SVN/`), separate from working directory:

```
svnadmin create ~/SVN
```

Then checkout a working copy of the repository into a new working directory (`~/wdir`), referring to the repository via its URL:

```
svn checkout file://${HOME}/SVN ~/wdir
```

Note that your new working directory has a hidden `.svn/` subdirectory. This contains, among other things, the URL of your repository (see `svn info`). Therefore, inside the working directory, it is no longer necessary to add that repository URL as an argument to `svn` operations.

Now populate the repository with content:

- Create or move into the working directory some files
- Register them with Subversion (`svn add`)
- Push them into the repository (`svn commit`)

Then every team member, after their own initial `svn checkout`, does:

- Pull the latest version (`svn update`)
- Make edits
- Push them back to the repository (`svn commit`)

`svn` – directory edits

- `svn add` *`filenames`* — Put new files/folders under version control

  Warning: adding a directory adds all content as well, unless you use `svn add -N` *`dirnames`*.
- `svn delete` *`filenames`* — Delete files/folders
- `svn copy` *`source destination`* — Copy files/folders
- `svn move` *`source destination`* — Move files/folders

The above four operations will not transfer the requested changes to the repository before the next `commit`, however the `delete`/`copy`/`move` operations perform the requested action immediately on your working files.

Remember not to use `rm`/`cp`/`mv` on working files that are under Subversion control, otherwise these operations will not be reflected in the repository after your next commit.

If you delete a version-controlled file with `rm`, the next `svn update` will restore it.

`svn` – querying the working-directory status

- `svn status` – List all files that differ between your working directory and the repository. The status code shown indicates:
  - A=added: this file will appear in the repository
  - D=deleted: this file will disappear from the repository
  - M=modified: you have edited this file
  - R=replaced: you used `svn delete` followed by `svn add`
  - C=conflict: at the last update, there was a conflict between your local changes and independent changes in the repository, which you still need to resolve manually
  - ?=unversioned: file is not in repository (suppress: `-q`)
  - !=missing: file in repository, but not in working dir.
- `svn diff [`*`filenames`*`]` —
  Show what you changed so far compared to the "base" version that you got at your last checkout or update.
- `svn info` — Show metadata about the working directory (revision of last update, URL of repository, etc.)

svn – commits and updates

- `svn commit [`*`filenames`*`]` — Check into the repository any modifications, additions, removals of files that you did since your last checkout or commit.

  Option `-m '...'` provides a commit log message; without it, `svn commit` will call `$EDITOR` for you to enter one.

- `svn update [`*`filenames`*`]` — Apply modifications that others committed since you last updated your working directory.

  This will list in the first column a letter for any file that differed between your working directory and the repository. Apart from the letter codes used by `status`, it also may indicate

  - U=updated: get newer version of this file from repository
  - G=merged: conflict, but was automatically resolved

  Remaining conflicts (indicated as C) must be merged manually.

  To assist in manual merging of conflicts, the `update` operation will write out all three file versions involved, all identified with appropriate filename extensions, as well as a `diff3`-style file that shows the differing lines next to each other for convenient editing.

svn – some more commands

- `svn resolved` *filenames* — Tell Subversion you have resolved a conflict. (Also cleans up the three additional files.)
- `svn revert` *filenames* — Undo local edits and go back to the version you had at your last checkout, commit, or update.
- `svn ls [`*filenames*`]` — List repository directory entries
- `svn cat` *filenames* — Output file contents from repository
  Use "`svn cat` *filenames*`@`*rev*" to retrieve older revision *rev*.

Some of these commands can also be applied directly to a repository, without needing a working directory. In this case, specify a repository URL instead of a filename:

```
svn copy file://${HOME}/SVN/trunk \
         file://${HOME}/SVN/tags/release-1.0
```

An `svn copy` increases the repository size by only a trivial amount, independent of how much data was copied. Therefore, to give a particular version a symbolic name, simply `svn copy` it in the repository into a new subdirectory of that name.

svn – a working example (User 1)

User 1 creates a repository, opens a working directory, and adds a file:

```
$ svnadmin create $HOME/example-svn-repo
$ svn checkout file://$HOME/example-svn-repo wd1
Checked out revision 0.
$ cd wd1
wd1$ ls -AF
.svn/
wd1$ svn status
wd1$ echo 'hello world' >file1
wd1$ svn status
?       file1
wd1$ svn add file1
A         file1
wd1$ svn commit -m 'adding my first file'
Adding         file1
Transmitting file data .
Committed revision 1.
wd1$ svn status
```

svn – a working example (User 2)

User 2 checks out own working directory and start working on it:

```
$ svn checkout file://$HOME/example-svn-repo wd2
A  wd2/file1
Checked out revision 1.
$ cd wd2
wd2$ ls -AF
.svn/  file1
wd2$ svn status
wd2$ echo 'hello dogs' >file1
wd2$ svn status
M      file1
```

svn – a working example (User 1)

Meanwhile, User 1 also adds and changes files, and commits these:

```
wd1$ ls -AF
.svn/  file1
wd1$ cat file1
hello world
wd1$ echo 'bla' >file2
wd1$ svn add file2
A         file2
wd1$ echo 'hello humans' >file1
wd1$ svn status
M       file1
A       file2
wd1$ svn commit -m 'world -> humans'
Sending        file1
Adding         file2
Transmitting file data ..
Committed revision 2.
wd1$ svn status
```

svn – a working example (User 2)

User 1 managed to commit her changes first, so User 2 will be notified of a concurrent-editing conflict, and will have to merge the two changes:

```
wd2$ svn commit -m 'world -> dogs'
Sending        file1
svn: E155011: Commit failed (details follow):
svn: E160028: File '/file1' is out of date
wd2$ svn update
Updating '.':
C    file1
A    file2
Updated to revision 2.
Conflict discovered in 'file1'.
Select: (p) postpone, (df) show diff, (e) edit file, (m) merge,
        (mc) my side of conflict, (tc) their side of conflict,
        (s) show all options: p
Summary of conflicts:
  Text conflicts: 1
```

svn – a working example (User 2)

```
wd2$ cat file1
<<<<<<< .mine
hello dogs
=======
hello humans
>>>>>>> .r2
wd2$ svn status
?       file1.mine
?       file1.r1
?       file1.r2
C       file1
wd2$ echo 'hello humans and dogs' >file1
wd2$ svn resolved file1
Resolved conflicted state of 'file1'
wd2$ svn status
M       file1
wd2$ svn commit -m 'k9 extension'
Sending        file1
Transmitting file data .
Committed revision 3.
```

svn – a working example (User 1)

```
wd1$ svn update
Updating '.':
U    file1
Updated to revision 3.
wd2$ cat file1
hello humans and dogs
wd1$ svn diff -c3
Index: file1
===================================================================
--- file1 (revision 2)
+++ file1 (revision 3)
@@ -1 +1 @@
-hello humans
+hello humans and dogs
```

svn – remote access

The URL to an svn repository can point to a

- local file — `file://`
- Subversion/WebDAV Apache server — `http://` or `https://`
- Subversion server — `svn://`
- Subversion accessed via ssh tunnel — `svn+ssh://`

Setting up an `svn://` or `http://` server may be complicated in some environments (administrator account required, firewall rules, etc.).

