COMENZI

CUT → cut -b -5 ia primele 5
→ cut -b 5- ia de la chr 5
→ cut -d: -f1,3 fields 1 și 3 cu delim ":"

FIND merge în toate subdirectoarele
- -maxdepth 1 merge în cel curent
- -type f/d caută fișiere/dirs
- -name "ceva" caută după nume

UNIQ -u dă toate fără duplicate
-c numără duplicatele

SORT -n sortează numeric
-r reverse

GREP grep -E " " < fis

. orice chr
\\. chr punct
[abc] oricare din ele
[a-z] range
\d o singură cifră
[^abc] orice nu e printre ele
? cel mult odată
* de 0 sau mai multe ori
+ de 1 sau mai multe ori
\\< început de cuvânt
\\> final de cuvânt

-c count
-i ignore case
-v invert match
-e multiple patterns
-n line number
-R caută recursiv

( ){ }
↳ expresie

SED sed -r "s/de înlocuit/înlocuitor/g"

ca să am o porțiune pun ([ ]) apoi pot înlocui cu \1\2...
sed '1d' < file (prima linie); sed '1,5d' < file; → 1-5 delete inclusiv sed '2,4,6d' < fis (în afară de ele), sed '$d' < file (ultima linie); sed '1d; $d' < fis (first and last)

AWK awk '{print $1}' < fis

$0 - toată linia
length($0) lung linie

→ FS=":" schimbă field separator în :
→ se aplică linie cu linie
→ ex: awk 'BEGIN{FS=":"; sum=0}
{if($1>$2){sum+=$3}} END{print sum}' < fis

DUP/DUP2

DUP: int dup(int fd)
face o copie a handler-ului (returnează file descriptorul) pt. chestia pe care vreau să o duplic.
ex: int x=dup(myfifo) => x - nou handler legat de myfifo

DUP2 int dup2(int old, int new)
copiază handler-ul de la old la new
ex: dup2(p[0],0) pipe-ul suprascrie standard input

scriu în consolă, nu pipe:
int fd=dup(1) → dup2(fd,1)

RANDOM

→ un hard link poate fi creat doar între fișiere din același file system (nu e valid să arăți spre un i-node din alt file system având legătură cu securitatea. fiecare partiție are file system propriu)
→ 0: standard input; 1: standard output; 2: error
→ **FISIERE**: -d director, -f file, -r readable, -w writable, x executable, -s size>0, -e există -n not empty
→ **NUMERE** -eq, -ne, -gt, -lt, -ge, -le
→ **STRINGS** =, !=, -z (zero size), -n (non zero size)
→ **BOOLEENI** (pt if): !, -o (or), -a (and)

PROCESE

FORK() la apelul fork() pid-ul copilului e returnat în părinte, iar copilul returnează 0
→ la eroare returnează -1

SIGNAL()
signal(SIGINT, f) ^C tastatură, funcția noastră
SIGCHLD: child stopped or terminated

```
ex: void f(int sgn)                int main()
① { wait(0); }                   ② { signal(SIGCHLD, SIG_IGN)
  int main()                          ..
  { signal(SIGCHLD,f);            }
   .. }
```

împiedică să apară zombie

EXECLP(L)
dacă e cu succes șterge tot ce e după și afiș output al ei, altfel nu face nimic
! Toate variabilele din copil sunt independente. La fel sunt și cele din părinte declarate după crearea copilului. ! pipe() 0 - citire 1 - scriere

THREAD-URI

→ e mai ușoară comunicarea între threaduri decât procese

MUTEX

```
pthread_mutex_t m;
pthread_mutex_init(&m,NULL);
pthread_mutex_lock(&m);
pthread_mutex_unlock(&m);
pthread_mutex_destroy(&m);
```

BARRIER

```
pthread_barrier_t barr;
pthread_barrier_init(&barr,NULL,nr_așteptări);
pthread_barrier_wait(&barr)
pthread_barrier_destroy(&barr)
```

! Folosesc același stackframe și zonă de mem. cu procesul spre deosebire de procese, care își creează stackframe nou.

