< fin - input is in file
> fout - create/overwrite file
>> fout - append to existing file
comm1 | comm2 → pipe

[a-zA-Z] [0-9]
^ - line beginning, $ - end of line
\< - start of word, \> end
* - 0 or more appartitions
\+ - 1 or more appartitions
? - 0 or 1 apparitions
\s - blank character
(abc) - group, | - or

grep - q (quiet)
- -c: dear nr de linii
- -v: only those not matching
- -n: display matching lines + the line no.
- -r: recursively
- -i: case insensitive
- -x: match exactly the entire line
- -o: only the part that mat.

sort - r: reverse
- -u: uniq
- -M: compares lines

sed - f: from a file
- p: print | d - delete
- i: insert (before)
- a: append (after)
- s/regex/string/flag(opt)
- \n

awk - if, for, while = C
- BEGIN{ } / END{ }
- NF - no. of fields $1, $2...
- NR -
- $0 - entire line
- FS - field separator trophy
- length(string)
- split (string, array, separ.)
- index(s1, s2) ⇒ strstr

wc - l: lines
- -w: words; -c - characters

shell #!/bin/bash
- chmod +x file
- if ...; then ...; fi
- $ var → access the value
- for var in ...; do ... done
- " " - interpreted (spec. chars)
- ' ' - luate ca atare
- $"var1" = $"var2"
- test -lt, -le, -gt, -eq, -ge, -ne
- test -z (string), -n string

shell $((expr)) regex
`command` - executes command and replaces it with the result of the execution
shift - shift arguments to the left
$# - argc, $@ - argv

UNIX - interactive, time-sharing, multi-user, multi-tasking
KERNEL - nucleum (codul nuc) loaded at the PC startup
System CALL: Kernel mode to user mode for access to disk, network, video mem.
KERNEL MODE = level 0 protection
USER MODE = level 3 protection
levels 1, 2 = services

FILE TYPES: regular, directory, hard/soft links, FIFO, socket
SYMBOLIC LINKS: ln -s target name
- behave as the target file/directory
- can reference on another disk
- delete file - points to a file that doesn't exist
HARD LINKS: ln target file
- new i-node for the same f.
- have to be
- file is deleted only when there are no more links to it
ROOT (superuser): BOSS

METODE DE ALOCARE A MEM
① SINGLE TASKING:
- one single program, to open another one, we need to close this one
② MULTITASKING:
Ⓐ Real:
a) absolute: i) fixed partitions
- ⊕ more procs
- ⊖ the max dim of proc
- ⊖ program compiled on a specific partition
ii) partiții relocabile: the addresses are stored as OFFSETS inside a partition
b) dynamic partitions: fragmented memory ⇒ memory needs to be compressed (compactată?) (close processes): all of it or as much as needed for one process
↳ SOLUTION: paged memory

Ⓑ VIRTUAL: a) paged
- virtual addresses are generated in the form of (page, offset), each proc. has a table of pages
address: offset + pf * dim-page
b) segmented: code + data segments grouped in
- the access to the seg. is controlled by the OS, the seg. can be reentered (entered from another program)
⊖ fragmenting
c) paged + segmented ♥
- segments are divided into more virtual pg. the proc. has a table of segments, each of which has a table of pages

LOADING POLICIES
① all pages in the beginning
- ⊕ fast access at runtime
- ⊖ slow startup
- ⊖ loading pages that are not necessarily needed
② each page when needed
- ⊕ fast startup
- ⊕ loading only what is needed
- ⊖ slow access at runtime

LOCATION PRINCIPLE | the pages neighbouring the page that has just been loaded are more likely to be needed in the future, so we load them preemptively

REPLACING POLICIES
- how to choose the victim pg
FIFO: each pg. has an age, the oldest ones go first
NRU: each pg. has two bits (r, w) that are set to 1 at r/w and the OS is periodically setting them to 0 ⇒ classes (0,0), (0,1), (1,0), (1,1)
↓ first used
LRU - last recently used - matrix of N×N, N = no of pg; when accessing pg. i, line i is populated with 1, column i — 0; pages with the lesser 1s are first to go

PIPE | pipe (int p[2])
0 - read, 1 - write

FIFO | mkfifo (char * name, int mode)
empty - read blocks until no more data
full - write block until all data is read → space
O_NDELAY → ret 0 if fifo empty / full
no process to read/write:
- O_NDELAY ⇒ error
- normal ⇒ wait
- procs need not be related

STATES OF A PROCESS
Hold →(1) Ready ⇄(2, 3, Swap) Run →(8) Fin; Run →(5) wait →(6) Ready
1st - create proc, 2 - get CPU, 3 - release CPU, 4th - move on disk, release CPU, 5 - i/o op. wait for resources, 6 - op. from 5 done, free disk space (deallocate memory), 8 - release CPU, free mem, destroy process
** - alloc. mem
* - free mem.