An `svn+ssh://` server is much easier to set up and only needs

- an existing SSH server
- your public key in `~/.ssh/authorized_keys`
- `/usr/bin/svnserve`

A command like

```
svn list svn+ssh://mgk25@linux/home/mgk25/SVN
```

will run "`ssh mgk25@linux svnserve -t`" to start the remote server, and will then talk to it using the Subversion protocol over stdin/stdout.

svn – sharing an svn+ssh:// server securely

The ease of setting up your own svn+ssh:// server makes it tempting to share that access with other team members, to grant them access to your repository.

However: simply granting others access to your ssh login, by adding their public keys to your ~/.ssh/authorized_keys file, would also give them full shell access to your account!

Fortunately, OpenSSH allows you to restrict what access someone else's public key in your ~/.ssh/authorized_keys file grants. In particular, the command="..." option allows you to constrain the owner of that key to only execute a single program running under your user identity:

```
command="svnserve -t --tunnel-user=gsm10 -r /home/mgk25/SVN",
no-port-forwarding,no-agent-forwarding,no-X11-forwarding,
no-pty ssh-rsa AAAB3...ogUc= gsm10@cam.ac.uk
```

Adding gsm10's ssh public key with the above prefixes to my own ~/.ssh/authorized_keys file (all in one line!) will only allow him to start up and communicate with an svnserve -t command, which is further constrained to access only repositories under /home/mgk25/SVN, and to log all his commits under his user identity "gsm10". (The other options disable various network port-forwarding facilities of ssh that are not required here. See man sshd and the svn book for details.)

Consult your system administrator before using this technique.

Distributed revision control systems

Popular examples: git, mercurial, bazaar

- No single central repository – more reliable and "democratic"
- Each participant holds a local repository of the revision history
- Committing and uploading a commit to another site are separate actions, i.e. commits, branches, etc. can be done off-line
- Creating and merging branches are quick and easy operations
- Branches do not have to be made visible to other developers
- Revisions identified by secure hash value rather than integer

Distributed version control systems are far more flexible and powerful than centralized ones (like Subversion), but require more experience.

`git` – a distributed revision control system

- Created by Linus Torvalds in 2005 to manage Linux kernel
- Now the most popular distributed revision control system
- Used by a large number of open-source projects
- Integrated in many free and commercial development environments
- Supported by major web source-code hosting sites (github, etc.)
- Can interface with Subversion repositories
- Git archives snapshots ("commits") of a directory tree, rather than trying to record edit operations (e.g. renaming files)

git – branching and merging

Commits form a directed acyclic graph in the repository.

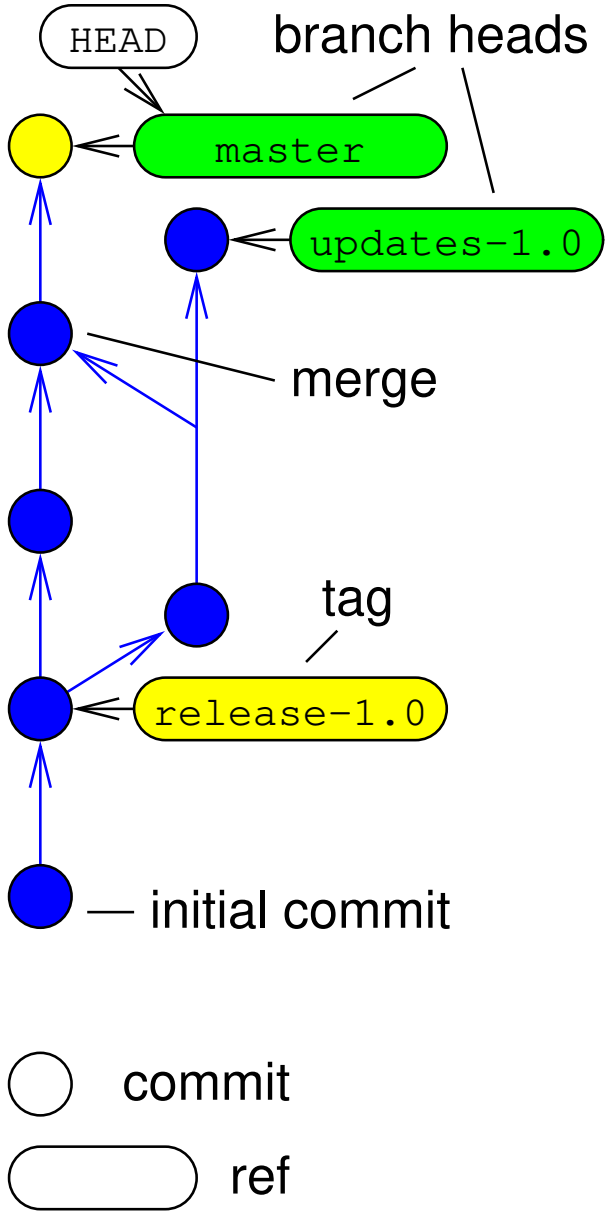
Each commit can be the parent of multiple other commits (branches) and each commit can have multiple other commits as parents (merge).

Branches reflect diverging lines of development, merges bring these together again.

Tags add human-friendly names to some commits.

Branch heads move along with each commit to them, tag references remain where they are.

Commits not reachable from any tag or branch head will be garbage collected.



`git` – objects in repository

commit A saved snapshot of a project (= tree of files and directories), stored in a repository. Includes metadata:

- pointer(s) to parent commit(s)
- date and author of the commit
- description

Identified through a 160-bit string (40 hex digits), the SHA-1 hash of the file tree and metadata. It represents a version of your software, the state of your project's files at some point in time.

tag Descriptive name (e.g., `release-2.4.7`) pointing to a particular commit. Can be associated with commits after their creation. Tags are intended to keep pointing at the same commit, but can be edited later.

branch A line of development. Another descriptive name with a pointer to a commit (branch head), but (unlike a tag) that pointer is updated automatically whenever a new commit arrives on that branch. Name of default branch: `master`

man gitglossary

`git` – basic usage (your own project)

- `git init` — Start using `git` by executing this once in the root of your project working directory. This creates the hidden subdirectory `.git` which now contains your repository, your staging area ("index"), and a configuration file `.git/config`.
- `git add` *`file ...`* — Prepare creating the first snapshot ("commit") of your project by copying ("adding") your files to the staging area. (Specifying a directory will recursively add all files and directories in it, e.g. "`git add .`" imports everything.)
- `git status` — List which files are already in the staging area.

  Option `-s` results in a compacter output format
- `git commit` — Create the first snapshot of your project by saving whatever is in the staging area as your first commit in the repository.

  `git commit` will fire up `$EDITOR` for you, to enter a commit message. This can also be provided via command-line option `-m 'commit message'`.

git – getting started

```
$ mkdir wd1
$ cd wd1
wd1$ git init
Initialized empty Git repository in [...]/wd1/.git/
wd1$ ls -AF
.git/
wd1$ echo 'hello world' >file1
wd1$ git status -s
?? file1                              (untracked file)
wd1$ git add file1
wd1$ git status -s
A  file1                              (changes to be committed)
wd1$ git commit -m 'adding my first file'
[master (root-commit) 5f3b6b2] adding my first file
 1 file changed, 1 insertion(+)
 create mode 100644 file1
```

Each commit is automatically named using a 20 bytes (= 40 hex digits) pseudo-random byte string, often abbreviated into a unique prefix ("5f3b6b2").

