

Introduction to UNIX

Libor Forst, SISAL MFF UK

- General introduction
- History, principles
- File system, organization, tools
- Processes, life cycle, communication
- Shell: conception, commands
- Text processing (ed, grep, sed, vi, awk)

Literature

- L.Forst: Shell v příkladech aneb aby váš UNIX skvěle shell; Matfyzpress 2010
`www.yq.cz/SvP`
- The Single UNIX® Specification, Version 3 (POSIX),
The Open Group Base Specifications Issue 7,
IEEE Std 1003.1-2008
`www.opengroup.org/onlinepubs/9699919799`
- manual pages

Literature (basic)

- G. Todino, J. Strang, J. Peek: Learning the UNIX Operating System; O'Reilly & Associates 2002; ISBN 0-596-00261-0
- A. Robbins: UNIX in a nutshell; O'Reilly & Associates 2006; ISBN 978-0-596-10029-2
- L. Lamb: Learning the vi Editor; O'Reilly & Associates 1990; ISBN 0-937175-67-6

Literature (programming)

- C. Newham, B. Rosenblatt: Learning the bash Shell; O'Reilly & Associates 2005; ISBN 0-596-00965-8
- D. Dougherty: sed & awk; O'Reilly & Associates 1997; ISBN 978-1-565-92225-9
- A. Robbins, N. Beebe: Classic Shell Scripting; O'Reilly & Associates Inc., 2005; ISBN 978-0-596-00595-5
- C. Albing, J. Vossen, C. Newham: bash Cookbook; O'Reilly & Associates Inc., 2007; ISBN 978-0-596-52678-8
- E. Quigley: UNIX Shells by Example; Pearson Education Inc. (Prentice-Hall), 2005; ISBN 0-13-147572-X
- S. Kochan, P. Wood: Unix Shell Programming; SAMS, 2003; ISBN 0-672-32390-3

Literature (principles)

- M.J.Bach: The Design of the UNIX Operating System; Prentice-Hall 1986
- E. Raymond: The Art of UNIX Programming; Addison Wesley; 2004; ISBN 0131429019

Conventions

- Fixed part of command (non-proportional font)

- used as it is written:

man [-k] [section] topic

- Variable part of command (italics)

- requested text (word, number etc.) is used:

man [-k] [section] topic

- Optional part of command:

man [-k] [section] topic

- Selection from more alternatives:

{ **BEGIN** | **END** | /regexp/ | cond | } { cmds }

UNIX History

- 1925 - **Bell Laboratories** - communication research
- the 60s - with General Electric and MIT: OS **Multics** (MULTIplexed Information and Computing System)
- 1969 - Bell Labs leaves project, **Ken Thompson** writes assembler, basic OS and file system for PDP-7
- 1970 - Multi-cs => **Uni-x (Brian Kernighan?)**
- 1971 - Thompson requests a new machine PDP-11 for further development - denied
- Thompson fakes work on project of automated office system - a machine granted => text processing tools
- 1973 - UNIX rewritten in C language made for this purpose by **Dennisem Ritchiem**

UNIX Divergence

- mid of the 70s - releasing UNIX to universities: namely University of California **Berkeley**
- 1979 - in Berkeley UNIX rewritten for 32bit VAX as **BSD Unix** (Berkeley System Distribution) version 3.0; today version 4.4
- Bell Labs migrate under **AT&T** and development goes on: version **III** to **V.4** - so called **SVR4**
- UNIX release for commerce: Microsoft and SCO develop for Intel **XENIX**
- established UNIX International, OSF (Open Software Foundation), X/OPEN,...

UNIX Variants

- SUN: **Sun OS, Solaris**
- Silicon Graphics: **Irix**
- DEC: **Ultrix, Digital Unix**
- IBM: **AIX**
- HP: **HP-UX**
- Siemens Nixdorf: **SINIX**
- Novell: **UNIXware**
- SCO: **SCO Unix**

- FreeBSD, NetBSD, OpenBSD,...
- Linux

UNIX Standards

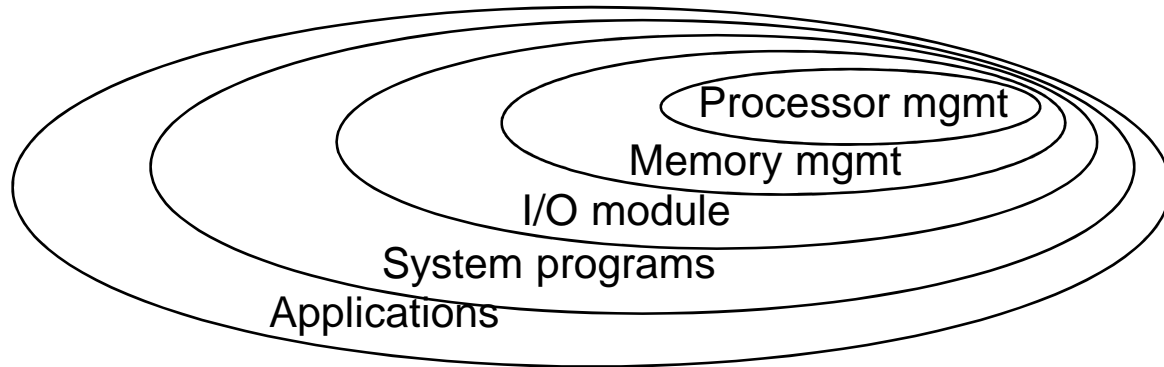
- SVID (System V Interface Definition)
 - “The Purple Book”, issued by AT&T for the first time in 1985 as a standard, conformance with which was required for branding a system “UNIX”
- POSIX (Portable Operating System based on UNIX)
 - series of standards by IEEE marked P1003.xx, gradually overtaken by international top standard organisation ISO
- XPG (X/Open Portability Guide)
 - recommendation of X/Open consortium, founded by main workstation producers in 1984
- Single UNIX Specification
 - standard of Open Group organisation, founded in 1996 by joining of X/Open and OSF
 - Version 2 (**UNIX98**), Version 3
 - conformance is now required for branding a system “UNIX”

UNIX Characteristics

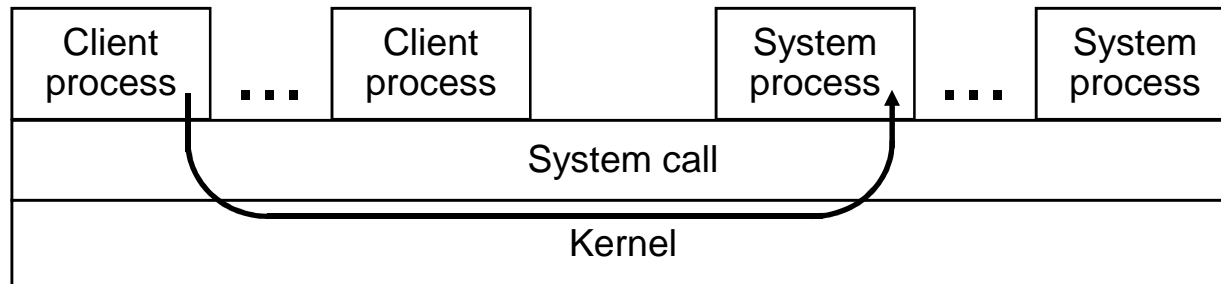
- inspired but not burdened by the past
- noncommercial environment
- open operating system
- file system
- users, groups
- processes, communication
- command interpreter, GUI
- utilities, C language
- portability, flexibility
- networking support
- public domain SW (e.g. GNU)
- command **man**

OS Models

Classic OS



UNIX



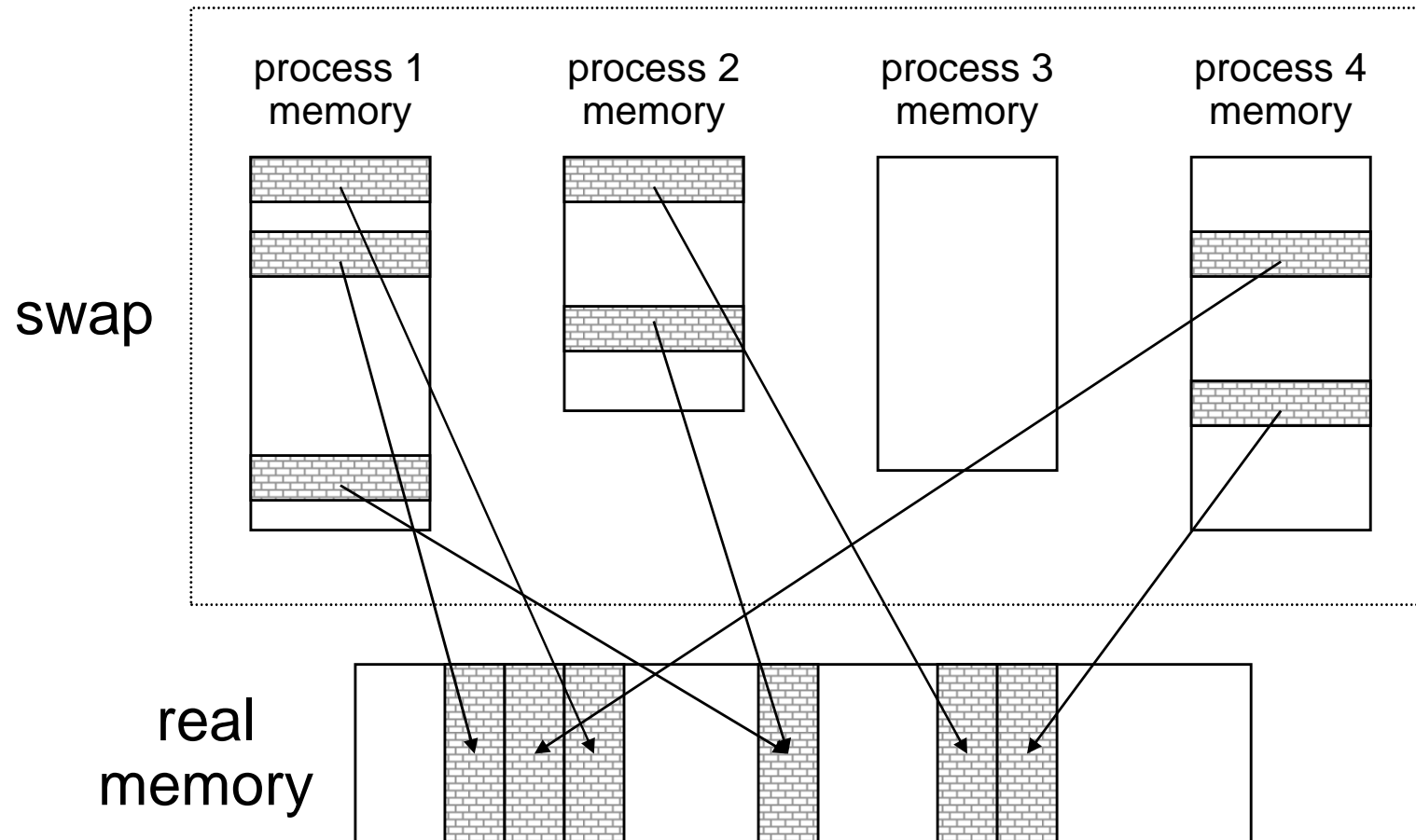
OS kernel functions

- Job execution control (creation, termination, suspending, communication, peripherals,...)
- File system management (disc organisation, file creation and removal, protection, consistency keeping,...)
- Memory management (allocation, releasing, protection, holding of temporarily unused memory - *swapping* or *paging*,...)
- Process scheduling for CPU time sharing (scheduling algorithm, time slices management, priorities,...)

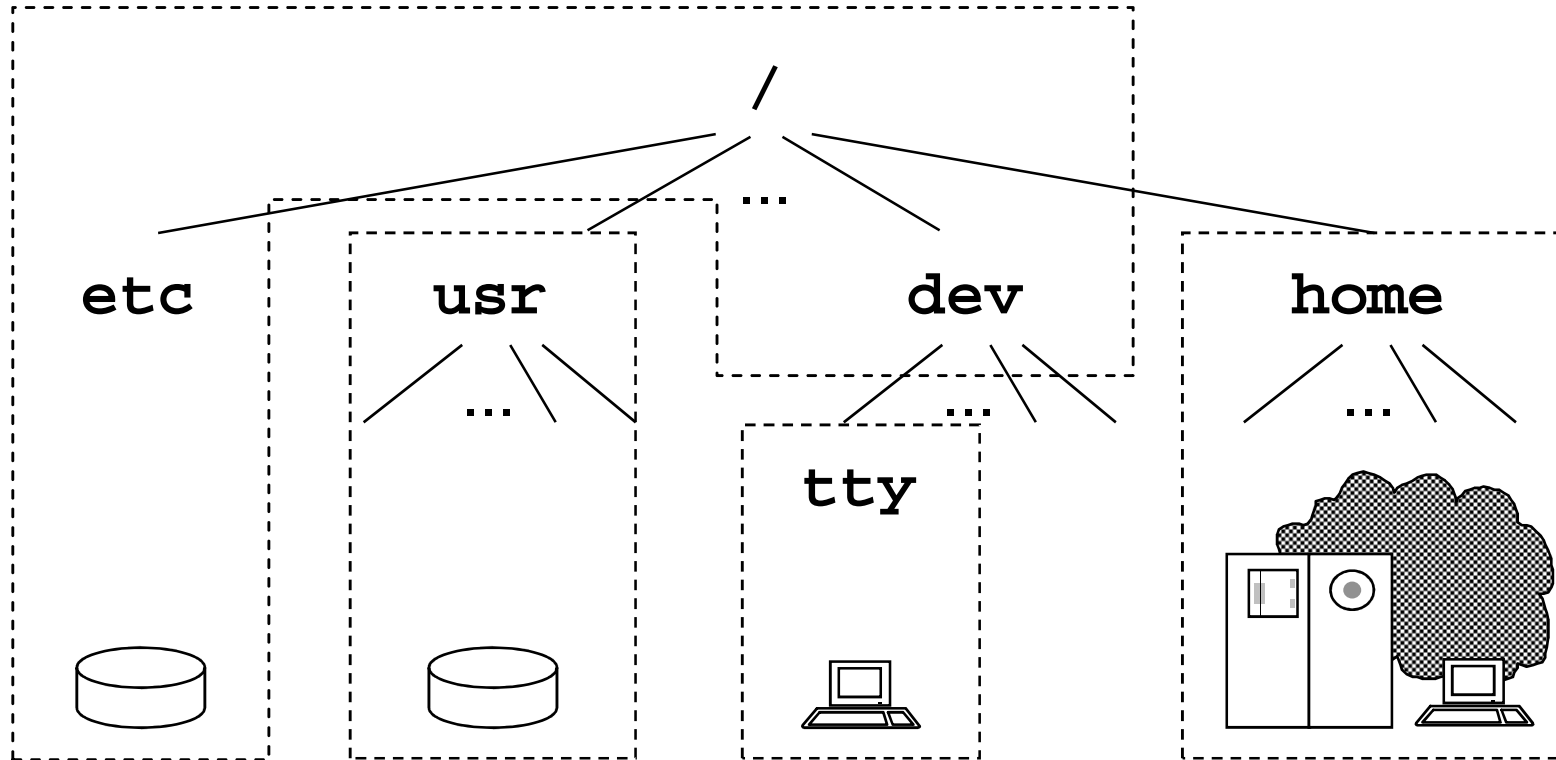
HW requirements

- Possibility to run in two modes:
 - user mode: limited access to memory, instructions etc.
 - kernel mode: unlimited privileged mode
- Hierarchical handling of interrupts
 - external: hardware (disc, peripherals, ...)
 - internal: CPU conditions (e.g. addressing error, division by zero, ...)
 - software: special instruction
- Memory management for virtual memory usage

Virtual memory



Integrated hierarchical file system



Directory tree

- **/bin** - essential system commands
- **/dev** - special files (*devices*)
- **/etc** - configuration files
- **/lib** - essential system libraries
- **/tmp** - public directory for temporary files
- **/usr/include** - C language headers
- **/usr/man** - manual pages *
- **/usr/spool** - *spool* (printing, email,...) *
- **/usr/local** - local installations *
- **/home** - root of home directories *

* may vary on some systems

Process, communication

- Process
 - general idea: running user or system program
 - created by duplication of parent process
 - process list: command `ps`
- Communication
 - when started, parent prepares data for son;
no way to share data vice versa
 - pipe - data flow from a producer to a consumer:
`ls | more`
 - advanced tools (e.g. shared memory)

Command Interpreter (*shell*)

- essential program for UNIX operating
- independent system component: more shells exist
- command format:

command -options operands e.g. `ls -l /etc`

- metacharacters, e.g.:

`ls *.c > "output *.c"`

- commands:
 - internal: e.g. `echo`, `cd`, `pwd`
 - external: files in file system (path to search: **PATH**)

Shell language

- shell interprets own programming language
 - control flow statement (e.g. `for`, `if`)
 - variables

```
PATH=/bin:/usr/bin:$HOME/bin
```

- language controls text substitutions (*text processor*)
- programming directly on the command line
- shell-script - file with stored shell program

```
sh test.sh; ./test.sh
```

man command

- Call:
`man [-k] [section] topic`
- Manual pages sections:
 - 1 - general user commands
 - 2 - kernel functions (*syscalls*)
 - 3 - library (C language) functions
 - 4 - devices and device drivers
 - 5 - formats of (configuration) files
 - 6 - trivial application programs
 - 7 - miscellaneous
 - 8 - administrator commands and programs

List of users (`/etc/passwd`)

```
forst:DxyAF1eG:1004:11:Libor Forst:/u/forst:/bin/sh
```

Field semantics:

- user login name
- encoded password (today e.g. in `/etc/shadow...`)
- user number (*UID*); superuser (*root*) has UID 0
- number (*GID*) of user's primary group
- full name (optionally with comment)
- home directory
- login-shell

List of groups (/etc/group)

```
users::11:operator,novak
```

Field semantics:

- group name
- *unused*
- group number (*GID*)
- group members

Users having a group as their primary group are members of the group, too.