PLACEMENT POLICIES
- how to handle malloc?
- FIRST FIT:
  - ⊕ fast ⊖ no attempt to control fragmentation
- BEST FIT
  - ⊕ economic in terms of large memory areas/chunks
  - ⊖ slower ⊖ creates small areas of free memory ⇒ fragmentation
- WORST FIT
  - ⊕ leaves behind larger chunks
  - ⊖ slower
- BUDDY → always alloc. a chunk of the smallest power of two, greater than the required size; keep lists of free chunks by powers of 2; alloc. the largest available chunk and distribute the remaining memory in pow of 2 chunks
  - ⊕ speed

WAIT - suspends the exec until a child finishes (any, or a specific one)
EXEC - replace the whole image with the given command
EXIT - exits the current proc and goes back to the parent

DEADLOCK
- proc. thread enters an ∞ waiting state because it needs a resource that is held by another

CAUSES OF DL
1. Mutual exclusion when accessing resources
2. Hold and wait: block a res. while waiting for another
3. Non-preemption: resources cannot be "stolen"
4. Circular wait

SOLUTION TO DL
- reboot - pick a victim proc → create a backup state

DETECTION OF DL
graf: ○ proc, □ res.
→ is held by, - - → is waiting
cycle ⇒ deadlock

PREVENTION OF DL
→ pick an order for resources and always lock in that order

ZOMBIE ⇒ process that has finished execution but still has an entry in the proc table; lost wait to avoid it or before creating: signal(SIGCHLD, SIG_IGN)
ORPHAN = still executing, parent is dead

SCHEDULING ALGO
① FIRST COME, FIRST SERVED
② SHORTEST FIRST:
- ⊖ we need to know durations
- ⊖ not always right
③ Priority Based
- ⊖ starvation
④ Deadline Scheduling: Scheduling based on the finish time (it needs to finish f...)
⑤ Round Robin: time sharing, procs are running simultaneously, in READY, one → R...

SEMAPHORE (value with two operations)
P(S) - wait for pr(A)
V(S) --
if (V(s) < 0)
  state(A) = wait
  q(s) ← A / A w...
  pass ctrl to sys
else
  pass ctrl to A

SEMAFOR (value, queue)
V(S) - SIGNAL for A:
V(S)++
if(V(S)<=0)
q(S) → B
// B comes through state(B) = Ready
pass ctrl to Sys. scheduler
else
pass ctrl to A

Process vs. Thread
P contains one or more T
⊕ quicker to create T, P
⊕ quicker to switch between T
⊕ T share data easily
⊖ no security between T
∃ one T blocks ⇒ all T in that P block

UNIQ - -c: counts
-u: displays

UNIX filesystem structure

Block	Content
0	Boot block ⊕
1	Super block ↓
2	Inode blocks ⊗
n	
n+1	Data blocks

⊕ a kind of map for...

FOR I-NODES addr
- 1-10: for the first 10 blocks of 512 bytes of the file
- 11: indirectare simplă (the next 128 blocks of 512 bytes)
- 12: indirectare dublă ($128^2 * 512$)
- 13: indirectare triplă ($128^3 * 512$) (B|A)^n

CACHE ① Direct Cache: store the page in location page_addr % cache_size
⊕ fast ⊕ simple
⊖ cache trashing (collisions)

② Associative Cache place each page on a free location found through iterating the cache ⊖ slow

---

CACHE ③ Set Associative Cache
- hash table; cache org in sets of pages
- the set of a page is calc as page_addr % cache_set, then a free location is found iterating through the set
⊕ flexibility ⊕ avoid col.
⊖ doesn't use all available cache lines

SIGNAL
- sets a function to be called for a certain signal (RUTINA DE TRATARE)
- does not signal anyth.

MOUNT associate a storage device to a particular location in the directory tree

int dup(int descr_vechi)
int dup2(int d_veche, int d_nouă)

Read la FIFO: waits until there is no proc open for writing or until there is data; returns the read data or 0 if ∄ a proc to write

---

int open(char* name, int flag, [int rights])
O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_TRUNC, O_NDELAY (FIFO)
read/write(handle, buff, nr_bytes)

WAIT → RUN?
↳ program loaded into mem waiting for CPU (OS scheduler decide)

RUN → WAIT
↳ additional cond.

How many threads to process 1M files? A: the number of available cores (cores of the processor)

Why do I/O ops cause a proc to move from state RUN to Wait?
A: The proc has to pause itself while it gets / sends the data through the I/O ops and it frees the proc in the meantime for other procs.

How is the address calculation done in the absolut fixed pool?
A: multiply pred. part. and the RAM locations are set. first prog, first portion, first add - beginning of part

---

Line containing a and not b
grep "a" filename | grep -v b

---

fifos
A → X → B → Y → C → Z → A
if(a==0){
open("X", O_RDONLY);
open("Z", O_WRONLY);
}
if(b==0) ... X,4
if(c==0) ... Z,4

---

2 sets of 4 vs (4 sets of 2)

---

at least 3 vowels
grep -E '^(.*[aeiou]){3}.*$'

---

4 rules that specify the no of rep
+ → 1≤
* → 0≤
? → 0/1
{n} - n times

FIFO for ~~reading~~ writing; proc?
∞ wait, until a prog opens the ~~read~~ fifo.

---

every open pipe → DON'T forget to close the pipe

Hard-link only in the same partition → when creating a hard link, the OS creates a new directory entry that points to the existing inode of the file; it works seamlessly in the same partitions bc. the inode is unique in that part; the OS cannot assign that inode no. to a file a diff. part bc. part share unique inodes within them.

---

at least x vowels/numbers
grep -E '^([^aeiou]*[aeiou][^aeiou]*){n,}$'

---

having a group
grep -E '^([^:]*:){4}[^:]*\(ion\)$'

---

an odd no of chars
grep -E '^(..)*.$' a.txt

---

swap neighbouring digits
sed -E 's/([0-9])([0-9])/\2\1/g'
d1 d2 swap global

---

multiple of binary 4
grep -E '.*([01]{2}){2}$' | grep -E '^[01]{5,}$'
binary multiple of 4
5 digits
100
1010