`HEAD` can be used as a synonym for the latest commit.

git – your first commit

Setting your name

Git records your name and email address with each commit. If you have never used git before, "`git commit`" will invite you to first set those details once in `~/.gitconfig`, using

```
$ git config --global user.name "Your Name Comes Here"
$ git config --global user.email you@yourdomain.example.com
```

- `git log` — Show the list of commits you have made so far. Option `-p` (patch) shows changes made to each file
- `gitk` — GUI tool for browsing commits
- `git tag` *tagname commit* — Associate a human-readable name ("tag") with a commit

```
wd1$ git log
commit 5f3b6b271a048135615805fd705adf2a89225611
Author: Markus Kuhn <Markus.Kuhn@cl.cam.ac.uk>
Date:   Tue Nov 25 10:11:09 2014 +0000
    adding my first file
```

git – adding more commits, tagging

```
wd1$ echo 'have a nice day' >>file1
wd1$ git add file1
wd1$ git commit -m 'second line added'
[master a324cdc] second line added
 1 file changed, 1 insertion(+)
wd1$ git tag release-1.0 HEAD
wd1$ git log --graph --decorate --oneline
* a324cdc (HEAD, tag: release-1.0, master) second line added
* 5f3b6b2 adding my first file
wd1$ sed -i s/world/humans/ file1
wd1$ git commit -a -m 'world -> humans'
[master 610b41a] world -> humans
 1 file changed, 1 insertion(+), 1 deletion(-)
wd1$ git log --graph --decorate --oneline
* 610b41a (HEAD, master) world -> humans
* a324cdc (tag: release-1.0) second line added
* 5f3b6b2 adding my first file
```

`git` – branching and merging

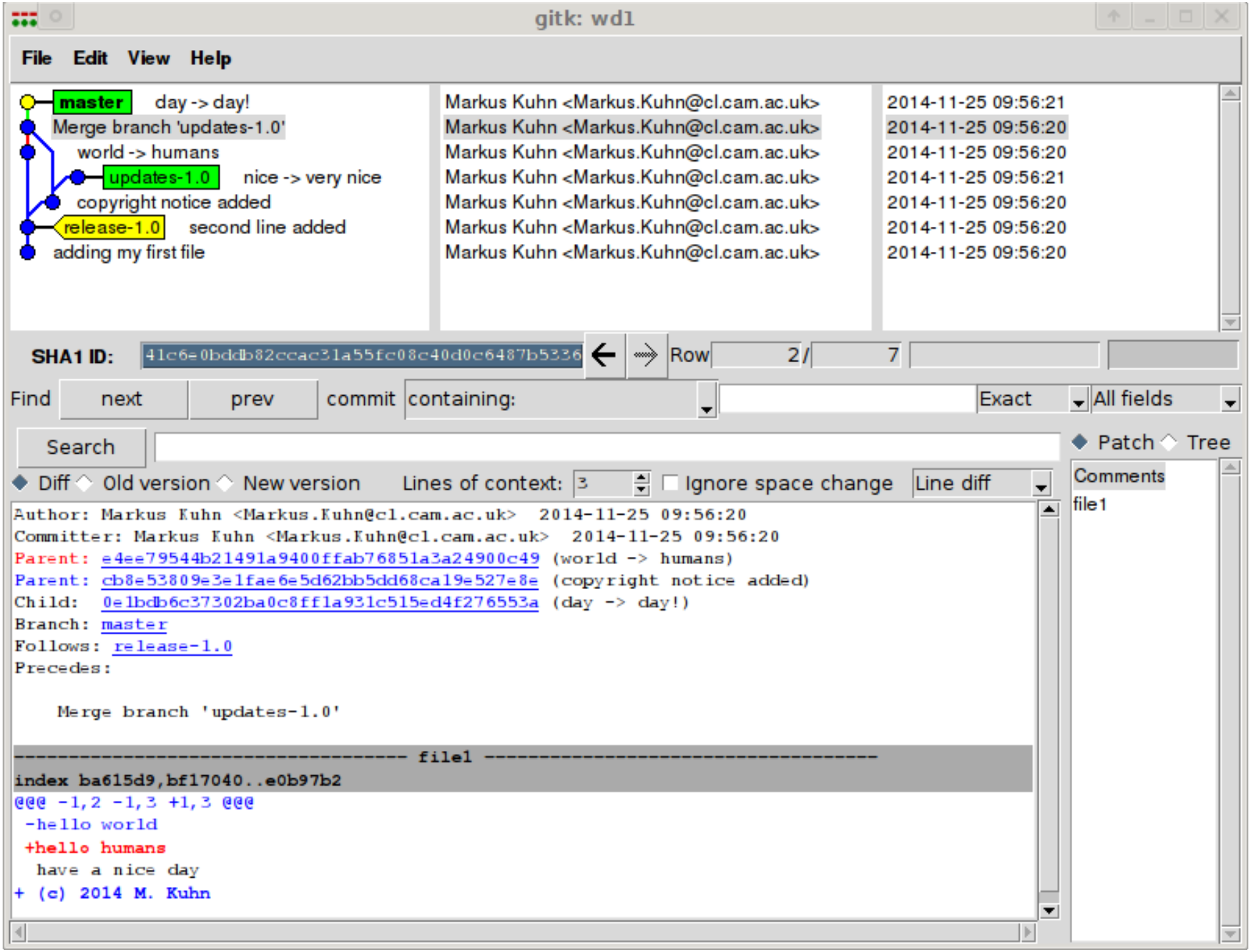
- `git branch` — list branches, mark current one (`HEAD`) with `*`.
- `git branch` *`branchname`* — start a new branch (off `HEAD`)
- `git branch` *`branchname startpoint`* — start a new branch, which diverges from the existing commit *startpoint*
- `git switch` *`branchname`* — switch the content of the working directory to branch head *branchname*, in preparation for working on that branch. Points `HEAD` at the new branch head and updates working files and index accordingly. Commits will now be added to that branch.
- `git switch -c` *`branchname`* `[`*`startpoint`*`]` — start a new branch head, then switch to it (= `git branch` + `switch`).

  In older git versions: `git checkout [-b]` *`branchname`*
- `git merge` *`commit`* `...` — Merge one or more other branches into the current one. Identifies all ancestors of the listed commit(s) (e.g., branch names) that are not yet ancestors of `HEAD`, and tries to apply their changes to `HEAD`, then make a new commit with all the listed commits as parents. May require manual resolving of conflicts.