User session

After logging into system (locally or remotely - e.g. via `ssh`, `putty.exe`) user's *login-shell* is started.
Thereby, the user *session* is started.

- closing session: `logout`
- closing shell: `exit`
- user (login-shell) change: `login user`
- another user shell start: `su [-] [user]`
- user identity display: `id, whoami, who am i`
- information about system: `uname [-amnrsv]`
- list of logged-in users: `who, w`
- session log listing: `last`

Inter-user communication

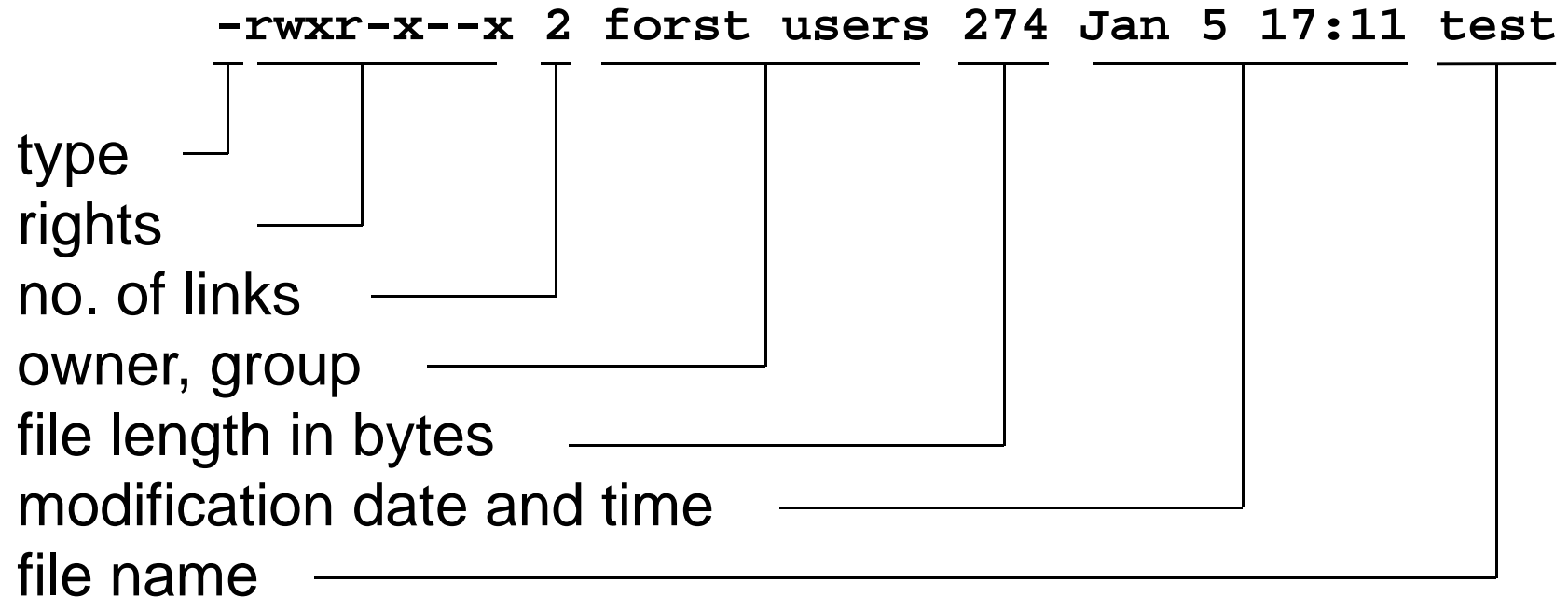
- on-line (messages):
 - sending: `write user`
 - receipt perm./denial: `mesg [y | n]`
- on-line (dialogue):
 - command: `talk user[@host]`
- off-line: e-mail
 - reading: `mail`
 - sending: `mail [-v] [-s subject] email...`
 - receipt message: `biff [y | n]`
 - mail forwarding: `$HOME/.forward`

```
forst@ms.mff.cuni.cz  
" | /usr/local/bin/filter"
```

File system

- hierarchical system
- unified approach to directories, devices etc.
- disc partitioning, remote disc mounting
- consistency, synchronization (**sync**, **fsck**)
- protection (access rights)
- naming rules (length, charset, case sensitivity, hidden files)
- paths (absolute, relative, . and ..)
- text files format (<**LF**>)

ls command



options: long output (**l**), single column (**1**), include hidden files (**aA**), sort by time (**t**), reverse sort (**r**), flag file type (**F**), traverse recursively (**R**), don't follow directories (**d**), follow links (**L**)

File types

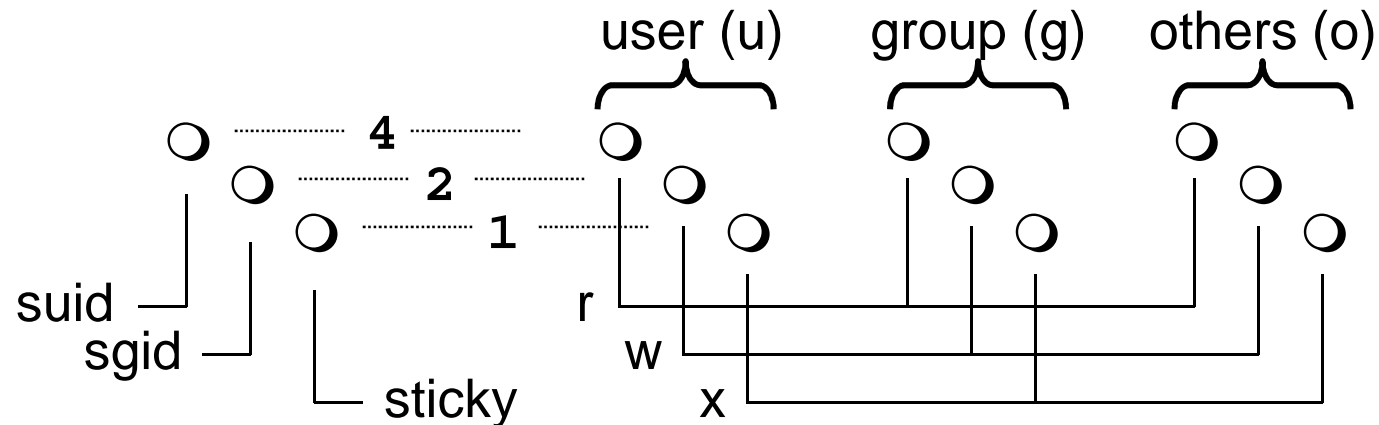
- File types in `ls` command output:
 - regular file - sequence of bytes
 - d** directory - set of binary records describing files and subdirectories
 - b** block device
 - c** character (raw) device
 - l** symbolic link
 - p** named pipe
 - s** socket

- Type recognition: `file` command

Access rights (file modes)

- ownership categories: user (**u**), group (**g**), others (**o**); exactly user's most special category is significant
- three permissions: read (**r**), write (**w**), execute file and work with directory (**x**)
- setUID, setGID (**s**) for executable files: run under owner (user and/or group) identity
- setGID for directory: new files will have directory's group owner (default on many systems)
- sticky bit (**t**) for directories: only file owners and root can remove and rename files (e.g. /tmp)

File mode change



- access rights change (only owner and root):
`chmod [-R] 751 file...`
`chmod [-R] og-w,+x file...`
- owner change (only root): `chown`
- group owner change (only group member): `chgrp`
- default file mode mask: `umask [masked_bits]`
- shell with new default group: `newgrp group`

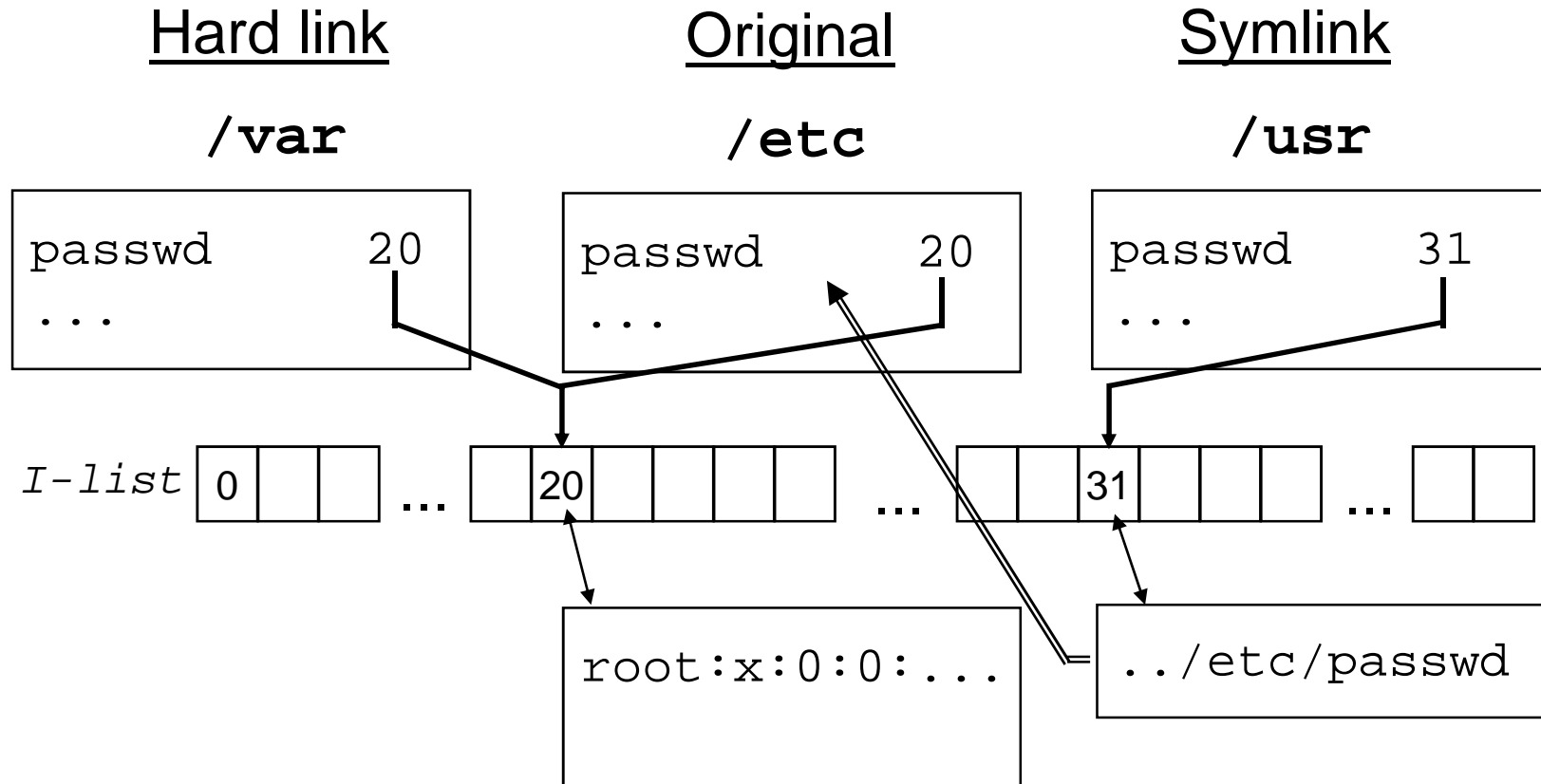
Disc organisation

- Physical: *sector, track, cylinder, surface*
- Logical: *partition* (correspond to block/raw device)
 - display filesystems: **df** command
 - configuration file **/etc/fstab**
- System-level: *filesystem*
 - boot block
 - superblock(s)
 - i-list (list of i-nodes)
 - data blocks
- Filesystem image kept in memory (**sync, fsck**)

Index node (i-node)

- Every file in filesystem has exactly one index node structure containing:
 - number of links
 - user and group owner (ID)
 - permissions
 - file type
 - file size
 - time of
 - last file modification
 - last access to file
 - last i-node modification
 - data block references
- List files including i-node numbers: `ls -li`
- List i-node content (not in SUSv3): `stat`