READ/WRITE

-> return EOF dacă nu are ce citi/unde scrie și nu e deschis capătul celălalt
-> scrie/citește cât poate și returnează nr bytes citiți (dacă are ce/unde puțin)
-> așteaptă altfel

-> before return wait până e deschis pt READ

FIFO
```
int fd1 = open("abc", O_WRONLY);
int fd2 = open("bca", O_RDONLY);
```
!Le blochez ÎN ORDINE! mkfifo("nume", 0600)

SEMAFOARE

-> sem. binar = mutex
-> metoda P (decrementează)
also called wait, sleep or down
-> metoda V (incrementează)
also called signal, wake-up or up

SISTEME DE FIȘIERE

i-node: punct de intrare în fișiere
Cât de mare poate fi un fișier? -> dim adresă A
Câte adrese încap într-un bloc N?
$10\cdot N\cdot A + N\cdot N\cdot A + N^2 N A + N^3 N A$ -> dim max fiș

!

A - dim. adresă
N - câte adrese încap într-un bloc
B - dimensiune bloc adică $B = N\cdot A$
avem
- bloc dim B
- conține A adrese (N-ul din teorie)

aflăm dim unei adrese (A-ul din formulă)
$B = A\cdot Q \Rightarrow Q = \frac{B}{A}$ dim unei adrese

$\Rightarrow \underbrace{5\cdot N\cdot A}_{\text{directă}} + \underbrace{N\cdot N\cdot A}_{\text{simplă}} + \underbrace{N^3\cdot N\cdot A}_{\text{triplă}}$ (ca teoria)

(înlocuim $N = A$, $A = \frac{B}{A}$)

$5\cdot A\cdot\frac{B}{A} + A\cdot A\cdot\frac{B}{A} + A^3\cdot A\cdot\frac{B}{A} = 5B + AB + A^3B$

SHELL

`$*` toate args, `$#` nr. args, `$?` cod retur ultima cmd.
`$(( ))` pt. op. matematice

-> for x in $* \n do \n done
-> var=$(( )) sau var=`grep ...`
-> if [cond] \n then \n fi
-> #!/bin/bash

exemple:

①
```
#!/bin/bash
for arg in $*
do
    if [ -x $arg ]
        echo $arg
    fi
done
```

③
```
#!/bin/bash
nr_fis=$(find -type f -name "" | wc -l)
nr_dirs=$(find -type d | wc -l)
med=$((nr_fis/$((nr_dis+1))))
echo med
```

②
```
#!/bin/bash
FILES=$(find -maxdepth 1 -type f -name "*.txt")
for FILE in FILES
do
    NR=`grep -E "\<cat\>" $FILE | wc -l`
    if [ NR -ge 1 ]
    then
        echo $FILE
    fi
done
```

④
```
for arg in $*
do
    if [ $((arg%2)) -eq 0 ]
    then
        echo $arg
    fi
done
```

⑤
```
cnt=0; sum=0
for arg in $*
do
    cnt=$((cnt+1))
    if [ $((cnt%3)) -eq 0 ]
    then
        sum=$((sum+arg))
    fi
done
echo sum
```

⑥
```
for arg in $*; do
    if [ ! -f $arg -a ! -d $arg ]
    then
        echo $arg
    fi
done
```

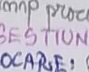
CURS 12

① - faza de prezentare -> hold (până când am resursele necesare
-> Ready -> run (când am procesor) -> finish -> elib. resurse
- pendulează între ready și run (luptă procesor)
- wait: dăm jos de pe procesor p.-a lăsa celelalte procese să își facă treaba până când primim datele de care avem nev. (citește hard-ul/consola). Apoi merge în ready. Ținem procesor degeaba pt. că discul e lent (de nu știe când)
- swap: nu e destul spațiu pt. a rula un program, mutăm unele pagini dintr-un proces în swap (acolo merge lent)

② PLANIFICAREA PROCESELOR:
- primul venit primul servit
- shortest job first (starvation pt. task-uri mari); clientul trb. să aproximeze o durată - priorități (nobotul)
- deadline scheduling: task/durată/termen
- round Robin: se alocă fiecărui proces câte o cuantă de timp procesor (prioritățile ar primi mai multe cuantums)

③ GESTIUNEA MEMORIEI (alocare/înlocuire/plasare)
ALOCARE: REALĂ
- sisteme single tasking ①
- sisteme multi-tasking
  - partiții fixe -> absolute ② / relocabile ③
  - partiții variabile ④

VIRTUALĂ: - paginată ⑤
- segmentată ⑥
- segmentat-paginată ⑦

⑤ RAM


-> fragmentare rezolvată, dar calcul adr. complicat
-> avem nev. de o tabelă de pagini a fiecărui proces
-> AE -> adresă virtuală (PV, offset)