git – branching example

```
wd1$ git branch
* master
wd1$ git branch updates-1.0 release-1.0
wd1$ git switch updates-1.0
Switched to branch 'updates-1.0'
wd1$ echo "(c) `date +%Y` M. Kuhn" >>file1
wd1$ git commit -a -m 'copyright notice added'
[updates-1.0 c1f332c] copyright notice added
 1 file changed, 1 insertion(+)
wd1$ git switch master
Switched to branch 'master'
wd1$ git merge updates-1.0
Auto-merging file1
Merge made by the 'recursive' strategy.
 file1 |    1 +
 1 file changed, 1 insertion(+)
```

```
wd1$ git switch updates-1.0
Switched to branch 'updates-1.0'
wd1$ sed -i 's/nice/very nice/' file1
wd1$ git commit -a -m 'nice -> very nice'
[updates-1.0 cb87e76] nice -> very nice
 1 file changed, 1 insertion(+), 1 deletion(-)
wd1$ git switch master
Switched to branch 'master'
wd1$ sed -i 's/day/day!/' file1
wd1$ git commit -a -m 'day -> day!'
[master ad469e5] day -> day!
 1 file changed, 1 insertion(+), 1 deletion(-)
wd1$ git log --graph --decorate --oneline --branches
* ad469e5 (HEAD, master) day -> day!
*   152b0d0 Merge branch 'updates-1.0'
|\
* | 610b41a world -> humans
| | * cb87e76 (updates-1.0) nice -> very nice
| |/
| * c1f332c copyright notice added
|/
* a324cdc (tag: release-1.0) second line added
* 5f3b6b2 adding my first file
```


gitk: wd1
File Edit View Help
master day -> day!
Merge branch 'updates-1.0'
world -> humans
updates-1.0 nice -> very nice
copyright notice added
release-1.0 second line added
adding my first file
Markus Kuhn <Markus.Kuhn@cl.cam.ac.uk>
2014-11-25 09:56:21
2014-11-25 09:56:20
SHA1 ID: 41c6e0bddb82ccac31a55fc08c40d0c6487b5336
Row 2/ 7
Find next prev commit containing: Exact All fields
Search
Patch Tree
Diff Old version New version Lines of context: 3 Ignore space change Line diff
Comments
file1
Author: Markus Kuhn <Markus.Kuhn@cl.cam.ac.uk> 2014-11-25 09:56:20
Committer: Markus Kuhn <Markus.Kuhn@cl.cam.ac.uk> 2014-11-25 09:56:20
Parent: e4ee79544b21491a9400ffab76851a3a24900c49 (world -> humans)
Parent: cb8e53809e3e1fae6e5d62bb5dd68ca19e527e8e (copyright notice added)
Child: 0e1bdb6c37302ba0c8ff1a931c515ed4f276553a (day -> day!)
Branch: master
Follows: release-1.0
Precedes:
Merge branch 'updates-1.0'
file1
index ba615d9,bf17040..e0b97b2
@@ -1,2 -1,3 +1,3 @@
-hello world
+hello humans
have a nice day
+ (c) 2014 M. Kuhn

`git` – storage areas

working dir Where you edit the files of your project.

index A staging area separate from the working directory (also located inside `.git/`) where `git add` transfers modified files, to prepare the next commit.

repository Database holding a directed acyclic graph of commits (vertices) and parent pointers (edges), along with "refs" (i.e., tags, branch heads, `HEAD`). Located in `.git/` subdirectory of the root of the working directory.

stash A temporary storage area (also located inside `.git/`) into which work-in-progress items from the working directory can be transfered, if the working directory is needed for other work.

`git` – more useful commands

- `git rm` *`filename ...`* — delete files from both working directory and index.
- `git add -p` *`filename`* — stage only some of the changes you made to *filename* in the index (if you do not want all your changes to go into the same commit).
- `git diff` — compare working directory and index
- `git diff --staged` — compare index and `HEAD`
- `git diff HEAD` — compare working directory and `HEAD`
- `git stash` — moves all current (usually unfinished) differences between `HEAD` and the working directory and index into a stash memory, then reset the working directory and index to `HEAD` (like `git reset --hard`); useful to work on something else for a while, in the middle of something
- `git stash pop` — retrieve stashed changes back into the working directory

`git` – fixing accidents

Fixing uncommitted mistakes in the working directory or index:

- `git reset` *`filename`* — restore *filename* in the index from `HEAD`, useful to undo "`git add|rm` *`filename`*".
- `git checkout` *`filename`* — restore *filename* in the working directory from `HEAD`, to undo uncommitted changes.

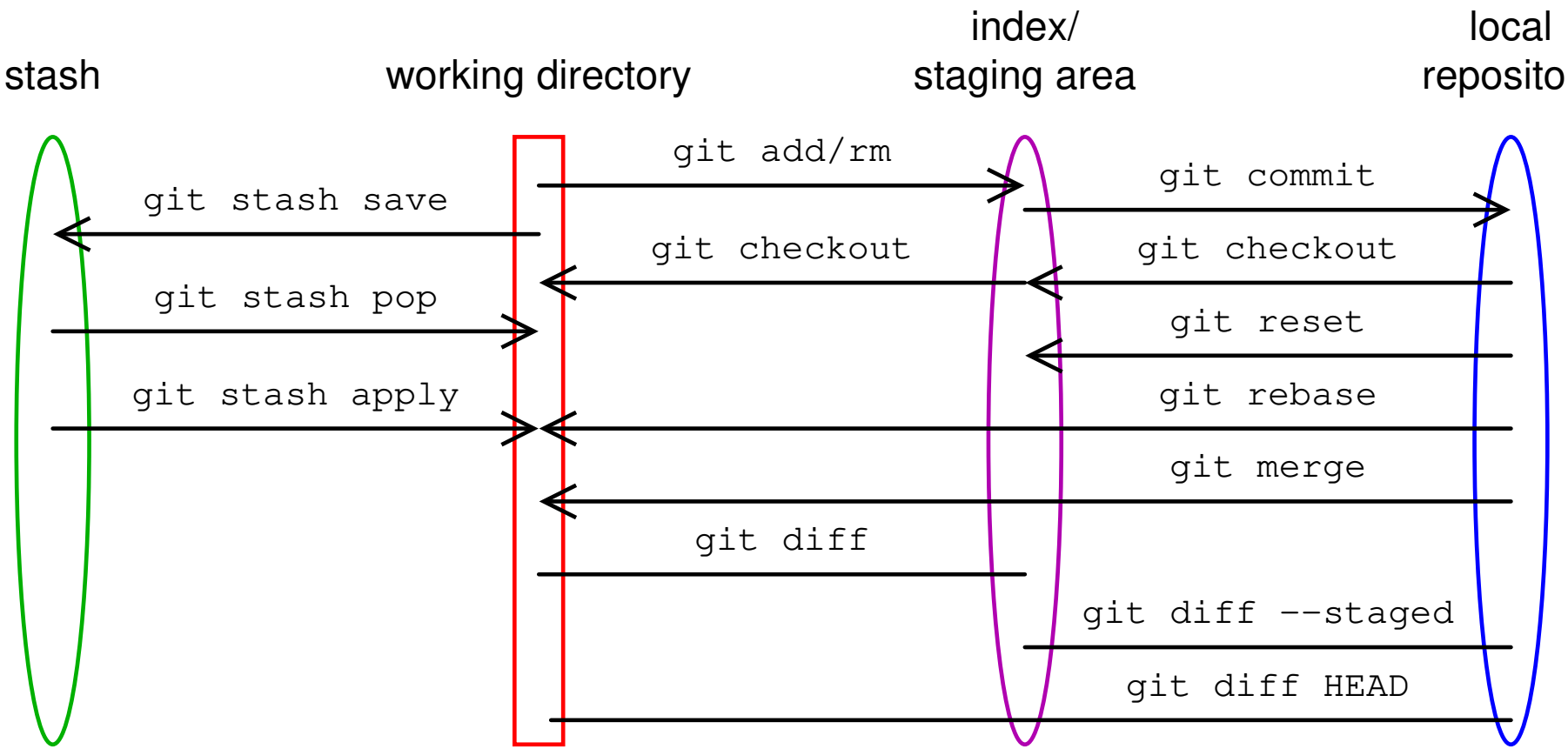
Fixing very recently committed mistakes (aka "rewriting history"):