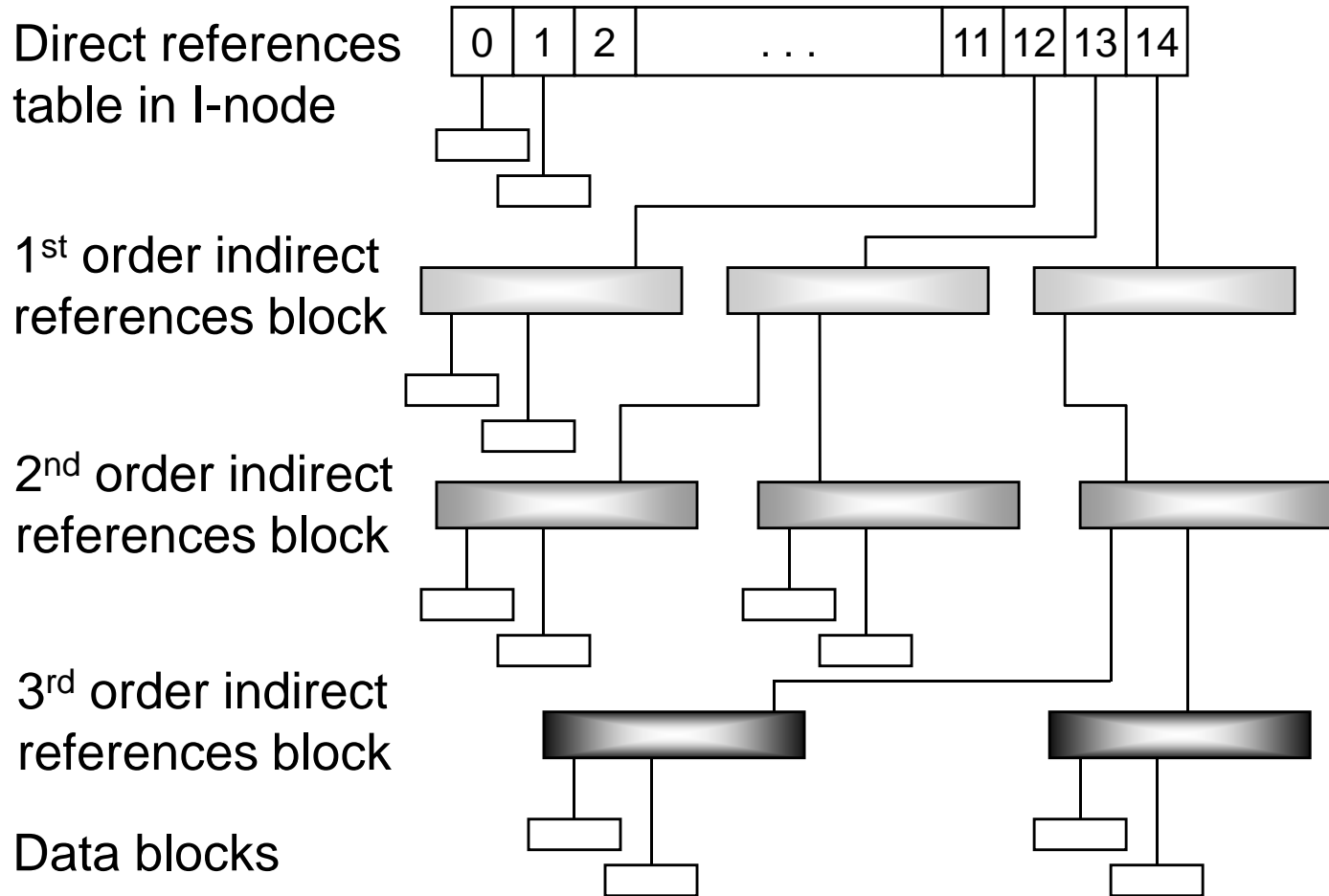
Links



```
ln /etc/passwd passwd
```

```
ln -s ../etc/passwd passwd  
(loop: Too many symlinks)
```

Data block addressing



General commands

- copy file: `cp [-pR]`
- move (or just rename) file: `mv`
- remove file: `rm [-rfi]`
- change date + time: `touch [{ -t time | -r file }]`
- change current directory: `cd`
- print working directory path: `pwd [-P]`
- make directory: `mkdir [-p] [-m mode]`
- remove directory: `rmdir`

- no undelete command!

File content output

- output (concatenate) files: **cat** [*files*]
- output per pages: **more**, **pg**, **less**
- file beginning output: **head** [**-n** *n*] [*files*]
- file end output: **tail** [{**-n**|**-c**} [**+**]*n*] [**-f**] [*files*]
- file output for printer: **pr**
- file output with numbering lines: **nl**
- count bytes, words and/or lines: **wc** [**-cwl**]
- duplication to output and file: **tee** [**-a**] *file*

- binary file output: **od** [**-t** *fmt*] [**-j** *off*] [**-N**/*len*]
- extract strings: **strings**

more command

- Call:
`more [-n] { +line | +/regexp | } [files]`
- Commands (* - multiplication prefix *k* accepted):
 - space, **d** ... next page, next half of page (*)
 - Enter ... next line (* - *k* will set a default)
 - **s**, **f**, **b** ... skip *k* lines, pages, pages backward (*)
 - */regexp*, **n** ... search for *k*-th string occurrence (*)
 - **'** ... return to search beginning
 - **!cmd**, **v** ... start shell, editor
 - **=**, **h** ... file position output, display help
 - **:n**, **:p** ... skip to next/previous file

Printing

- | | <u>SUSv3</u> | <u>System V</u> | <u>BSD</u> |
|------------------------|-------------------------|-------------------------|-------------------------|
| • print: | <code>lp [file]</code> | <code>lp [file]</code> | <code>lpr [file]</code> |
| | <code>-d printer</code> | <code>-P printer</code> | <code>-d printer</code> |
| • show printer status: | | <code>lpstat job</code> | <code>lpq job</code> |
| | | | <code>-d printer</code> |
| • cancel printing job: | | <code>cancel job</code> | <code>lprm job</code> |
| | | | <code>-d printer</code> |
- “printers” description: `/etc/printcap`
 - default printer: `PRINTER` variable
 - spool location: `/var/spool/*`
 - print formatting: `pr`, `mpage`

Text processing

- files and/or directories comparison:

```
diff [ -bBi ] { -e | -Cn | -rqs } file1 file2
```

```
comm [ -123 ] file1 file2    (have to be sorted)
```

- cutting parts of lines (cannot change order of parts):

```
cut [ -s ] { -clist | -flist -dchar } [files]
```

- pasting “columns” of files; pasting all lines of one file:

```
paste [[ -s ] -dchars ] [files]
```

- splitting file per lines or blocks:

```
split [{ -llines | -bbytes[{k|m}] } ] [ file [ name ] ]
```

- character conversion:

```
tr [ -cds ] table1 [table2] př.: tr 'A-Z\n' 'a-z:'
```

sort command

- Call:
`sort [-s] [-kbeg[,end][mod]] [-td] [-ucm] [files]`
- Sorts files to output, or to a file (`-o file`)
- Key fields definition:
 - *beg* ... first character position, *end* ... last char pos
 - format: *field*[*.char*] ... numbered from 1
- Modifiers: **b** (w/o blanks), **f** (ignorecase),
n (numbers), **r** (reverse)
- Options: **t** (field separator, default: sequence of spaces),
u (exclude equal keys), **m** (merge only),
c (check only), **s** (stable - not in SUSv3)
- Beware of local settings (`LC_ALL=C`)
- Similar command: **uniq** (does not sort, can count)

find command

- Call: `find path... condition... action`
- Conditions:
 - `name, path, size, type, links, inum, fstype`
 - `user, group, perm`
 - `atime, ctime, mtime, newer`
 - depth within the tree
 - negation (`!`), `-o`, `-a`, parentheses
 - numerical values: `n`, `+n`, `-n`; filenames: wildcard patterns
- Actions:
 - `print` (usually default)
 - `exec`; filename subst.: `{ }`, end of params: semicolon
- Example:
`find / -name *core -atime +7 -exec rm { } ";"`
- Searching for executable files: **which, whereis**

dd command

- Provides data copying and conversions
- Name and parameter syntax derived IBM 360 system JCL statement DD (Data Definition)
- Parameters:
 - **if=***file* - input (default: standard input)
 - **of=***file* - output (default: standard output)
 - **bs=***expr* - block size (*n[k][xn[k]]...*)
 - **count=***n* - number of blocks
 - **skip=***n* - seek from the file beginning
 - **conv=***c[,c]...* - conversion(s)
- Conversion ASCII/EBCDIC, fixed line length/LF
- Example:

```
dd if=myfile bs=8 count=1
```

join command

- Provides database *join* operation - file merge based on parity of key within records
- Options:
 - `t c` - field separator [sequence of whitespace]
 - `{1|2} f` - key field number in file 1 or 2 respectively [1]
 - `a n` - take also unpaired lines from file *n*
 - `v n` - take only unpaired lines from file *n*
 - `e str` - substitution for empty fields []
 - `o list` - output format [key and then all fields in a row]
- Output format syntax:
 - list of field descriptions separated by commas, spaces, or written in more parameters
 - field description: *n.f* or 0 (join field)
- Illustration example:

```
ls -l | tr -s ' ' : |  
join -1 3 -t : -o1.9,2.3 - /etc/passwd
```

xargs command

- call: **xargs *command***
 - calls *command*, standard input content is used for command parameters
 - e.g.: `xargs rm < files_to_delete`
- call: **xargs {*-L**lcnt* | *-n**wcnt*} *command***
 - repeats *command* for every *lcnt* lines or *wcnt* words from standard input; particular portion of input is used for command parameters
- call: **xargs *-I**fn* *command***
 - repeats *command* for every input line replacing every occurrence of symbol defined as *fn* by line text
 - e.g.: `ls *.c | xargs -I{} cp -p {} {}.bak`

Archiving

- directory archiving: `tar {c|t|x} [f file] [files]`
 - e.g.: `tar cf - . | ssh host tar xf -`
 - SW package distribution
- in SUS replaced by `pax` command
- file compression
 - historical standard (`.Z`): `compress`
 - GNU (`.gz`): `gzip`, `gunzip`
- system backup: `backup`, `dump`, `restore`
- remote backup: `rdump`, `rrestore`

Line-oriented editors

- ed** - editor available often even in diagnosis mode
- edits file copy, result has to be written back
- commands are taken from standard input
- batch editing (**ed**-scripts)

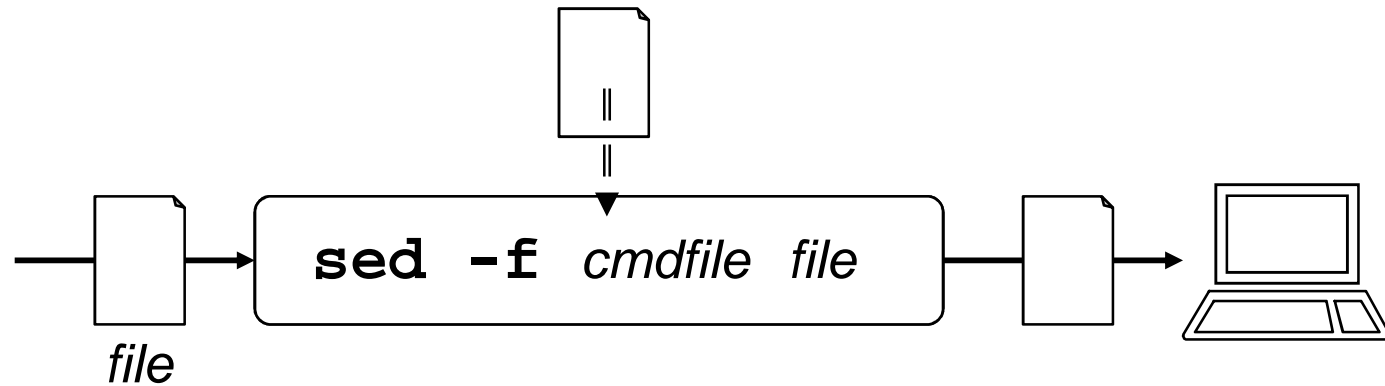
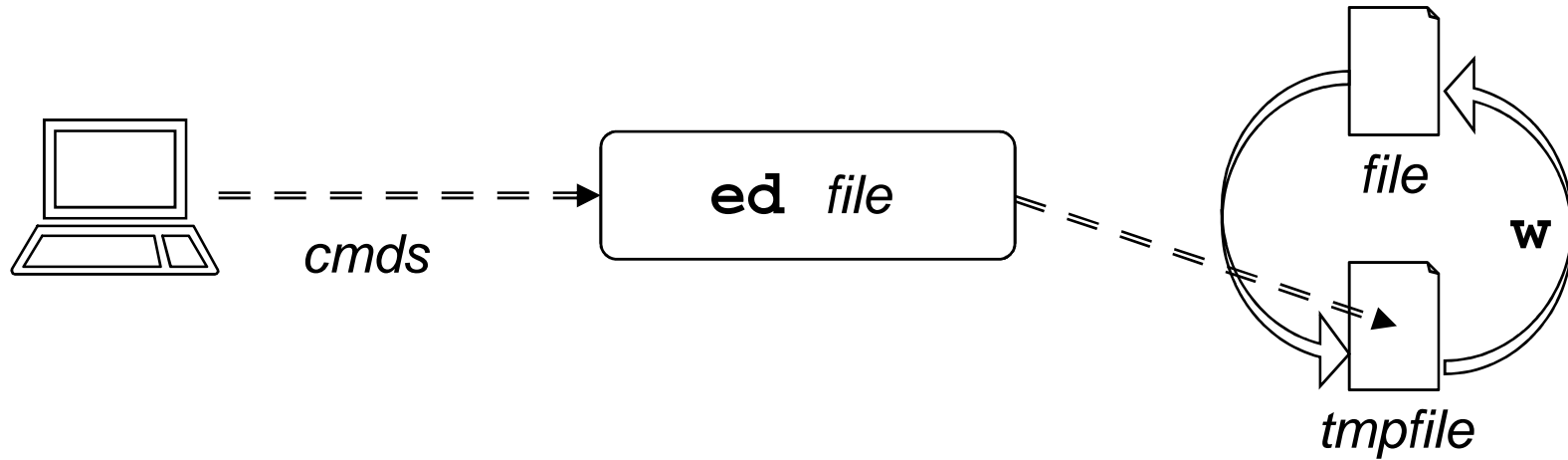
- call: `ed file`

- sed** - stream editor
- edits standard input, result to standard output
- editor commands are given as call parameter

- call: `sed commands [file ...]`

- example: `hostname | sed 's/\.*//'`

Workflow of `ed` and `sed`



Command format, line address (ed)

- Command syntax:

$[address[, address]]cmd[parameters]$

- In every moment, one line holds current line status (last line touched by last command; last line of file on start)
- Line address formats and semantics:
 - current line (usual default)
 - $\pm[n]$ line relative to current line
 - n line with absolute number n (numbered from 1)
 - $\$$ the last line of the file
 - $/pat/$ following line containing pattern
 - $?pat?$ previous line containing pattern
 - $'x$ line marked by mark (letter) x
 - $adr\pm[n]$ line relative to line addressed by adr

Basic regular expressions (`ed`, `sed`, `vi`)

Way how to define strings in many utilities. Metachars:

- `.` ... any character
- `[list]`, `[^list]` ... any char from the list, or list complement
e.g.: `[a-zA-Z0-9_]`, `[^]`, `[]^-]`
- `[[:class:]]` ... any character from the class
e.g.: `[[:alnum:]]`, `[[:xdigit:]]`
- `^`, `$` ... start, or end of line (used on start or end of regexp)
- `\c` ... metachar used as regular char (e.g.: `\.` means dot)
- `exp*` ... any number of occurrence of the last subexpression
e.g.: `a*`, `[0-9][0-9]*`
- `exp\{n\}`, `exp\{m,[n]\}` ... repeating n times, $m-n$ times
- `\(, \)`, `\n` ... grouping of subexpression, backreference
e.g.: `\(ab\)*`, `A\(.\)1A`

Positional commands of `ed`

Commands having current line as default address,
commands marked by * cannot have block address:

print, num, list ... print, numbered, incl. control chars

delete ... removing lines

append*, change, insert* ... inserting lines (end: a single dot)

```
e.g.: 0a
      new line 1
      new line 2
      .
```

move, to ... moving, copying lines

```
e.g.: /begin/, /end/ t $
```

mark* (**kx**) ... setting a mark x (letter)

join ... joining lines (deletes LF, def. +1)

substitute ... string replacement

substitute command (ed)

Syntax:

`s / pattern / replacement / {g|n}`

Character after command name defines string delimiter

e.g.: `s / \ / $ / /` or `s = / $ = =`

Pattern: regexp, replacement: text with metacharacters:

- `\n` ... backreference (slow!)
e.g.: `s / \ (. * \) \ (. * \) / \2 \1 /`
- `&` ... whole original text matching pattern
e.g.: `s / . * / (&) /`

Global substitution starts search of next pattern occurrence after the last character it has already modified:

e.g.: `s = / \ . / = / =g` ... does not replace „/ . / . /“

A star “eats” the longest string that matches:

e.g.: `s / \ (. * \) - / \1 /` ... removes the last hyphen

Global commands of `ed`

Commands having “entire file” as default address:

global, invert (`⌵`) ... command execution on selected lines

`g / pattern / cmd [\<LF>cmd]`

write (`w [file]`) ... saving (under original name)

(when address part used, only these lines are saved!)