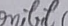
⑥ -> nu adresează fragmentarea
-> grupează codul și datele în seg. cu protecție la acces
-> calcul adresă: tabela de segm.; AE -> virtuală (S, offset)

⑦ AE -> virtuală (S, PV, offset)

POLITICI DE ÎNCĂRCARE
① toate la început (pornire lentă, încărcăm date de care posibil să nu avem nevoie, dar după merge repede)
② încărcăm când e nev. (merge lent)
③ principiul vecinătății (referă o pagină -> încarc și vecinii)

POLITICI DE ÎNLOCUIRE
① FIFO: nu tare bine
② NRU (not recently used): victima cea mai mică clasă nevidă
0: 00 (necitite și nescrise recent)
1: 01 (scrise recent)
2: 10 (citite recent) ~ sunt resetate periodic
3: 11 (citire și scrise recent)
③ LRU (least recently used): considerăm că avem n pag. în RAM, întreținem o matrice de $n \times n$ biți -> când pagina k e accesată populăm cu 1 linia k și o coloana k -> alegem victima pagina cu cea mai mică sumă a liniei

POLITICI DE PLASARE
-> cu 2 liste înlănțuite pt. spațiul alocat și free, căutăm prin ele unde plasăm malloc-ul (fragmentare iar)
-> cum evităm $\boxed{\text{fragmentarea}}$!
① first fit: prima zonă free destul de mare (paralel cu fragmentarea, dar e rapid)
② best fit: cel mai mic spațiu suficient de mare (lent, fragm. foarte mărunță)
③ worst fit: cel mai mare sp. disponibil (ok. fragmentare)
④ buddy fit: alocăm cea mai apr. putere de 2

CACHE-URI:
- direct: pagina k merge în cache pe partiția $k \% \overset{\text{câmpuri}}{c}$
-> coliziuni care conduc la trashing
- set: pagina k e pusă pe prima poz. liberă (lent)
- set-asociativ: organizăm cache-ul în grupuri de pagini, det. grupul cu % apoi căutăm în grup:

k -> grup $k \% 4$
c = 8 seturi de 2 pg

DEADLOCK:
ce e?: un ciclu în graful de alocare a resurselor
prevenire: blocăm resursele în aceeași ordine

HARD LINKS - SIMBOLIC LINKS
-> simbolic: 2 files point to the same i-node. If we delete the files, the links point to smth that doesn't exist. file sys.
-> hard: 2 i-nodes point to the same data; when we delete the file, the data remains and we also need 2 delete the other links

sed 'id' < fișier (prima linie); sed '1,5d' < fișier; sed '$d' < fișier (ultima linie); sed 'id'; sed < p5 (5 linii [illegible] last) | [illegible] director de proces, care își creează stack/nume nou.

DUP/DUP2

```
int p[2];
int fd=dup(1);
pipe(p);
dup2(p[1],1);
dup2(fd,1);
printf("asta");
```

} undo la dup2

printf("asta"); acum scrie în consolă, nu pipe

AWK

```
awk '{print (($1+$2))}' < fisier
```

VARIABILĂ DE CONDIȚIE

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER
pthread_cond_wait(&cond, &mtx)
pthread_cond_signal(&cond)
pthread_cond_destroy(&cond)
pthread_cond_broadcast(&cond)
```

} împreună (una trimite la toate, una nu)

RANDOM

-> diviz. cu 2, binar: un 0 la final

-> utilizatori activi: `ps -ef | grep -E "^.... [0-9]{4}" | awk '{print $1}' | sort | uniq -c | sort -n -r`

-> are x, nu are y

`"^[^y]{0,}[x]{1,}[^y]{0,}$"`

-> folosesc threaduri un multiplu de nr. de core-uri

-> BEGIN: do all computation from the code block following this keyword BEFORE parsing through the lines of the file

END: do all computations from the code block following this keyword AFTER parsing through the lines of the file

FS: ":" change the field separator to ":"

->popen: pt. a primi sau trimite informații la/din comenzi shell

C

FILE* fopen(calea, modul)

↓

r, w, a

fișiere binare: open(cale, O_RDONLY)

write/read(fd, &c, sizeof(type c))

SHARED MEMORY

-> identificați printr-un nr. unic

-> IPC (inter-process communication)

-> ipcs: vezi toate ipcs existente

-> ipcrm: șterge un ipc

-> shmget(key, size, flags): creează sau utilizează un sm existent (un id la return)

-> shmat(...): atașează shmr unui pointer

shmdt(...): detașează memoria de la un pointer