- `git reset --soft HEAD^` — reset the current branch head to its parent (e.g., to immediately undo the last `git commit`).
- `git commit --amend` — replace the most recent commit to fix a mistake in the commit message.
- `git rebase` *`commit`* – rewite the commit history of the current branch to use *commit* as the new starting point

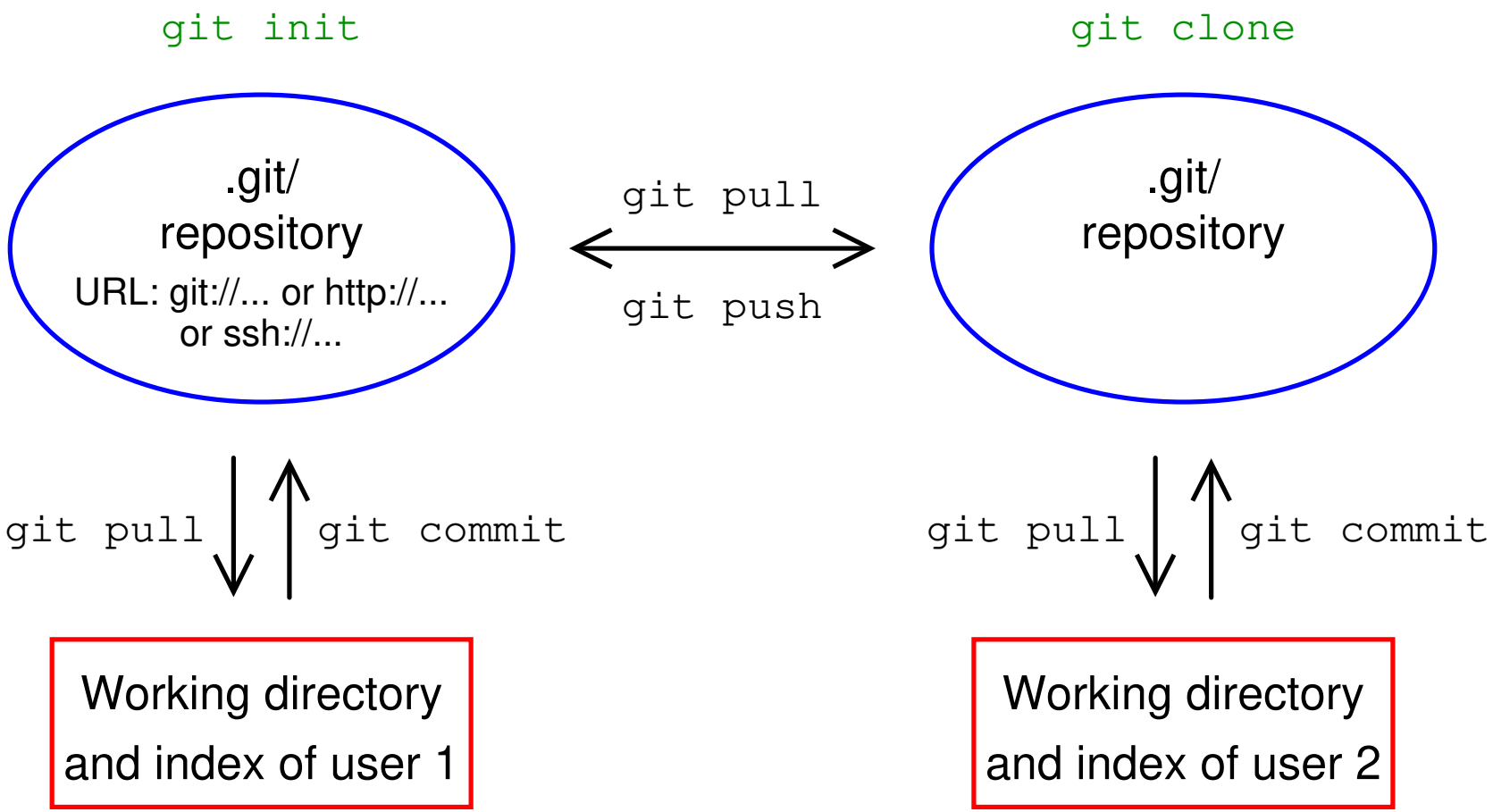
If collaborators might already have fetched your erroneous commit, it is usually far better to fix the mistake with a follow-up commit, than rewriting history. Otherwise, all collaborators working from your erroneous commit will have to rebase their branches to your fixed commit.

Read "man git-reset" carefully!

git – information flow between storage areas



git – multiple working directories, collaboration



Each working directory contains an associated repository, index and stash in a `.git/` subdirectory of its top-level directory.

`git` – basic collaboration commands

- `git clone` *`url`* – Copy an existing repository and create an associated working directory around it (unless `--bare` is used). A cloned repository will remember the URL of the `origin` repository where it came from.

  Supports remote repository access via SSH, e.g.
  `git clone ssh://mgk25@linux.cl.ds.cam.ac.uk/home/mgk25/wd1`

- `git init --bare` – Create a new "bare" repository, without associated a working directory (i.e., not located inside a `.git/` subdirectory of a working directory), for others to "push" into.
- `git pull` – Fetch updates from another repository (default: `origin`) and merge them into local working directory.
- `git fetch` – Fetch updates from remote branches from another repository (default: `origin`), but do not yet merge them into the local working directory.
- `git push` – Forward local revisions to another repository. Works only if the destination repository is bare or has no uncommitted changes in its working directory.
- `git remote -v` – list tracked other repositories (with URLs)

Version-control etiquette

- **Use diff before commit:** carefully review what you actually changed.
  This often uncovers editing accidents and left-over temporary changes never intended to go into the repository.
- **Provide a useful commit message:** a complete, meaningful, honest, and accurate summary of everything you changed.
  Don't write just "bug fixed" (which?) or "API changed" (how and why?).
  Under git, there is a format convention for commit messages: a one-line summary, followed by an empty line, followed by details. Include function names, bug numbers.
- **Commit unrelated changes separately.**
  Others may later want to undo or merge them separately.
- **Commit related changes together.**
  Do not forget associated changes to documentation, test cases, build scripts, etc.
- **Leave the repository in a usable and consistent state.**
  It should always compile without errors and pass tests.
- **Avoid dependent or binary files in the repository.**
  Diffs on binary files are usually incomprehensible. Compiled output should be easy to recreate. It just wastes repository space and others can never be sure what is in it.

git – repository read access via HTTP ("dumb" protocol)

Setting up a git server requires a continuously running Unix server.
But if you have already access to an HTTP server that serves files, e.g.

```
$ ssh shell.srcf.net
$ echo 'Hello world' >~/public_html/readme.txt
$ curl https://$USER.user.srcf.net/readme.txt
Hello world
```

you can also grant anonymous read access to a git repository that way.

Create or clone a bare git repo in `public_html`:

```
$ git clone --bare wd1 ~/public_html/project.git
$ mv ~/public_html/project.git/hooks/post-update{.sample,}
$ ( cd ~/public_html/project.git && git update-server-info )
```

Renaming in the `hooks/` directory the shell script `post-update.sample` into `post-update` causes the command `git update-server-info` to be run after each commit received, updating index files that make such a repository usable via HTTP.