`w file` ... appending to file

`w! cmd` ... writing to command pipe

Commands having “the last line of the file” as default address:

read (`r [file]`) ... inserting text of another file

`=` ... displaying line number

Non-positional commands of `ed`

Commands without address part:

<code>undo</code>	... undoing last change
<code>edit (e [<i>file</i>])</code>	... (re-)opening file
<code>file (f <i>file</i>)</code>	... changing file name
<code>quit</code>	... ending editor
<code>help</code>	... explanation of the last error

Examples of `global` command usage

- `g/integer/s//longint/g`
changes all integers to long ones
- `g/procedure/i\
{ begin of procedure }\
.`
inserts a comment line before every procedure
- `g/^Chapter/ . W index\
./ W index`
writes chapter index
- `g/^/ m 0`
rewrites file “crablike”

grep command

- Name origin: **g**/*re*/**p**
- Variants:
 - **egrep** (**-E**, *extended* regular expressions)
 - **fgrep** (**-F**, *fixed* string only)
- Options:
 - c**(*count*), -**l**(*listfiles*), -**n**(*number*), -**q**(*quiet*)
 - i**(*ignorecase*) , -**x**(*exact*), -**v**(*invert*)
 - e** *expression*, -**f** *filename*
- Extensions:
 - w**(*word*) , -**H**(*head*)
 - n* ... print *n* lines surrounding matching ones
- Quick implementation of regular expressions!

sed filter

- stream editor
- edits input (usually another program output)
- modified lines (and/or printed ones) writes to output
- call:

```
sed [-n] { [-e] cmd | -f script } [file]
```

- commands similar to `ed` ones
- separated by semicolon or end-of-line
- executed in given order
- cannot end with a space

Command format, line address (**sed**)

- Command syntax:

[address[, address]]cmd[parameters]

- No current line exists, commands with empty address part are applied to every line

- Line address formats and semantics:

n line with absolute number *n* (numbered from 1)

\$ the last line of the file

/pat/ every line matching pattern

- Address space complement: *address ! command...*

- Compound statement: *address {
 commands...
 }*

- Comment: *# comment...*

Commands of `sed` (I)

- the same commands as in `ed`:
 - `p`, `d`, `s`, `w`, `q`
 - `a`, `c`, `i`
command itself and all lines but the last ended by “\
`sed '3a\
fourth\
fifth'`
- new parameters of `substitute` command
 - `p` ... line is printed after editing
 - `w file` ... line is written to a file after editing
- character conversion
 - `y /intable/ outtable/`
function similar to the `tr` command

Commands of `sed` (II)

- control flow
 - `n(ext)` ... closing work with line, reading next one
 - `:label` ... label definition
 - `b(ranch)[label]` ... jump to a label (to the end of cmds)
 - `t(est) [label]` ... conditional jump
(jump if some substitution was made since last reading-in of a line, or `test` command execution)

Example:

```
:loop
```

```
s:/\./:/:g
```

```
t loop
```

... removes all “/ ./” sequences from a path

Commands of `sed` (III)

- more lines in *pattern space* (separator: `\n`)
 - **N**(ext) ... appending next line from input
 - **P**(rint) ... printing first line from pattern space
 - **D**(elete) ... removing first line from pattern space

Example:

```
:loop
/foo([ ^) ]*$/ {
    N
    b loop
}
/foo(/s/);/, true);/
```

... adds a new parameter to function calls

Commands of `sed` (IV)

- work with *hold space*
 - `h`, `H`(old) ... copying (appending) to hold space
 - `g`, `G`(et) ... copying (appending) to pattern space
 - `x`(change) ... exchanging of content of both spaces

Example:

```
/procedure/h
/^end/ {
    p
    g
    s/procedure/{ end of/is/ *(.*/ }/
}
```

... adds comments with name behind procedures

sed command examples (I)

- `sed /record/,/end/d program.pas`
prints program without record definitions
- `sed '/procedure/i\
{ begin of procedure }' program.pas`
inserts a comment line before every procedure
- `sed '1p;$p' program.pas`
prints file with duplicated first and last line
- `sed -n '4,6!p' program.pas`
prints file without lines #4 to #6

sed command examples (II)

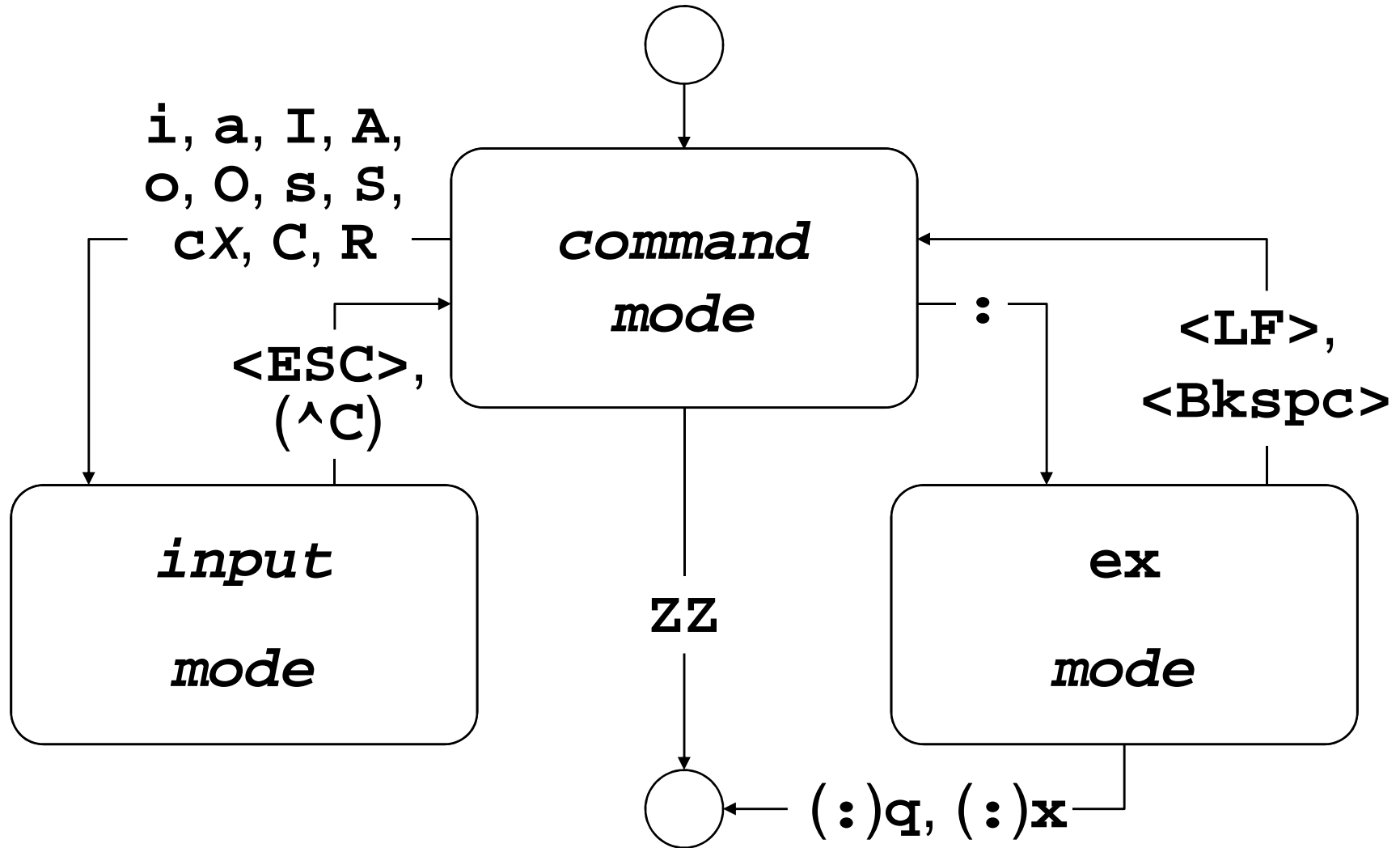
- `sed 's/:.*//;s/$/./' /etc/passwd`
result: forst.
- `ls *.c | sed 's/\(.*\)./cp -p & \1.bak/'`
result: cp -p test.c test.bak
- `echo ab | sed 's/a/b/;s/b/a/'`
result: ab
correct: `y/ab/ba/`
or: `s/a/\n/g;s/b/a/g;s/\n/b/g`
- `sed 's/.*:\(.*\) \(.*\):.*\/\2 \1/' /etc/passwd`
result: Cooper:/home/spock Sheldon
correct: `s/.*:\(.*\) \([^:]*\):.*\/\2 \1/`

vi editor

- visual editor
- genesis: **ed** ⇒ **ex** ⇒ **vi**
- fullscreen editor
- available on every UNIX
- wide spectrum of commands
- small number of necessary commands
- editing on temporary copy of file
- call:

```
vi [-rR] {+[line] | +/pattern } [files]
```


Modes of vi



Essential commands of `vi`

- `vi file` ... editor call
- `i` ... text inserting mode
- *text being inserted*
- `<ESC>` ... finishing input mode
- `h, j, k, l` ... cursor movements
- `/pattern` ... string pattern searching
- `x, dd` ... deleting a char, a line
- `A` ... appending to the end of line
- `J` ... joining lines
- `ZZ, :x` ... closing editor
- `:q!` ... cancelling editor

Movement commands (I)

Commands may be prefixed by repeating factor k

- **h** (<BKSPC>), **j**, **k**, **l** (<SPACE>) ... k places (←, ↓, ↑, →)
- **w**, **b**, **e**, **W**, **B**, **E** ... k words (forward, backward, to the end of word, ignoring punctuation)
- **(**, **)**, **{**, **[** ... to next (prev.) sentence, paragraph, section
- **+** (<LF>), **-** ... to the beginning of next (prev.) line
- **\$**, **o**, **^** ... to the end, beg., first nonspace char on the line
- **fX**, **Fx**, **tX**, **Tx**, **;**, **,** ... char x on line (forward, backward), char before/after x , repeat, repeat in opposite direction
- **/regexp**, **?regexp**, **/**, **?**, **n**, **N** ... search string, forward, repeat pattern, repeat search, repeat in opposite direction
- **^F**, **^B**, **^D**, **^U** ... page forward, backward, half a page

Movement commands (II)

Commands may be prefixed by absolute value *k*:

- *k*| ... *k*-th position on the line
- [*k*]H ... move to *k*-th line on the screen [1]
- [*k*]L ... move to *k*-th line from the end of screen [1]
- M ... move to mid-line on the screen
- [*k*]G ... move to *k*-th line of the file [last]

Mark *x* (letter) handling:

- *x* ... move to position marked by mark *x*
- `` ... move to last position marked
- '*x* ... move to beginning of the line marked by mark *x*
- '' ... move to beginning of the last marked

(mark is being placed by **m***x* command)

Insertion, modification

Commands may be prefixed by repeating factor k

- **i**, **a**, **I**, **A** ... inserting before (behind) cursor, line
- **o**, **O** ... opening new line above (below) current one
- **~** ... changing case of letter under cursor *
- **rX** ... replacing character under cursor by character x *
- **R** ... replacing text (move to input mode in replace mode)
- **cm** ... changing text from cursor up to position defined by any movement command m
- **cc**, **C** ... changing whole line, changing to end of line
- **s**, **S** ... removing char (line) and enter input mode

Commands marked * do not switch to input mode.

Removal, work with buffers

Commands may be prefixed by repeating factor k

- \mathbf{x} , \mathbf{X} ... deleting character under (before) cursor
- $\mathbf{d}m$... deleting text from cursor up to position defined by any movement command m
- \mathbf{dd} , \mathbf{D} ... deleting whole line, deleting to end of line

Deleted text is stored into numbered buffer.

- \mathbf{p} , \mathbf{P} ... pasting buffer behind (before) cursor (or line)
- $\mathbf{"n}p$, $\mathbf{"n}P$... pasting n -th last buffer
- $\mathbf{"x}p$, $\mathbf{"x}P$... pasting buffer named x (lowercase letter)

Copying text into (named) buffer:

- $\mathbf{["x]ym}$... copying text up to position defined by m
- $\mathbf{["x]yY}$, $\mathbf{["x]Y}$... copying whole line

Miscellaneous commands (**vi**)

- **.** ... repeating last modification command
- **u** ... undoing last modification command
- **U** ... restoring last changed line to original state
- **J** ... joining line with next one
- **%** ... move to pair **)**, **]** or **}** (not **>**)
- **^L** ... redraw screen
- **z<LF>**, **z.**, **z-** ... scroll, current line will be placed on the top (middle, bottom) of the screen
- **^E**, **^Y** ... scroll by one line
- **^G** ... current file and line info displaying
- **!m cmd**, **!!cmd** ... extracting block of text, using it as input for *cmd* and storing output back to the text
- **<m**, **>m** ... indenting
- **@x** ... executing commands stored in buffer *x*
- **^W**, **^V** ... (in input mode) deleting word, inserting control char

ex - command extensions (I)

- addresses may be separated by semicolon - the first line will become current instead of the last one
- extension of **substitute** command
 - **c** parameter ... replacing with confirming (**y<LF>**)
 - **~** metachar in regexp ... previous regexp
 - **\<** and **\>** sequences in regexp ... beg. and end of word
 - **\u**, **\l**, **\U** and **\L** sequences in replacement ... lettercase changing (whole word)
- new commands
 - **co** (copying, an alias of **t** command)
 - **j[!]** ... joining lines; after **.** adds two spaces, after **)** none, one otherwise (**!** ... means no spaces)
 - **ya[x]**, **pu[x]** ... work with (named) buffers

ex - command extensions (II)

- **sh**, **!cmd** ... shell run, command run
- **so** ... executing `ex` source code
- **w!**, **w>>** ... writing to read-only file, appending to file
- **x**, **wq** ... storing file and exiting editor
- **q!** ... quitting editor without storing changes
- **n[!]** ... editing next file from list (without storing changes)
Named buffers, the last regexp and editing command remain available.
- **e[!]** [*file*] ... editing another file (% substitutes current file name, # the last used file name)
- **ab** *word string*, **una** ... abbreviation
- **map[!]** {*char* | #*n*} *string*, **unm** ... character or function key mapping (for input mode); control chars entered using **^v**

vi editor settings

Options setting: `set`, list of all options: `set all`

- `autoindent`, `ai` ... new lines autoindentation [`noai`]
- `directory=dir`, `dir` ... working directory [`=/tmp`]
- `ignorecase`, `ic` ... ignorecase search [`noic`]
- `number`, `nu` ... displaying line numbers [`nonu`]
- `shell=path`, `sh` ... path to shell [`=/bin/sh`]
- `showmatch`, `sm` ... pair characters matching [`nosm`]
- `tabstop=n`, `ts` ... tab size [`=8`]
- `wrapscan`, `ws` ... search over end of file [`ws`]
- `wrapmargin=n`, `wm` ... right margin for wrapping [`=0`]

Default setting for `ex` and `vi`

Before editor start, commands for default setting are executed; order of execution:

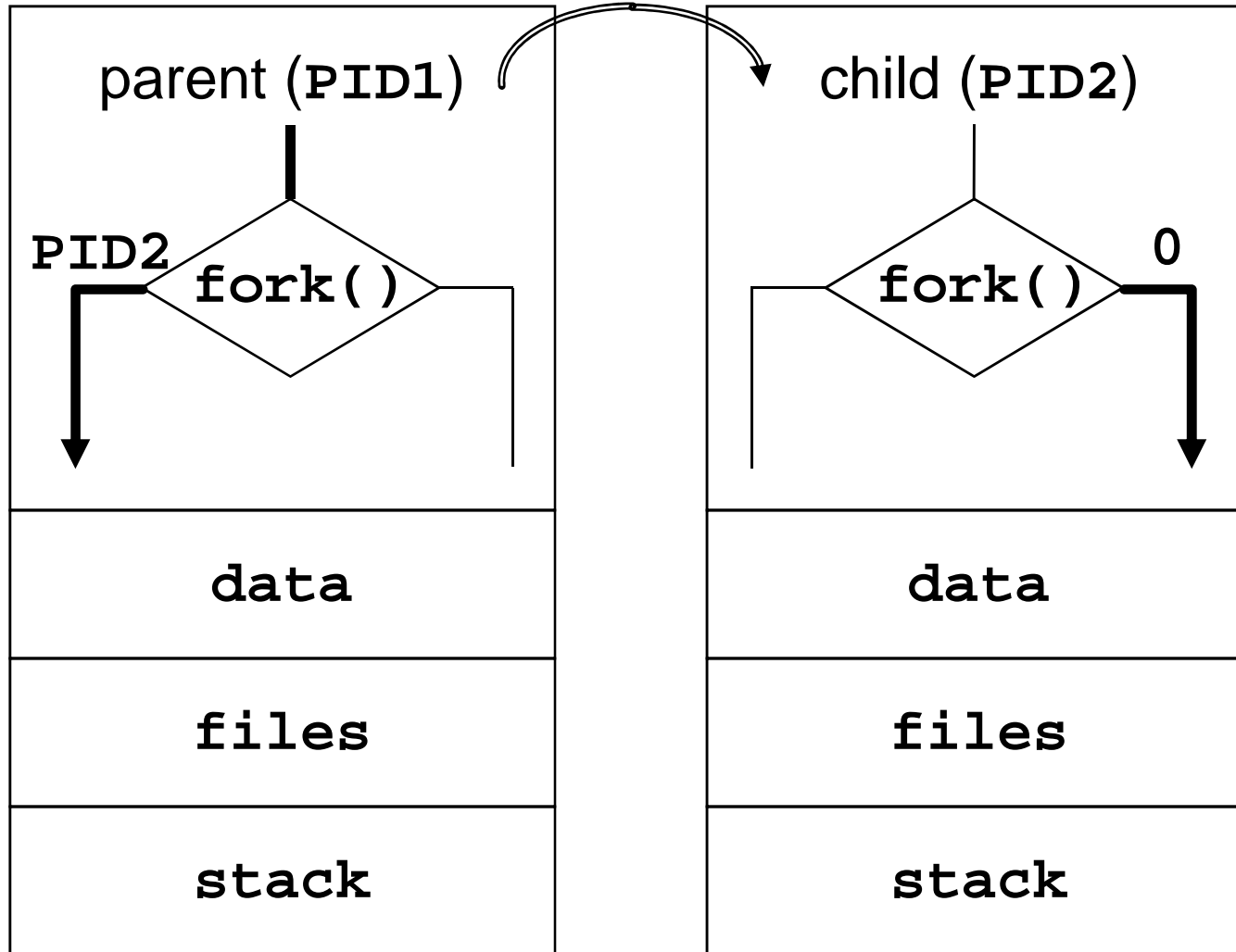
- the `EXINIT` variable
- script `.exrc` in home directory
- script `.exrc` in current directory
if option `exrc` is set (unset by default)

Commands are written like in `ex` (without colon).

Process

- running program ... (at least one) process
- process scheduling - priority
- list: `ps` command
- PID
- parent process \Rightarrow child process
- context of process
 - memory, files, environment variables,...
- communication
 - signals, pipes, sockets, shared memory,...
- return code (0..255)
- foreground/background run, daemon

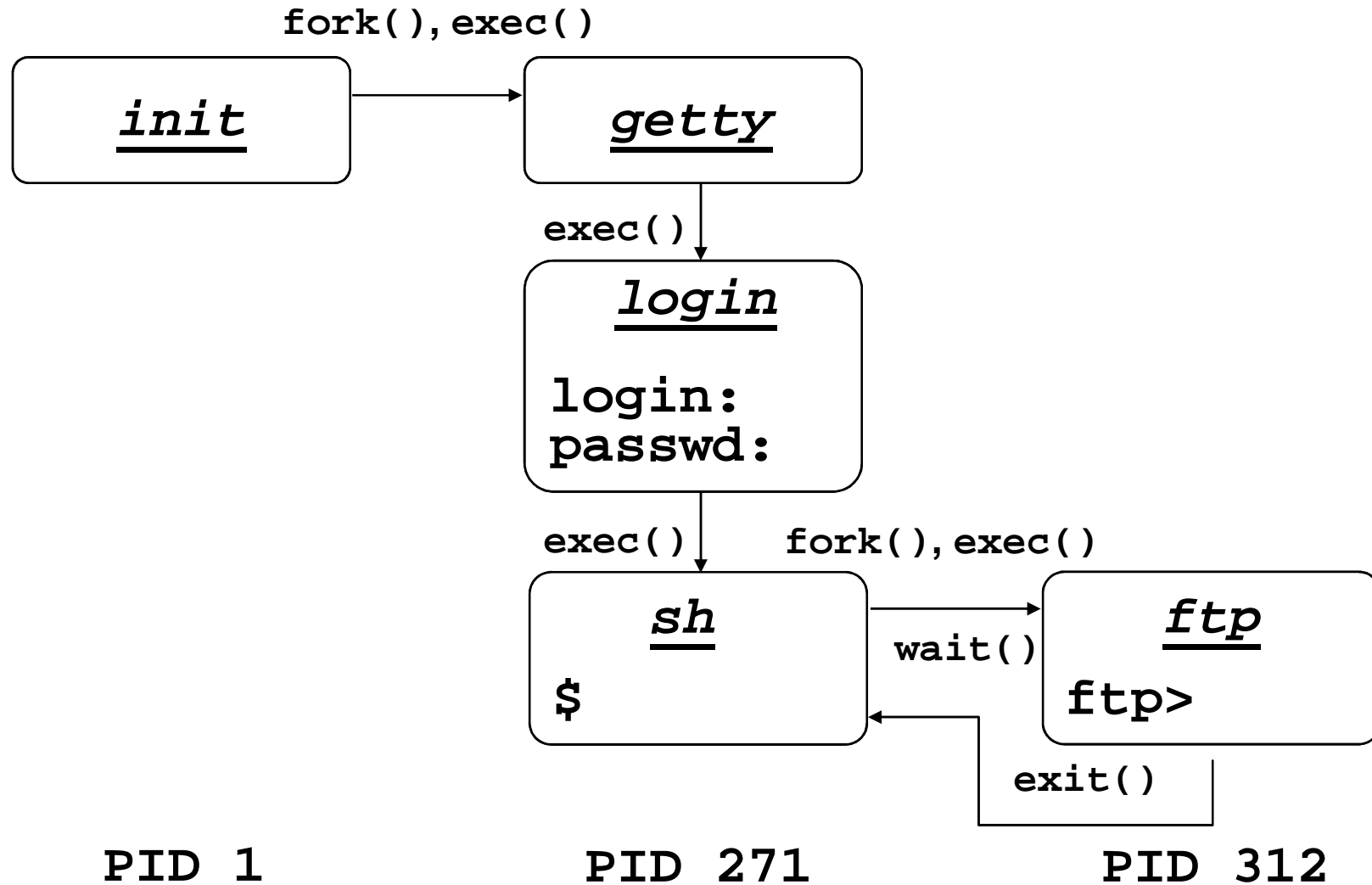
Process creation



Process control functions

- `fork()` ... creates copy of parent process;
“Cannot fork” error must be handled
- `exec()` ... overlaps the address space of the process
by given program
- `wait()` ... (parent process) waits for end of child
processes
- `exit()` ... terminates process and passes return
code to parent process

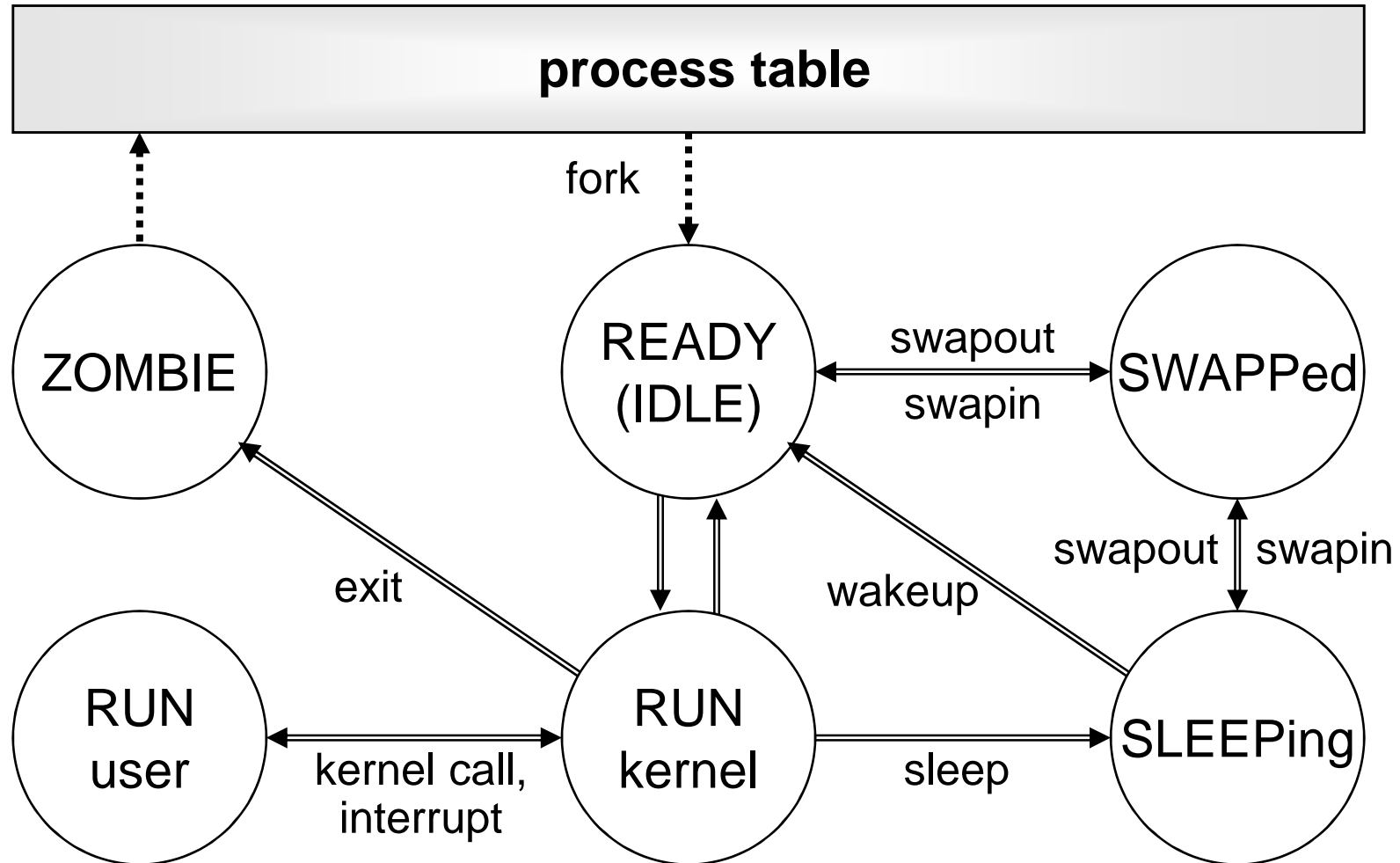
User session



Process context

- From user's point of view
 - program code, data, stack
 - open files
 - system (*environment*) variables
- From system point of view
 - general registers, program counter, processor status register, stack pointer, floating point registers, memory mapping registers...
 - memory allocated for process in user mode
 - kernel memory bound to process (e.g. process kernel stack)

Process states



Process priority

- One of the factors used by process scheduler
- Positive number (the higher one, the “nicer” process)
- Child process inherits parent’s priority
- It is possible to change priority when process started
`nice -n incr cmd`
- Increment used to be allowed within -20 to +20
- Only root can use negative values
- Running process priority change
`renice -n incr PID...`

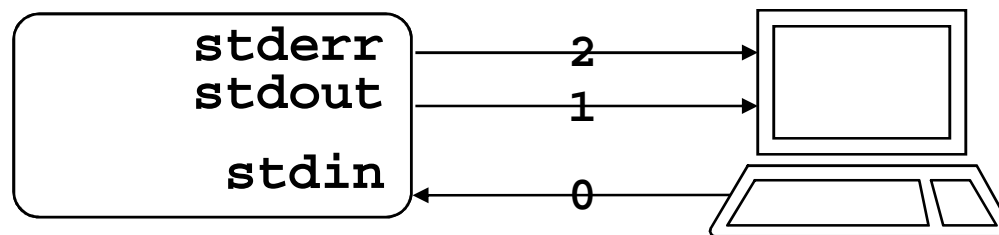
ps command

- **PID, TTY, STAT, TIME** a **COMMAND** of own processes

	<u>System V</u>	<u>BSD</u>	<u>POSIX</u>		
• process selection:	-e (<u>e</u> very)	-a (<u>a</u> ll users) -x (no tty)	-A (<u>A</u> ll)		
	-p <i>PIDs</i>	-t <i>ttys</i>	-U <i>users</i> -G <i>grps</i>		
• output content:	-l (<u>l</u> ong), ...	-l (<u>l</u> ong), ...			
	-o <i>key,...</i> (only given columns)				
	-O <i>key,...</i> (columns added)				
• sorting: (<i>PD program top</i>)	<table border="1"> <tr> <td>-r (<u>r</u>cpu)</td> </tr> <tr> <td>-m (<u>m</u>em)</td> </tr> </table>			-r (<u>r</u> cpu)	-m (<u>m</u> em)
-r (<u>r</u> cpu)					
-m (<u>m</u> em)					

Process and I/O

- access to input and output file through so called *file-descriptors*
 - 0 - standard input (`stdin`)
 - 1 - standard output (`stdout`)
 - 2 - standard error output (`stderr`)
 - ... - further files being opened



Inter-process communications

- sending signals
 - asynchronous control
 - information of type: event N occurred
- input/output through pipes
- System V IPC
 - semaphores
 - sending messages
 - shared memory
- BSD Sockets
 - sending messages, establishing streams
 - inside a system (files of `s` type), or
between a client and a server via network

Signal handling

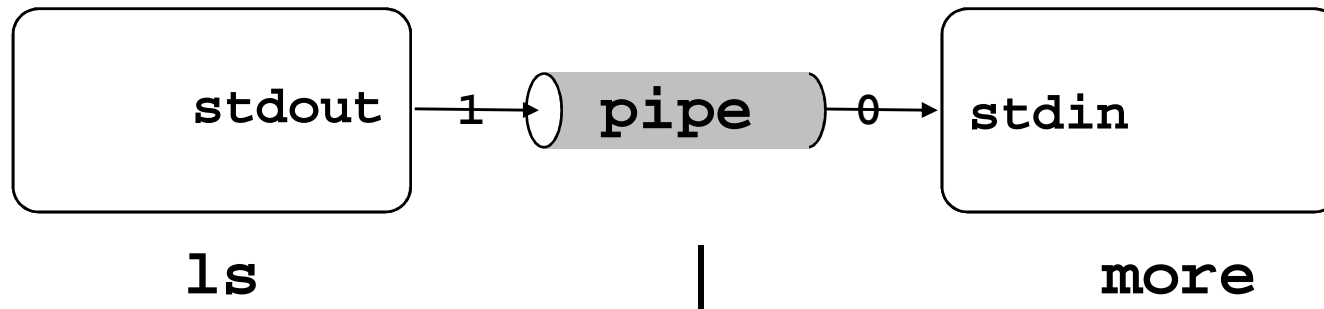
- signal sending:
 - command: `kill [-signal] PID`
 - function: `kill`
- signal trapping:
 - command: `trap [command] signal ...`
 - function: `signal, sigaction`
 - standard handlers: `SIG_IGN, SIG_DFL, SIG_ERR`
 - non-maskable signals: `KILL, STOP`
- list of signals: `kill -l`