Now others can clone and pull from that URL:

```
$ git clone https://mgk25.user.srcf.net/project.git
```

`https://www.srcf.net/`

git – more information

Tutorial:

`man gittutorial`

`https://www.kernel.org/pub/software/scm/git/docs/gittutorial.html`

Manual:

`https://www.kernel.org/pub/software/scm/git/docs/user-manual.html`

Reference Manual and Git Pro book:

`https://git-scm.com/doc`

Git for computer scientists (repository data structure):

`https://eagain.net/articles/git-for-computer-scientists/`

Git concepts simplified:

`https://gitolite.com/gcs.html`

cc/gcc – the C compiler

Example:

```
$ cat hello.c
#include <stdio.h>
int main() { printf("Hello, World!\n"); return 0; }
$ gcc -o hello hello.c
$ ./hello
Hello, World!
```

Compiler accepts source ("`*.c`") and object ("`*.o`") files. Produces either final executable or object file (option "`-c`"). Common options:

- `-W -Wall` — activate warning messages (better analysis for suspicious code)
- `-O` — activate code optimizer
- `-g` — include debugging information (symbols, line numbers).

gdb – the C debugger

Best use on binaries compiled with "`-g`".

- `gdb binary` — run command inside debugger ("`r`") after setting breakpoints.
- `gdb binary core` — post mortem analysis on memory image of terminated process.

Enter in shell "`ulimit -c 100000`" before test run to enable core dumps. Core dump can be triggered by:

- a user pressing Ctrl-\ (SIGQUIT)
- a fatal processor or memory exception (segmentation violation, division by zero, etc.)

gdb – some common commands

- `bt` — print the current stack (backtracing function calls)
- `p` *`expression`* — print variable and expression values
- `up`/`down` — move between stack frames to inspect variables at different function call levels
- `b ...` — set breakpoint at specified line or function
- `r ...` — run program with specified command-line arguments
- `s` — continue until next source code line (skip function calls)
- `n` — continue until next source code line (follow function calls)

Also consider starting `gdb` within `emacs` with "`ESC x gdb`", which causes the program-counter position to be indicated in source-file windows.

`make` – a project build tool

The files generated in a project fall into two categories:

- **Source files:** Files that cannot be regenerated easily, such as
  - working files directly created and edited by humans
  - files provided by outsiders
  - results of experiments
- **Derived files:** Files that can be recreated easily by merely executing a few shell commands, such as
  - object and executable code output from a compiler
  - output of document formatting tools
  - output of file-format conversion tools
  - results of post-processing steps for experimental data
  - source code generated by other programs
  - files downloaded from Internet archives

make – writing a Makefile

Many derived files have other source or derived files as *prerequisites*. They were generated from these input files and have to be regenerated as soon as one of the prerequisites has changed, and `make` does this.

A `Makefile` describes

- which ("target") file in a project is derived
- on which other files that target depends as a prerequisite
- which shell command sequence will regenerate it

A `Makefile` contains rules of the form

```
target1 target2 ... : prereq1 prereq2 ...
        command1
        command2
        ...
```

Command lines **must** start with a TAB character (ASCII 9).

make – writing a Makefile (cont'd)

Examples:

```
demo: demo.c demo.h
        gcc -g -O -o demo demo.c

data.gz: demo
        ./demo | gzip -c > data.gz
```

Call make with a list of target files as command-line arguments. It will check for every requested target whether it is still up-to-date and will regenerate it if not:

- It first checks recursively whether all prerequisites of a target are up to date.
- It then checks whether the target file exists and is newer than all its prerequisites.
- If not, it executes the regeneration commands specified.

Without arguments, make checks the targets of the first rule.

make – variables

Variables can be used to abbreviate rules:

```
CC=gcc
CFLAGS=-g -O
demo: demo.c demo.h
        $(CC) $(CFLAGS) -o $@ $<

data.gz: demo
        ./$< | gzip -c > $@
```

- `$@` — file name of the target of the rule
- `$<` — name of the first prerequisite
- `$+` — names of all prerequisites

Environment variables automatically become `make` variables, for example `$(HOME)`. A "$" in a shell command has to be entered as "$$".

make – implicit rules, phony targets

Implicit rules apply to all files that match a pattern:

```
%.eps: %.gif
        giftopnm $< | pnmtops -noturn > $@
%.eps: %.jpg
        djpeg $< | pnmtops -noturn > $@
```

make knows a number of implicit rules by default, for instance

```
%.o: %.c
        $(CC) -c $(CPPFLAGS) $(CFLAGS) $<
```

It is customary to add rules with "phony targets" for routine tasks that will never produce the target file and just execute the commands:

```
clean:
        rm -f *~ *.bak *.o $(TARGETS) core
```

Common "phony targets" are "clean", "test", "install".

`perl` – the Swiss Army Unix Tool

- a portable interpreted language with comprehensive library
- combines some of the features of C, `sed`, `awk` and the shell
- the expression and compound-statement syntax follows closely C, as do many standard library functions
- powerful regular expression and binary data conversion facilities make it well suited for parsing and converting file formats, extracting data, and formatting human-readable output
- offers arbitrary size strings, arrays and hash tables
- garbage collecting memory management
- dense and compact syntax leads to many potential pitfalls and has given Perl the reputation of a write-only hacker language
- widely believed to be less suited for beginners, numerical computation and large-scale software engineering, but highly popular for small to medium sized scripts, and Web CGI

perl – data types

Perl has three variable types, each with its own name space. The first character of each variable reference indicates the type accessed:

- `$...` *a scalar*
- `@...` *an array of scalars*
- `%...` *an associative array of scalars (hash table)*

`[...]` selects an array element, `{...}` queries a hash table entry.

Examples of variable references:

`$days`	= *the value of the scalar variable "*`days`*"*
`$days[28]`	= *element 29 of the array* `@days`
`$days{'Feb'}`	= *the 'Feb' value from the hash table* `%days`
`$#days`	= *last index of array* `@days`
`@days`	= `($days[0], ..., $days[$#days])`
`@days[3,4,5]`	= `@days[3..5]`
`@days{'a','c'}`	= `($days{'a'}, $days{'c'})`
`%days`	= (*key1, val1, key2, val2, ...*)

perl – scalar values

- A “scalar” variable can hold a string, number, or reference.
- Scalar variables can also hold the special `undef` value
  (set with `undef` and tested with `defined(...)`)
- Strings can consist of bytes or characters (Unicode/UTF-8).