Important signals

HUP(1)		program restart
INT(2), QUIT(3)		user interrupt (^C, ^\)
ILL(4)		bad instruction
ABRT(6)		abort function call
FPE(8)		error in arithmetic
KILL(9)	(non-maskable)	process termination
SEGV(11)		addressing exception
SYS(12)		bad system call
ALRM(14)		timer interrupt
TERM(15)	(maskable)	process termination (kill)
STOP(17), TSTP(18), CONT(19)		process stopping/resuming
CHLD(20)		child exiting
USR1(30), USR2(31)		user signals

Pipes

- in shell - binding input and output of two processes



- in program:
 - pipe executing a command: `popen`, `pclose`
 - pipe between (parent and) children: `pipe`
- permanent (named) pipes
 - included to filesystem (`p` type)
 - created by `mknode` function, or `mkfifo` command

System V IPC

- Every medium has unique ID
- Semaphores:
 - generalization of *P* and *V* operations [Dijkstra, Dekker]
 - *dead-lock* protection, process termination
 - functions: `semget`, `semop`, `semctl`
- Sending messages:
 - system creates communication channel
 - functions: `msgget`, `msgsnd`, `msgrcv`, `msgctl`
- Shared memory:
 - system adds requested area to process memory table
 - functions: `shmget`, `shmat`, `shmdt`, `shmctl`

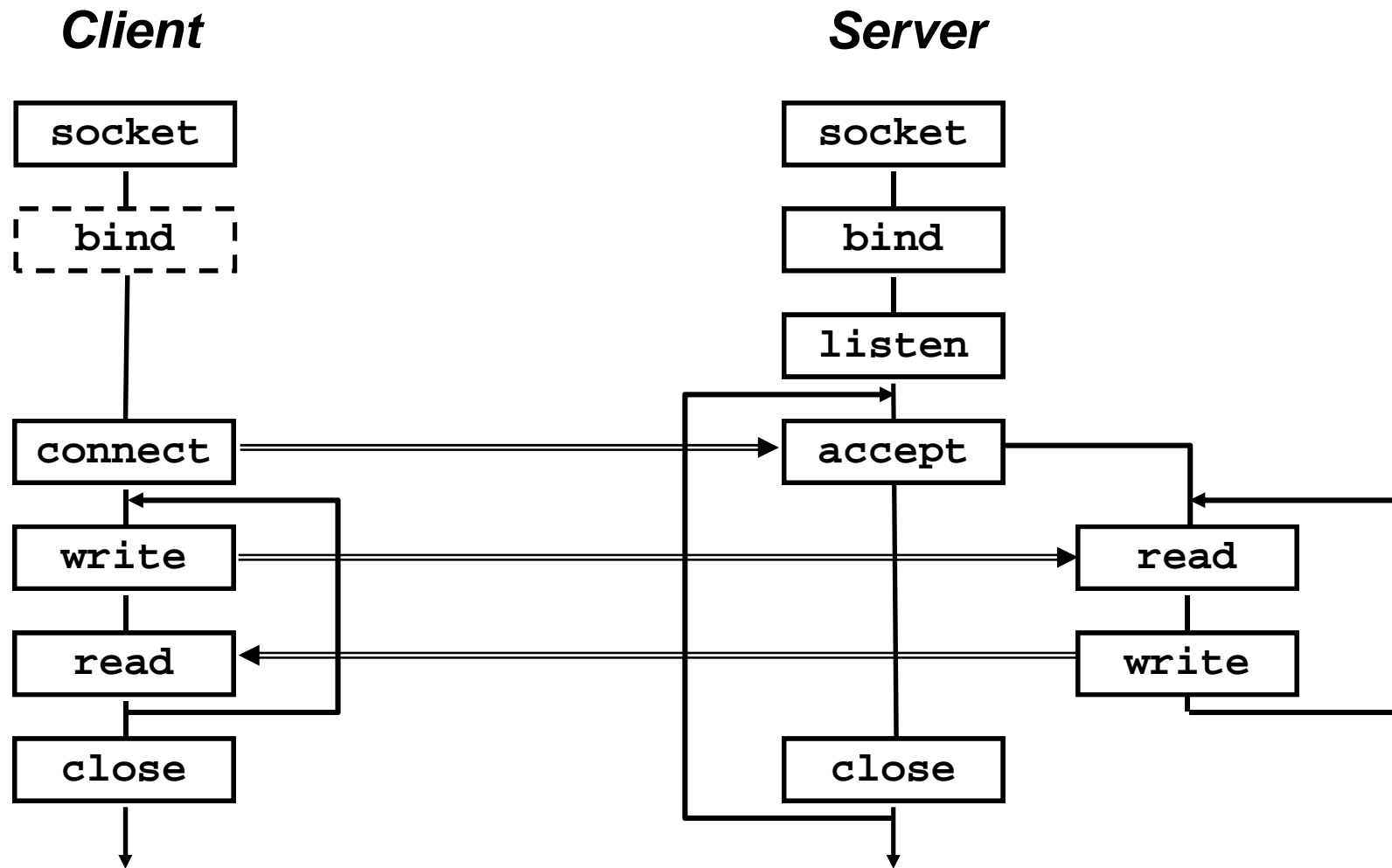
BSD Sockets

Socket - end of channel for client-server communication

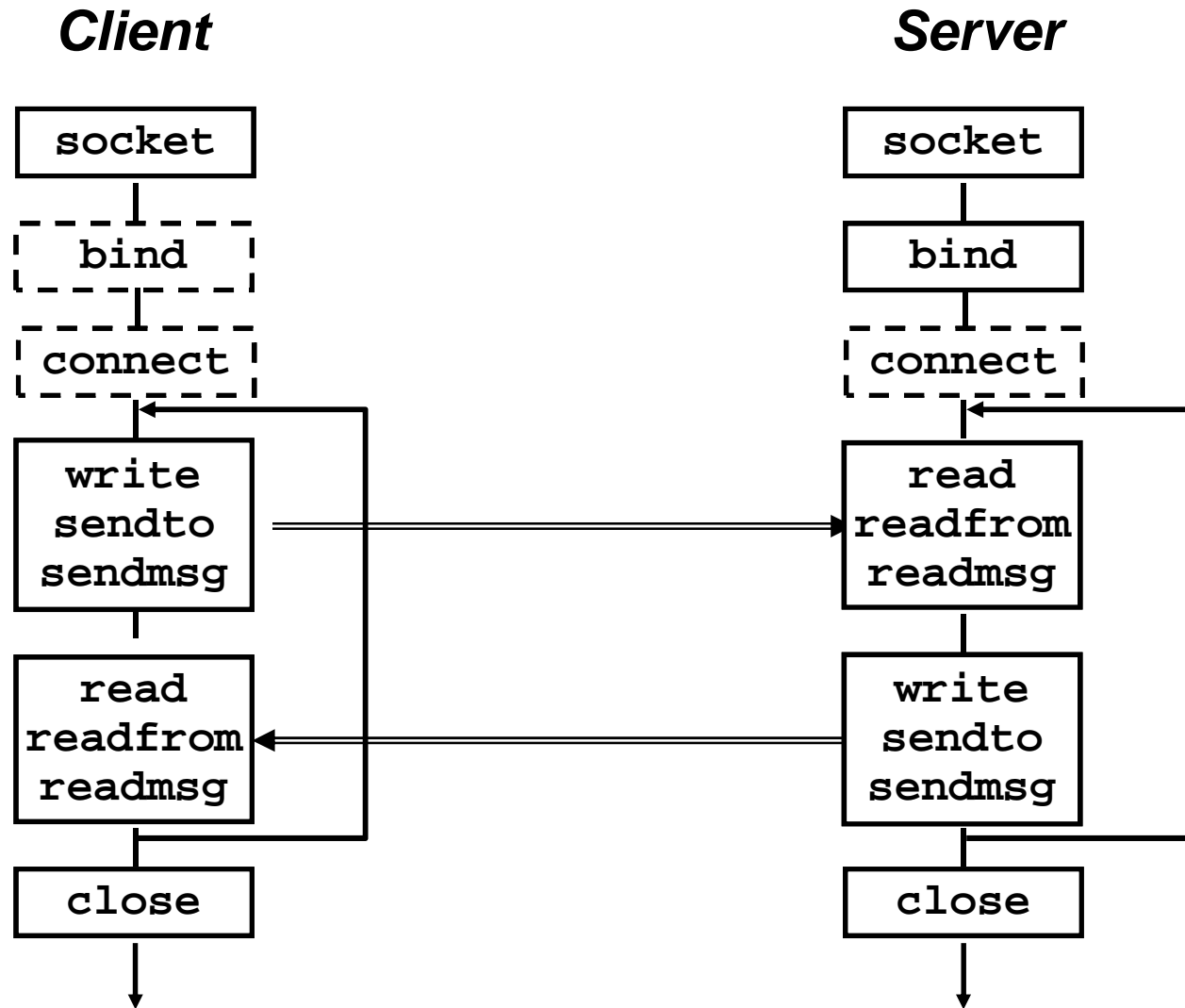
System functions:

- **socket** creates file descriptor according to
 - domain (*address family*): **AF_UNIX**, **AF_INET**
 - type: virtual circuit (*stream*), *datagram*
- **bind** assigns own address:
 - UNIX: name in filesystem (**s** type)
 - INET: IP address + port
- **listen** starts listening (e.g. sets queue length)
- **accept** (server) accepts request to channel from client
- **connect** (client) tries to make a connection to server

TCP application model



UDP application model



Start of network daemons

- **direct start**
 - within starting scripts
 - intensively used services, with complicated startup
- **indirect start** (on demand)
 - by `inetd` daemon
 - configuration in `/etc/inetd.conf`:

```
bootps dgram  udp wait   root   /etc/bootpd bootpd
tftp    dgram  udp wait   nobody /etc/tftpd  tftpd  /tftpboot
whois   stream tcp nowait  nobody /etc/whoisd whoisd
```

- reconfiguration: `kill -HUP PID`
- server communicates through file descriptors 0/1

Terminal

- user exploits system services by means of *terminal* - either real or *pseudoterminal*
- properties in `/etc/termcap` or `/etc/terminfo`
- terminal type in `TERM` variable
- terminal (re)initialization by `tset` command
- properties change by `stty` command
(e.g. `stty erase char`)
- access to own terminal through `/dev/tty`

Control characters

- some are configurable, others depend on shell
⇒ terminal and **TERM** accordance required
- usual default control sequences:
 - Ctrl+H - backspace
 - Ctrl+S - output stop
 - Ctrl+Q - output continuation
 - Ctrl+C - process termination (**SIGINT**)
 - Ctrl+\ - termination with dump (**SIGQUIT**)
 - Ctrl+D - end of input
 - Ctrl+Z - process suspension (**SIGTSTP**)
continuation: **fg** or **bg**

Shell

- basic program for communication with UNIX
- independent system component
 - Bourne shell, C shell, Korn shell
- reads lines and executes commands
 - own commands
 - programs stored in file system
- text preprocessor
 - meta-characters
 - variables
- program language & its interpreter
 - scripts

Basic built-in commands of shell

<code>:</code> [<i>arg...</i>]	- null command
<code>echo</code> [<code>-n</code>] <i>text</i>	- text output (with/without newline)
<code>printf</code> <i>fmt arg...</i>	- formatted text output
<code>pwd</code>	- current directory path output
<code>cd</code> [<i>dir</i>]	- current dir change (shell property)
<code>exit</code> [<i>rc</i>]	- shell exit (with return code)
<code>set</code> { <code>+</code> <code>-</code> } <i>opt...</i>	- shell options setting
<code>ulimit</code> [<i>limit</i>]	- work with user limits setting
<code>umask</code> [<i>mask</i>]	- work with new file mode mask

Formatting directives of `printf`

- General form: `%[flags][width][.prec]type`
 - `%c` ... print one char
 - `%s` ... print string
 - `%u`, `%d`, `%o`, `%x` ... print integer (unsign., dec., oct., hex.)
 - `%e`, `%f`, `%g` ... print real number
 - `%%` ... print percent sign
- Modifiers:
 - `%[-] width [.len] s` ... left alignment, maximum length
 - `%[+][0]width fmt-spec` ... sign enforced, leading zeroes
 - `%width [.precision] fmt-spec` ... real number precision
- Almost the same directives used in command in `awk` and function in C language

Meta-characters

- characters with a special meaning (e.g. *, >)
- special meaning may be suppressed (“quoted”) by
 - prefixing the char by “\” (so called *escape-sequence*)
 - enclosing to single quotes (suppresses all metachars)
 - enclosing to double quotes (keeps meaning of \$, ~, ", \)
- suppressed also:
 - <LF> ... do not execute command, just continue
 - space ... take several words as a single parameter
- take care namely in complex command parameters (e.g. `sed 's/ [0-9]*/ #/' ...`)
- comment: ... *#comment*

Shell patterns

Word with shell pattern meta-characters is replaced by list of all filenames matching the pattern.

- * - matches any string of characters
- ? - matches any single character
- [a-f0-9] - matches any character from the list
- [!a-z] - matches any character not on the list

Whitespaces have to be escaped by \.

Chars !,], - can be used same way like in regexps.

Expansion done by shell !

Expansion does not include leading dot in filenames,
does not cross directory border.

Shell variables

- name=value* - assigning a value
- name=value cmd* - setting just for *cmd* execution
- \$name, \${name}* - value expansion (substitution)
- \${#name}* - substitution for value length

Identifier - alphanumeric characters, case sensitive.

Variables have only text value.

Substitution of unset variable - empty string.

Output of variable value: **set**, **echo "\$name"**

Local and *environment* variables.

Child process (subshell, pipe) inherits only *exported* variables (by **export** *variable*).

Child cannot change variables of its parent!

Environment variables

- IFS** - Internal Field Separator,
default: IFS=<space><tab><LF>
- PS1, PS2** - prompt, continuation prompt
- PATH** - path: list of dirs with executable files
(current dir not included by default!)
- CDPATH** - path for `cd` command
- TERM** - terminal type
- SHELL** - running shell
- LOGNAME** - logged user name
- HOME** - user home directory
- MAIL** - user's incoming mailbox file

Conditional variables substitution

syntax	result if <i>name</i> variable is	
	defined	undefined
<code>\${name:-value}</code>	<code>\$name</code>	<i>value</i>
<code>\${name:=value}</code>	<code>\$name</code>	<i>value</i> +assigning <i>name=value</i>
<code>\${name:+value}</code>	<i>value</i>	""
<code>\${name:?value}</code>	<code>\$name</code>	"" +echo <i>value</i> and exit

Command files - scripts

- “direct” call (rights **+rx**):
script params
- call by shell (rights **+r**):
sh [*options*] *script params*
- code sourcing (runs in the same shell process, not as a new process):
. script
- the first line may define interpreter and options:
#! *abs_path_to_interpreter* [*options*]
- login startup scripts (sourced):
/etc/profile, .profile

Positional and special parameters

- `$n`** - n -th parameter (of script), $n \leq 9$
- `$#`** - number of parameters (of script)
- `$0`** - script name
- `shift [n]`** - shift positional parameters ($\$2 \Rightarrow \1)
- `set [--] text`** - reset positional parameters
 - e.g.: `set a + b` \Rightarrow `$1=a, $2=+, $3=b, $# = 3`
 - `IFS=:; set $PATH` \Rightarrow `$1=/bin, ...`
- `$*`** - all positional parameters as text
- `$@`** - all params, but "`$@`" is "`$1`" "`$2`" ...
- `$?`** - return code of last command
- `$$`** - current shell PID
- `$!`** - last background process PID

Command input redirection

syntax	redirects command input ...
<i>cmd < file</i>	... from <i>file</i>
<i>cmd << str</i>	... from shell input (shell script text); input processed like text in double quotes e.g.: <pre>ed xxx << END \${line_number}d ← <i>here document</i> END</pre>
<i>cmd << \str</i>	ditto, processed like text in single quotes e.g.: <pre>ed xxx << \END 1,\$d END</pre>
<i>cmd <<- str</i>	ditto, text may be indented (by tabs) e.g.: <pre>ed xxx <<- END 1,\$d END</pre>

Command output redirection

syntax	redirects ...
<code>cmd > file</code>	standard output to <i>file</i>
<code>cmd 2> file</code>	standard error output to <i>file</i> e.g.: <code>rm xxx 2> /dev/null</code>
<code>cmd >> file</code>	standard output to end of <i>file</i>
<code>cmd 2>> file</code>	standard error output to end of <i>file</i>
<code>cmd 2>&1</code>	standard error output to standard output, attention to redirection order: <ul style="list-style-type: none">- <code>grep xxx file > \$log 2>&1</code> both outputs go to <code>\$log</code> file- <code>grep xxx file 2>&1 > \$log</code> output goes to <code>\$log</code>, error to output

Command lists

- *cmd1* | [`<LF>`] *cmd2*
 - pipe between commands
e.g.: `ls -l *.c | wc -l`
- *cmd1*; *cmd2*
 - sequence of commands
- *cmd1* || [`<LF>`] *cmd2*, *cmd1* && [`<LF>`] *cmd2*
 - conditional sequence of commands
e.g.: `rm aa && echo File aa removed`
- { *cmd1*; *cmd2*; }
- compound statement
- (*cmd1*; *cmd2*)
 - run command(s) in subshell
e.g.: `(cd wrk; rm *)`

read command

- Command `read var` reads line from input to variable
- Command sets return code (success may be tested)
- In case of more parameters it assigns one-by-one fields from input line to variables (rest of line to the last one); field separators defined in `IFS`; subsequent whitespace separators grouped; *as-is* reading can be forced by `IFS= ' '`
- Character `\` in input is interpreted as quoting (escapes field separator, but `<LF>`, too!); may be suppressed by `-r` option
- When called from prompt it reads from terminal, can be redirected (`read var < file`), as well as terminal read can be enforced (`read var </dev/tty`)

read command examples

- `echo -n "Enter number: "; read x`
... reads answer
- `IFS=: read user x x x name x < /etc/passwd`
... reads login and full name of (1st) user
- `LHOST=ss1000.ms.mff.cuni.cz`
`echo $LHOST | cut -f1 -d. | read SHOST`
... does nothing (SHOST set in child process)
- `echo $LHOST | cut -f1 -d. > /tmp/x.$$`
`read SHOST < /tmp/x.$$`
`rm /tmp/x.$$`

Command output substitution

...`cmd`... - inserting output of *cmd* call into line text

- př.:

```
SHOST=`echo $LHOST | cut -f1 -d.`
```

↓

```
SHOST=ss1000
```
- command runs in a subprocess of the same shell
- the last LF is removed
- attention to nested call
 - inner back quotes (and slashes) have to be “quoted”
 - solution: storing to temporary variables
 - since **ksh** added new syntax `...$(cmd)...`
- e.g.:

```
rm `cat files`  
vi `grep -l '^\\.\\.\\.\\.\"' man8/*.8`
```

Control structures

```
if cmd  
then cmds  
[elif cmd  
  then cmds]  
[else cmds]  
fi
```

```
case word in  
pat1 | pat2 )  
  cmds;;  
*)  
  cmds;;  
esac
```

```
{while|until} cmd  
do  
  cmds  
done
```

```
for var [ in text ]  
do  
  cmds  
done
```


test command

- call: `test condition` or `[condition]`
- when true exits with return code 0
- attention to unset variables, spaces etc.:
 - right: `[-n = "$x"]`
 - wrong: `[-n = $x]`, `[-n="$x"]`
- logical operations (with unconditional evaluating):
 - conjunction: `cond1 -a cond2`
 - disjunction: `cond1 -o cond2`
 - negation: `! cond`
 - subexpression: `(cond)`

attention - quoting in shell needed

test command operators

- e *file* - *file* exists
- f *file* - *file* is regular file
- d *file* - *file* is directory
- L *file* - *file* is symbolic link
- r *file* - current user has **r** access right to *file*
- w *file* - current user has **w** access right to *file*
- x *file* - current user has **x** access right to *file*
- s *file* - *file* exists and is not empty
- z *string* - *string* is empty
- n *string* - *string* is not empty
- str1* = *str2* - string equality (really equality: $\$x = a^*$!)
- str1* != *str2* - string inequality
- int1* -eq *int2* - number equality (-ne, -lt, -le, -gt, -ge)

expr command

- call: `expr opndA op opndB ...`
- outputs text result and exits with return code
- logical operators: `=`, `<`, `>`, `<=`, `>=`, `!=`
- arithmetic operators: `+`, `-`, `*`, `/`, `%`
- string operators (in SUS only “:”):
 - `string : regexp` (anchor to beginning by default!)
 - `match string regexp`
 - `substr string pos len`
 - `length string`
 - `index string chars`
- attention to meta characters
- newer shells have arithmetic directly: `$((...))`

Control structures - `if`

Example: `if [-d tmp]; then
 echo directory exists
elif mkdir tmp; then
 echo directory created
else
 echo cannot create directory
fi`

Comments:

- A pipe can be used as the tested command.
- Command result may be negated: `if ! cmd`
- If the command produces output, it has to be handled:

```
if echo "$x" | grep ... > /dev/null
```

Control structures - case

Example: `case $1 in
-h | -\?) echo "Usage: ..."; exit;;
' ' | *[^0-9]*)
echo "No number entered"; exit;;
*) NUMBER=$1;;
esac`

Comments:

- Labels are formed by *wildcards*, however without the special meaning of dot and slash (no expansion made).
- Label order is important (sometimes it may be used to compensate missing negation or regexps).
- Variables (even tested ones) may be used in labels.

Control structures - `while`, `until`

Example: `while` read line; `do`
 case \$line in
 \#*) **continue**;;
 *) \$line;;
 esac
`done` < script

Example: `i=1; until` mkdir /tmp/\$i; `do`
 `i=`expr $i + 1``
`done`

Example: `while` [\$# -gt 0]; `do`
 case \$1 in
 -n) N=\$2; shift 2;;
 -n*) N=`echo \$1 | cut -c3-`; shift;;
 *) **break**;;
 esac
`done`

Control structures - `for`

Example: `list=MFF,FF,FaF,FTVS`
`for file in *; do`
 `case , $list, in`
 `*, $file, *) cp $file ${file}_bak;;`
 `esac`
`done`

Comments:

- Loop from 1 to n (`seq` is not in SUS):

```
for i in `seq 1 $n`; do
i=1; while [ $i -le $n ]; do i=`expr $i + 1`
i=:; while [ ${#i} -le $n ]; do i=:$i
```

- The `for` loop is not suitable for file reading:

```
for line in `cat file`
```

Example: input file reading

- `n=0`
`while read x < file; do`
 `n=`expr $n + 1``
`done`
 ... reads infinitely first line
- `n=0`
`cat file | while read x; do`
 `n=`expr $n + 1``
`done`
 ... the `n` variable is set only in child
- `n=0`
`while read x; do # < file`
 `n=`expr $n + 1``
`done < file`

Example: pipe output reading

- ```
n=0; find ... | (while read x; do
 n=`expr $n + 1`
done
echo Found $n files
)
```
- ```
... | while read x; do
    printf "Delete $x? (y/[n]) "
    read z
    case $z in
        ' ' | n* | N* ) continue;;
    esac
    rm $x; n=`expr $n + 1`
done
```

... the **z** variable is read also from the file
- ```
read z < /dev/tty
```
- ```
{ ... read z <&3 ... } 3<&0
```

Functions

Function *name* definition:

```
name( ) {  
    statements  
}
```

- runs in the same process
- variables are global, function can change them!
- call + parameters same as other commands call
- parameters are accessed via `$#`, `$1` etc.
(positional parameters are local, no change of caller ones!)
- function exits with return code of last command,
can be changed using command `return val`
- priority: functions, built-in commands, external programs
built-in commands can be forced by `command cmd`
- functions are not inherited into subshells

Line processing steps

Line is parsed from left to right in following steps:

1. breaking up to atoms (words) by operators
2. control structures and operators detection
3. redirection operators detection
and variables definition
4. variables and command substitution
- 5*. breaking substitution result by chars in **`$IFS`**
- 6*. shell patterns expansion
7. quote removal

*Steps 5 and 6 are not executed when setting variables.

Re-parsing of input line

`eval arg` - re-parsing and executing line made by concatenating all the arguments with spaces

- example: `read login x uid x < /etc/passwd`
`eval UID$uid=$login`



UID0=root

- using indirect variables (array compensation)

- example: `eval echo \$$#`

- using value of the last parameter

Process control

cmd & - execution in background

wait - waiting for background process exiting

... since *cs***h** more sophisticated management (*jobs*,...)

exec cmd - call of *exec*() with *cmd* command
(shell changes to given program)

... since *ks***h**, *exec* can be used for current shell file
descriptors redirection (e.g.: *exec 3<&0*)

Signal handling in shell

- Handler setting: `trap [cmd] sig...`
 - *sig*: number/name of signal or 0/EXIT
 - *cmd*: handler (executed within current process)
- Child process cannot handle signals masked off by parent.
- Masking signals off: `trap " " sig...`
- Default handler resetting: `trap sig...`

Shell options

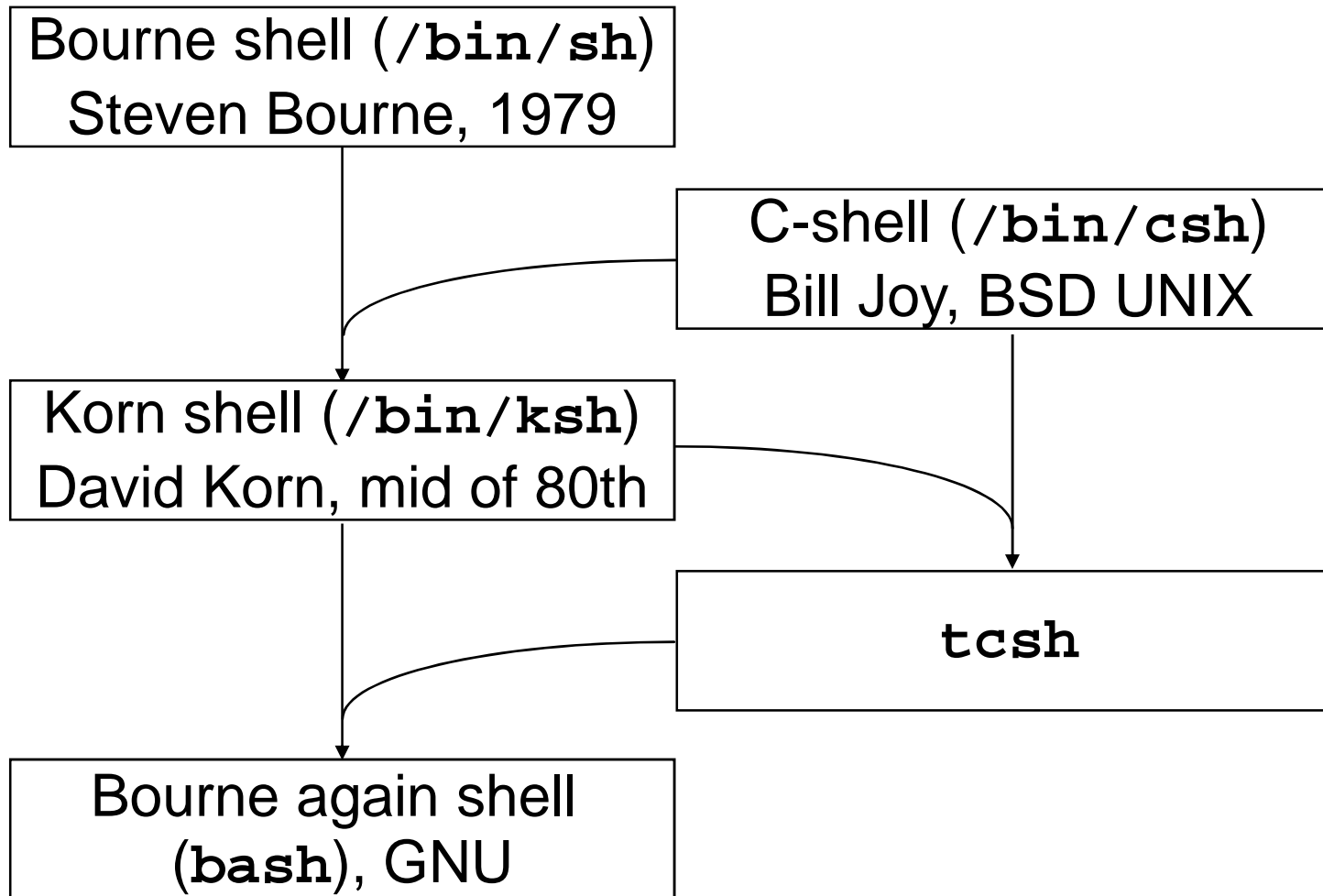
Shell options can be set

- on command line when starting shell
- on the first line of script
- by **set** command

Most important options:

- a** ... all variables are exported
- C** ... do not overwrite existing files with redirection
- e** ... stop shell when error occurs
- f** ... disable shell pattern expansion
- n** ... read commands but do not execute them
- u** ... expanding unset variable is error
- v** ... shell input lines are written to standard error
- x** ... executed cmds are written prior execution

Shell evolution



C-shell

Basic differences:

- `.login`, `.cshrc` ... startup script
- `set var=str`, `env`, `setenv`, `@ var expr` ... variables
- `foreach`, C-like expressions and commands
- `>&`, `>>&`, `|&` ... standard error output redirection
- `␣<` ... direct input from terminal

Features adopted (and modified) by successors:

- `~[user]` ... home directory
- `<ESC>` ... filename completion
- `history`, `![[-]n]`, `![[?]str]` ... command history
- `alias name str` ... command aliasing
- `pushd`, `popd` ... directory stack for `cd` command

Korn shell

- `cd old new, cd - ...` path change, undo `cd`
- `VISUAL, set -o ed ...` history with line editing
- `\` resp. `<Esc><Esc>` ... filename completion
- `FPATH` ... path for functions
- `*()`, `+`, `?`, `@()`, `!()` ... regexp-like shell patterns
- `${var#pat}`, `${##}`, `${%}`, `${%%}` ... `$var` trimmed by min.(max.) string matching pattern from start (end)
- `[[]]` ... internal `test` (`<`, `>`, `-nt`, `-ot`, `-O`, `-G`)
- `let var=exp, (())` ... arithmetic
- `${v[e]}`, `${#v[*]}`, `v[e]=s`, `set -A v str ...` arrays
- `select, getopt, typeset`

Options parsing (getopts)

```
while getopts :x:y NAME; do
    case $NAME in
        x )  opt_x=$OPTARG;;
        y )  opt_y=1;;
        \? ) echo "Unknown option $OPTARG" ;;
        : )  echo "Missing value of $OPTARG" ;;
    esac
done
shift `expr $OPTIND - 1`
```

Time handling utilities

- running command with time keeping:
`time command`
- process suspending:
`sleep seconds`
- output of current (or another*) date and time:
`date [+format]`

Format (same as C `strftime()`): text with %-directives

- `aAbB` ... short/long day/month name
- `dmyYHMS` ... (numeric) date and time
- `uUVjC` ... nr. of week-day, week, year-day, century
- `cxX` ... “normal” date and time format
- `s` ... seconds since “epoch” (1.1.1970) *

Synchronization

- If two processes share some resource, it is necessary to avoid concurrent approach to *critical sections* by a lock.
- File based synchronization: program tests the *lock* file; if it exists, resource is locked, process waits in loop (`sleep` !) and when the lock file disappears, the program creates new one by itself.
- Testing and re-locking must be *uninterruptible* operation from the operation system view, e.g. `mkdir`, or redirection (`>`) when `-C` is set.
- After leaving the critical section, the file must be removed; it is necessary to handle all exceptional cases (`trap` !). For the case of post-mortem check, the lock should be marked by PID.