  More on Unicode character strings: `man perluniintro`.
- Numeric (decimal) and string values are automatically converted into each other as needed by operators.
  (`5 - '3' == 2, 'a' == 0`)
- In a Boolean context, the values `''`, `0`, `'0'`, or `undef` are interpreted as “false”, everything else as “true”.
  Boolean operators return `0` or `1`.
- References are typed pointers with reference counting.

perl – scalar literals

- Numeric constants follow the C format:
  `123` (decimal), `0173` (octal), `0x7b` (hex), `3.14e9` (float)

  Underscores can be added for legibility: `4_294_967_295`

- String constants enclosed with `"..."` will substitute variable references and other meta characters. In `'...'` only "`\'`" and "`\\`" are substituted.

```
$header = "From: $name[$i]\@$host\n" .
          "Subject: $subject{$msgid}\n";
print 'Metacharacters include: $@%\\';
```

- Strings can contain line feeds (multiple source-code lines).
- Multiline strings can also be entered with "here docs":

```
$header = <<"EOT";
From: $name[$i]\@$host
Subject: $subject{$msgid}
EOT
```

perl – arrays

- Arrays start at index 0
- Index of last element of @foo is `$#foo` (= length minus 1)
- Array variables evaluate in a scalar context to array length, i.e.

```
scalar(@foo) == $#foo + 1;
```

- List values are constructed by joining scalar values with comma operator (parenthesis often needed due to precedence rules):

```
@foo = (3.1, 'h', $last);
```

- Lists in lists lose their list identity: `(1,(2,3))` equals `(1,2,3)`
- Use `[...]` to generate reference to list (e.g., for nested lists).
- Null list: `()`
- List assignments: `($a,undef,$b,@c)=(1,2,3,4,5);` equals `$a=1; $b=3; @c=(4,5);`
- Command line arguments are available in `@ARGV`.

perl – hash tables

- Literal of a hash table is a list of key/value pairs:

```
%age = ('adam', 19, 'bob', 22, 'charlie', 7);
```

Using => instead of comma between key and value increases readability:

```
%age = ('adam' => 19, 'bob' => 22, 'charlie' => 7);
```

- Access to hash table %age:

```
$age{'john'} = $age{'adam'} + 6;
```

- Remove entry: `delete $age{'charlie'};`
- Get list of all keys: `@family = keys %age;`
- Use `{...}` to generate reference to hash table.
- Environment variables are available in `%ENV`.

For more information: `man perldata`

perl – syntax

- Comments start with # and go to end of line (as in shell)
- Compound statements:

```
if (expr) block
  elsif (expr) block ...
  else block
while (expr) block [continue block]
for (expr; expr; expr) block
foreach var (list) block
```

Each *block* must be surrounded by { ... } (no unbraced single statements as in C). The optional continue block is executed just before *expr* is evaluated again.

- The compound statements `if`, `unless`, `while`, and `until` can be appended to a statement:

```
$n = 0 if ++$n > 9;
do { $x >>= 1; } until $x < 64;
```

A `do` block is executed at least once.

perl – syntax (cont'd)

- Loop control:
  - `last` immediately exits a loop.
  - `next` executes the continue block of a loop, then jumps back to the top to test the expression.
  - `redo` restarts a loop block (without executing the continue block or evaluating the expression).
- The loop statements `while`, `for`, or `foreach` can be preceded by a label for reference in `next`, `last`, or `redo` instructions:

```
LINE: while (<STDIN>) {
  next LINE if /^#/;     # discard comments
  ...
}
```

- No need to declare global variables.

For more information: `man perlsyn`

perl – subroutines

Example

```
sub max {
  my ($x, $y) = @_;
  return $x if $x > $y;
  $y;
}

$m = max(5, 7);
print "max = $m\n";
```

- Subroutine declaration:

  sub *name block*

- Subroutine call:

  name(*list*);
  name list;
  &*name*;

  A & prefix clarifies that a name identifies a subroutine. This is usually redundant thanks to a prior sub declaration or parenthesis. The third case passes @_ on as parameters.

- Parameters are passed as a flat list of scalars in the array @_.
- Perl subroutines are call-by-reference, that is $_[0], ... are aliases for the actual parameters. Assignments to @_ elements will raise errors unless the corresponding parameters are lvalues.
- Subroutines return the value of the last expression evaluated or the argument of a return statement. It will be evaluated in the scalar/list context in which the subroutine was called.
- Use my($a,$b); to declare local variables $a and $b within a block.

For more information: man perlsub

perl – operators

- Normal C/Java operators:

```
++ -- + - * / % << >> ! & | ^ && ||
?: , = += -= *= ...
```

- Exponentiation: `**`
- Numeric comparison: `== != <=> < > <= >=`
- String comparison: `eq ne cmp lt gt le ge`
- String concatenation: `$a . $a . $a  eq  $a x 3`
- Apply regular expression operation to variable:
  `$line =~ s/sed/perl/g;`
- `` `...` `` executes a shell command
- `..` returns list with a number range in a list context and works as a flip-flop in a scalar context (for sed-style line ranges)

For more information: `man perlop`

perl – references

Scalar variables can carry references to other scalar, list, hash-table, or subroutine values.

- To create a reference to another variable, subroutine or value, prefix it with `\`. (Much like `&` in C.)
- To dereference such a reference, prefix it with `$`, `@`, `%`, or `&`, according to the resulting type. Use `{...}` around the reference to clarify operator precedence (`$$a` is short for `${$a}`).
- Hash-table and list references used in a lookup can also be dereferenced with `->`, therefore `$a->{'john'}` is short for `${$a}{'john'}` and `$b->[5]` is short for `${$b}[5]`.
- References to anonymous arrays can be created with `[...]`.
- References to anonymous hash tables can be created with `{...}`.

For more information: `man perlref`

perl – examples of standard functions

`split` */pattern/*, *expr*

Splits string into array of strings, separated by pattern.

`join` *expr*, *list*

Joins the strings in *list* into a single string, separated by value of *expr*.

`reverse` *list*

Reverse the order of elements in a list.

Can also be used to invert hash tables.

`substr` *expr*, *offset* [, *len*]

Extract substring.

Example:

```
$line = 'mgk25:x:1597:1597:Markus Kuhn:/homes/mgk25:/usr/bin/bash';
@user = split(/:/, $line);
($logname, $pw, $uid, $gid, $name, $home, $shell) = @user;
$line = join(':', reverse(@user));
```

perl – more array standard functions

`map` *`block list`*

Evaluate *block* for each element of *list*, while loop variable `$_` holds that list element, then return the list formed of all the results returned by *block*.

`grep` *`block list`*

Evaluate *block* for each element of *list* and return the list of all elements of *list* for which *block* returns true.

`sort` *`block list`*

Return a list of all elements of *list*, sorted according to the comparison expression in *block*, which compares two list elements `$a` and `$b`.

Example:

```
@crsids = sort { $a cmp $b } grep { /^[a-z]+[0-9]+$/ }
          map { lc($_) } keys %users;
```

perl – more standard functions

`chop, chomp`
: Remove trailing character/linefeed from string

`pack, unpack`
: build/parse binary records

`sprintf`
: format strings and numbers

`shift, unshift, push, pop`
: add/remove first/last array element

`die`
: abort with error / raise exception

`warn`
: output a warning to standard error

`lc, uc, lcfirst, ucfirst`
: Change entire string or first character to lowercase/uppercase

`chr, ord`
: ASCII/Unicode $\leftrightarrow$ integer conversion

`hex, oct`
: string $\rightarrow$ number conversion

`wantarray`
: check scalar/list context in subroutine call

`require, use`
: Import library module

Perl provides most standard C and POSIX functions and system calls for arithmetic and low-level access to files, network sockets, and other interprocess communication facilities.