Batch processing

- Running a command with **HUP** and **QUIT** signals blocked and output sent to **\$HOME/nohup.out**
nohup command
- Running a command at given time (user must be allowed to use it in files **at.allow** or **at.deny**, command output is mailed to user):

at { **-t** *mmddHHMM* | *time* [**+***incr*] } *command*

The command can list (**-l**) and remove (**-r**) jobs.

- Scheduled regular running by **cron** daemon:

crontab [**-l**]

Record example:

0 1 * * 1-2,5 /usr/sbin/backup

awk filter

- Aho, Weinberger, Kernighan
- language similar to C, differences:
 - LF has significant meaning
 - easier working with strings
 - interpreted language
- dialects: **awk**, **nawk**, **gawk**
- call:

```
awk [opt] { -f script | pgm } { params | file | - }...
```
- filter parses records (lines) of given input files and executes awk-script commands on them
- example:

```
ls -l |  
awk '/^-/' { s += $5 } END { print s }'
```

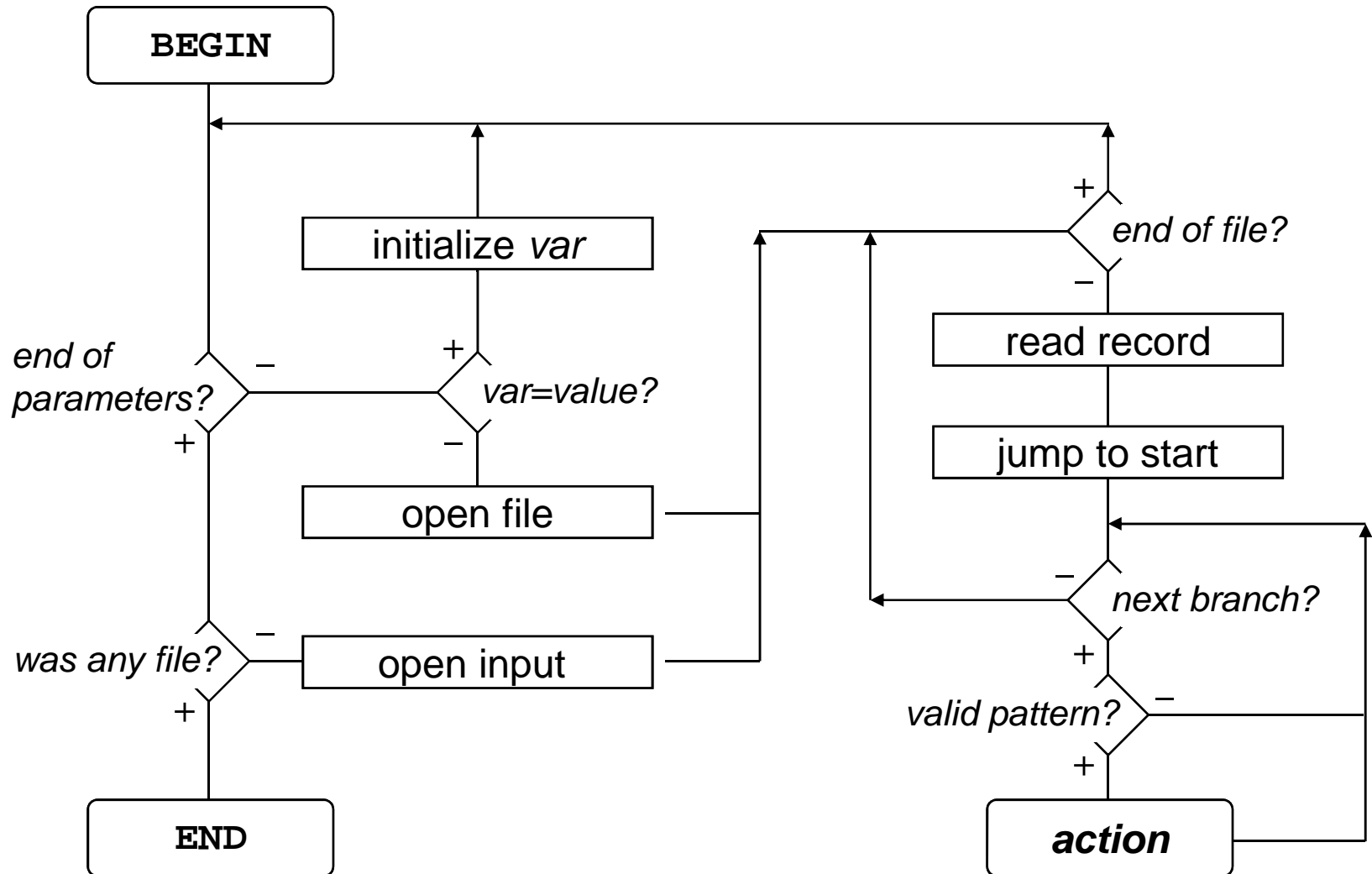
Patterns and actions (**awk**)

- Program (awk-script) is a series of branches in form
pattern { action }
- Pattern types:
 - BEGIN** executed once, at the beginning of work
 - END** executed once, at the end of work
 - / regexp /* executed on every matching record
 - expression* executed when condition is true
 - pat1 , pat2* executed since the time *pat1* is valid,
 until *pat2* is valid
- Default pattern: execute action on every record
- Default action: print record

awk program example

```
BEGIN    { procs=0; lines=0 }
/procedure/ { procs++; print;
            lines=1; level=0; next }
! lines { next }
        { lines++ }
/begin/  { level++ }
/end/    { level-- }
/end/ && ! level {
        print "Lines:", lines; lines=0 }
END      { print "Procedures: " procs }
```

awk control flow diagram



Extended regular expressions (**awk**)

Added (changed) meta characters

- exp^+ , $exp?$... repeating (>0 , ≤ 1)
- $exp1 | exp2 | exp3$... alternatives
- $(,)$... subexpression grouping

Meaning clarification

- $^$, $\$$... beginning and end of tested string

Missing meta characters (comparing to basic regexps)

- $\backslash <$, $\backslash >$, $\backslash \{$, $\backslash \}$, $\backslash ($, $\backslash)$, $\backslash n$

Regexps must be written as a literal (it is not possible to match with literal stored in variable)!

Records (**awk**)

- By default, a line is the record
- Record separator is stored in the **RS** variable and can be changed to another char: **RS="char"**
 - e.g. for HTML: **RS("<")**
- Special separator - empty line: **RS=" "**
- Separator change has effect to next record parsing
- Ordinal number of record: **NR** variable
- Output record separator (string, written at the end of **print** command work): **ORS=string**

Record fields (**awk**)

- Input record is parsed and broken into fields
- Number of fields: **NF** variable
- Individual fields accessible via “variable” $\$number$
- Number can be written as expression, e.g. $\$(NF-1)$
- Attention to difference between **NF** and $\$NF$!
- Entire record can be referenced as $\$0$
- Record fields can be altered, however, the exact form of the original record is lost (separators disappear)!

Field separator (**awk**)

- Field separator is stored in **FS** variable
- Can be set up from command line by **-Fsep** option
- Separator can have following forms:
 - space, then any whitespaces sequence breaks field
 - non space char, then any char occurrence breaks field
 - (**nawk**) regexp, e.g. a line `a==b`
 - has three fields, if **FS="="**
 - has two fields, if **FS=="=="** or **FS=="+"**
- Separator change has effect to next record *
- Parameter separator of **print** command: **OFS=string**

Basic syntax rules of `awk`

- The `awk` language is **line oriented**
- Commands are separated by semicolon or LF, entire command must be (usually) on a single line
- Line continuation is marked by backslash at the end of previous line
- Exceptions:
 - after condition of `if` and `while` commands
 - after commas, opening braces (“{”)
 - after `&&` and `||` operators
- Comment: any text beginning by “#” up to end of line

Constants, variables (**awk**)

- Constants
 - common arithmetic constants
 - strings are delimited by double quotes
 - *escape* sequences: `\b`, `\f`, `\n`, `\r`, `\t`, `\ooo`, `\xxx`
- Variables
 - have only text values
 - text is converted to number in arithmetic context
 - are initialized
 - associative arrays (string is index): `var[item]`
 - (**nawk**) special *member* operator: `item in var`

Expressions (**awk**)

- arithmetic operators:
 - common C-operators: +, -, *, /, % (modulo)
 - power: ^
 - assignment operators, in(de)crement: =, +=, ... , ++, --
- concatenation operator: space (!)
 - e.g.: "File: " FILENAME " opened"
- relational and logical operators (result is 1/0):
 - common C-operators: <, >, <=, >=, ==, !=, !, ||, &&
 - operator *match* (with regular expression written as literal, not as variable) and its negation: ~, !~
e.g. test, whether 2nd field starts by dot: \$2 ~ /^\. /
- (**nawk**) conditional expression: *cond ? then : else*

Basic commands (**awk**)

- `{cmd1;cmd2}` ... compound statement
- `if(cond)cmd[;else cmd]` ... conditional statement
- `while(cond)cmd` ... loop statement
- `do cmd;while(cond)` ... loop statement
- `for(init;cond;step)cmd` ... loop statement (*step* expression evaluated after each iteration)
- `for(var in array)cmd` ... loop statement (repeating loop body for each index, in random order!)
- `break, continue` ... exit loop, next loop iteration
- `next` ... end of current record processing
- `exit` ... end of program (jump to END branch)

Output commands (**awk**)

- **print**
printing whole record ended by ORS (LF by default)
- **print *str1*, *str2*, ...**
printing strings separated by OFS (" ") ended by ORS
- **printf *fmt*, *par1*, *par2*, ...**
formatted printing
- **print,printf > *filename***
output to file (maximum 10 opened files !)
- **print,printf >> *filename***
output to end of file
Example: `printf "%s::%d:\n",
grp, gid >> "/etc/group"`

Library functions (**awk**)

- mathematical functions: **int**, **exp**, **log**, **sqrt**
- (**nawk**): **sin**, **cos**, **atan2**, **rand**, **srand**
- string functions:
 - **index(s,t)** ... returns position of *t* in *s* or **0**
 - **length(s)** ... returns length of string *s*
 - **split(s,var,sep)** ... splits *s* to words by *sep* separator and assigns them to *var* array items; returns number of items; example: `split("194.50.16.1",ip,".")`
 - **sprintf(fmt,...)** ... returns formatted text as string
 - **substr(s,pos[,len])** ... returns substring starting at *pos*
- (**nawk**): **match**, **close**, **sub**, **gsub**
- (**gawk**): **tolower**, **toupper**, **strftime**

Own functions (**nawk**)

- **function** *name*(*parameter-list*) {
 statements
}
- **return** *expression*

- functions defined among branches
- order is not significant
- own function “library” : **awk -f lib -f script ...**
- variables are global, parameters local
- called within expressions
- not all parameters need to be entered

awk program configuration

- Input parameters via **echo** and standard input:

```
e.g.: echo $LOW $HIGH | awk '  
NR == 1 { low=$1; high=$2;  
FS=":"; next }  
... ' - /etc/passwd
```

- Shell variables substitution:

```
e.g.: awk /"$RE"/
```

- Initialization of variables from command line:

```
e.g.: awk var=value1 file1 var=value2 file2
```

- Environment variables (**nawk**): **ENVIRON** array

```
e.g.: file = ENVIRON["HOME"] "/log"
```

Built-in variables (**awk**, **nawk**)

- **RS**, **ORS**, **NR**, **FS**, **OFS**, **NF**
 - **FILENAME** - currently processed file name
example: `FILENAME == "-" { ... }`
-
- **FNR** - ordinal number of record in current file
 - **ARGC**, **ARGV** - number of parameters, values array
 - semantics like in C language
 - awk-script and options are not includedexample: `{ ARGV[ARGC++] = "file" }`
 - **SUBSEP** - dimension separator in array index
 - **RLENGTH** - length of string matched by `match()`

Communication with system in `awk`

- environment variable change: impossible !
 - `PATH='awk '{print path}'``
 - `eval `awk '{printf "PATH=%s;HOME=%s", p, h}'``
 - `awk '{print path; print home}' | {
 read PATH; read HOME; ...
}`
 - `{ read PATH; read HOME; } << EOF
`awk '{print path; print home}'`
EOF`
- system command call (**nawk**): **system(*command*)**
 - example: `system("rm " filename)`
 - function returns command return code, not output !
 - command runs in subshell !

getline command, pipe (nawk)

- **getline** [*var*] [<{ "-" | *filename*}]
reading new record from current input, from standard input,
or from other file to fields \$0, \$1, ... or to *var* variable
e.g.: `getline < "/etc/hosts"`
- **command | getline**
command (*pipe*) output reading
e.g.: `"pwd" | getline dir`
- **print | command**
printing to pipe
e.g.: `printf "Job %d ended", id | "mail " adm`
Maximum number of open pipes: 1 !

C language - files

<code>*.c, *.cpp</code>	source files
<code>*.h</code>	header files
<code>*.o</code>	compiled (<i>object</i>) modules
<code>a.out</code>	default name of compiler result
<code>/usr/include</code>	system header files root
<code>/lib/lib*.a, .so</code>	system libraries

C language - compiler

Call: `cc [options] file...`

Important options:

- `-o filename` output file name
- `-c` compile only (do not link)
- `-E` preprocess only (do not compile)
- `-Olevel` optimization level
- `-glevel` debugging level
- `-Dmacro` define preprocessor macro
- `-Umacro` undefine preprocessor macro
- `-Ipath` path to `#include` (header) files
- `-llib` use linkage library `llib.a`
- `-Lpath` path to linkage libraries (`-llib`)

Predefined macros

Besides standard ones (`__DATE__`, `__FILE__`, `__LINE__`, `__cplusplus`, etc.), following macros are defined in UNIX:

<code>unix</code>	always defined in UNIX
<code>mips, i386,...</code>	hardware architecture
<code>__osf__,...</code>	operating system clone
<code>SunOS</code>	operating system version
<code>_POSIX_SOURCE, _XOPEN_SOURCE, _ANSI_C_SOURCE</code>	compiling according to particular standard

Macro definitions output: `cc -dM -E file`

make program

- command generator
- SW project management
- example (file **Makefile**):

```
program: main.o util.o
    cc -o program main.o util.o
main.o: main.c program.h
    cc -c main.c
util.o: util.c program.h
    cc -c util.c
```

- compiling and linking proper modules:
make [program]

Input file syntax (**make**)

- target dependency: *targets : [files]*
- executed commands: <**Tab**>*command*
- comment: *#comment*
- line continuation: *line-beginning\
line-continuation*

Macros (**make**)

- macro definition:
name = string
- undefined macros are empty
- order is not significant
- cannot redefine
- command line definition:
make target name=string
- macro usage:
\$name, \${name} or *(name)*
- environment variables are macros

System administration

- Basic tasks:
 - installation (OS, SW packages)
 - configuration (filesystems, users, services, ...)
 - system backup
 - system monitoring (`syslog`, `cron`,...)
- In general, the tasks on various UNIX systems are similar, however, special admin tools vary quite a lot, even in case of the same vendor.

Start of system

- First, `init` process is started, it then controls system operation.
- BSD systems startup:
 - script `/etc/rc` („run control“)
 - scripts called from `/etc/rc` (e.g. `/etc/rc.local`)
 - configuration `/etc/rc.conf`
- System V startup:
 - script start is driven by run level and configuration file `/etc/inittab`
 - scripts are collected to directories `/etc/rc#.d`
- Current system usually uses some combination

Runlevels, `inittab`

- Selected on boot, or by `init level` call
- In details, they can slightly differ, however usually
 - 0 ... means system stopping
 - 1 ... means single-user mode
 - 3 ... means full user mode
- Configuration file `inittab`:
`l3:3:wait:/sbin/rc default`

Startup scripts

- Classic system:
 - for runlevel `#` in `/etc/rc#.d`
 - names: `S##service` and `K##service`
 - order given by number
 - script calls another script from `/etc/init.d` with parameter `start` or `stop` respectively
- Current systems typically use some variation; starting order is deduced by system itself due to dependency definitions in the scripts

The End