All built-in functions are listed in `man perlfunc`. A comprehensive set of add-on library modules is listed in `man perlmodlib` and thousands more are on `https://www.cpan.org/`.

perl – regular expressions

- Perl's regular expression syntax is similar to sed's, but `(){}` are metacharacters (and need no backslashes).
- Appending `?` after `*`/`+`/`?` makes that quantifier "non-greedy".
- Substrings matched by regular expression inside `(...)` are assigned to variables $1, $2, $3, ... and can be used in the replacement string of a `s/.../.../` expression.
- The substring matched by the regex pattern is assigned to $&, the unmatched prefix and suffix go into $` and $'.
- Predefined character classes include whitespace (\s), digits (\d), alphanumeric or _ character (\w). The respective complement classes are defined by the corresponding uppercase letters, e.g. \S for non-whitespace characters.

Example:

```
$line = 'mgk25:x:1597:1597:Markus Kuhn:/homes/mgk25:/usr/bin/bash';
if ($line =~ /^(\w+):[^:]*:\d+:\d+:(.*?):.*?:.*?$/) {
  $logname = $1; $name = $2;
  print "'$logname' = '$name'\n";
} else { die("Syntax error in '$line'\n"); }
```

For more information: `man perlre`

perl – predefined variables

`$_` The "default variable" for many operations, e.g.

```
print;                =    print $_;
tr/a-z/A-Z/;          =    $_ =~ tr/a-z/A-Z/;
while (<FILE>) ...    =    while ($_ = <FILE>) ...
```

`$.` Line number of the line most recently read from any file

`$?` Child process return value from the most recently closed pipe or `` `...` `` operator

`$!` Error message for the most recent system call, equivalent to C's `strerror(errno)`. Example:

```
open(FILE, 'test.dat') ||
  die("Can't read 'test.dat': $!\n");
```

For many more: `man perlvar`

perl – file input/output

- open *filehandle, expr*

```
open(F1, 'test.dat');      # open file 'test.dat' for reading
open(F2, '>test.dat');     # create file 'test.dat' for writing
open(F3, '>>test.dat');    # append to file 'test.dat'
open(F4, 'date|');         # invoke 'date' and connect to its stdout
open(F5, '|mail -s test'); # invoke 'mail' and connect to its stdin
```

- print *filehandle list*
- close, eof, getc, seek, read, format, write, truncate
- "<*filehandle*>" reads another line from file handle FILE and returns the string. Used without assignment in a while loop, the line read will be assigned to $_.
- "<>" opens one file after another listed on the command line (or stdin if none given) and reads out one line each time.

perl – invocation

- First line of a Perl script: `#!/usr/bin/perl` (as with shell)
- Option "`-e`" reads code from command line (as with sed)
- Option "`-w`" prints warnings about dubious-looking code.
- Option "`-d`" activates the Perl debugger (see `man perldebug`)
- Option "`-p`" places the loop

  ```
  while (<>) { ... print; }
  ```

  around the script, such that `perl` reads and prints every line. This way, Perl can be used much like `sed`:

  ```
  sed -e 's/sed/perl/g'
  perl -pe 's/sed/perl/g'
  ```

- Option `-n` is like `-p` without the "`print;`".
- Option "`-i[`*backup-suffix*`]`" adds in-place file modification to `-p`. It renames the input file, opens an output file with the original name and directs the output of `print` into it.

For more information: `man perlrun`

perl – a stream editing example

"Spammers" send out unsolicited bulk email (junk email), for marketing or fraud. They harvest millions of valid email addresses, for example from web pages.

To make email addresses in your web pages slightly harder to harvest, you can avoid the "@" sign in the HTML source. For instance, convert

```
<a href="mailto:jdoe@acm.org">jdoe@acm.org</a>
```

into

```
<a href="mailto:jdoe%40acm.org">jdoe&#64;acm.org</a>
```

The lines

```
perl -pi.bak - <<'EOT' *.html
s/(href=\"mailto:[^@\"]+)@([^@\"]+\")/$1%40$2/ig;
s/([a-zA-Z0-9\.\-\+\_]+)@([a-zA-Z0-9\.\-]+)/$1&#64;$2/ig;
EOT
```

will do that transformation for all HTML files in your directory.

More sophisticated methods: hide an email address in JavaScript or use a CAPTCHA.

perl – a simple example

Generate a list of email addresses of everyone on the Computer Lab's "People" web page, sorted by surname.

Example input:

```
[...]
<tr id='sja55'><td class=name>Aaron, Dr Sam<td class=room>[...]
<tr id='ama55'><td class=name><a href='/~ama55/'>Adams, Andra</a><td [...]
<tr id='sfa27'><td class=name><a href='/~sfa27/'>Aebischer, Seb</a><td [...]
[...]
```

Example output:

```
Dr Sam Aaron <sja55@cl.cam.ac.uk>
Andra Adams <ama55@cl.cam.ac.uk>
Seb Aebischer <sfa27@cl.cam.ac.uk>
[...]
```

perl – a simple example

Possible solution:

```
#!/usr/bin/perl -w
$url = 'https://www.cl.cam.ac.uk/people/index-b.html';
open(HTML, "curl -sS '$url' |") || die("Can't start 'curl': $!\n");
while (<HTML>) {
    if (/^<tr id='([a-z]+[0-9]*?)'>/) {
        $crsid = $1;
        if (/<td class=name>(<a href='.*?'>)?(.*?), (.*?)(<\/a>)?</i) {
            $email{$crsid} = "$3 $2 <$crsid\@cl.cam.ac.uk>";
            $surname{$crsid} = lc($2);
        } else { die ("Syntax error:\n$_") }
    }
}
foreach $s (sort {$surname{$a} cmp $surname{$b}} keys %email) {
    print $email{$s}, "\n";
}
```

Warning: This simple-minded solution makes numerous assumptions about how the web page is formatted, which may or may not be valid. Can you name examples of what could go wrong?

perl – email-header parsing example

Email headers (as defined in RFC 822) have the form:

```
$header = <<'EOT';
From Ian.Grant@cl.cam.ac.uk  21 Sep 2004 10:10:18 +0100
Received: from ppsw-8.csi.cam.ac.uk ([131.111.8.138])
        by mta1.cl.cam.ac.uk with esmtp (Exim 3.092 #1)
        id 1V9afA-0004E1-00 for Markus.Kuhn@cl.cam.ac.uk;
        Tue, 21 Sep 2004 10:10:16 +0100
Date: Tue, 21 Sep 2004 10:10:05 +0100
To: Markus.Kuhn@cl.cam.ac.uk
Subject: Re: Unix tools notes
Message-ID: <514FGFED.mailVJ3982Y@cl.cam.ac.uk>
EOT
```

This can be converted into a Perl hash table as easily as

```
$header =~ s/\n\s+/ /g;  # fix continuation lines
%hdr     =  (FROM => split /^(\S*?):\s*/m, $header);
```

and accessed as in `if ($hdr{Subject} =~ /Unix tools/) ...`

Conclusions

- Unix is a powerful and highly productive platform for experienced users.
- This short course could only give you a quick overview to get you started with exploring advanced Unix facilities.
- Please try out all the tools mentioned here and consult the "`man`" and "`info`" online documentation.
- You'll find on

  `https://www.cl.cam.ac.uk/teaching/current/UnixTools/`

  easy to print versions of the `bash`, `make` and `perl` documentation, links to further resources, and hints for installing Linux on your PC.

⋆ ⋆ Good luck and lots of fun with your projects ⋆ ⋆