CLI: SISTEME DE OPERARE (NR. 3). 16.10.2008.

APELURI SISTEM UNIX

- se împart în două categorii:
  - care returnează int (în caz de eroare, returnează -1)
  - care returnează pointer (în caz de eroare returnează null)

int errno : variabilă globală folosită pt. a depista erorile int.

errno = 0 => succes
≠ 0 => eroare

void perror (char *c) : afișează mesajul asociat variabilei eroare

ÎNCEPEREA PROCESELOR

- dp.d.v. al unui programator, procesul începe cu funcția main().
- main : funcție apelată de modulul start

int main (int argc, char **argv, char ** env)

env : environment

- în UNIX main poate avea 3 argumente.

env : listă de șiruri de caractere de forma var = val.

ultimul pointer este null.

TERMINAREA PROCESELOR

- orice proces emite un cod de retur (între 0 și 255)
- emiterea unui cod de retur se face prin apelul funcției:

```
void exit (int k);
exit (0);   s-a terminat normal
exit (1);
   ;
```

dacă nu se face un exit(0) explicit, funcția care apelează pe main face automat un exit(0)

main apelează pe f, f pe g, g pe h, h pe main

return k din main = exit(k) doar dacă main este pe primul nivel de iterație.

CREAREA PROCESELOR

int fork() : singurul mod în care programul poate crea procese copil.

```
a = b;
fork();
c = d;
```

: creează un nou proces f. asemănător cu procesul deja creat.
procesul execută imediat c=d
execuția copilului începe după fork
codul de retur al lui fork: -1 = eroare (nu s-a creat proces copil)
>0 procesată, pid copil
=0 proces copil.

```
int pid;
pid = fork();
if(pid <0) {
            /* err */
          }
if(pid >0) {
            /* tata */
          }
else {                    (pid = 0)
     /* copil */
     }
```

int wait(int *st) - așteaptă terminarea procesului copil
- în codul de retur pid
-1 : err (∄ procese copil)

errno ECHILD

Dacă s-a terminat procesul copil, iar procesul tată n-a făcut față, procesul intră în stare zombie.

st : parametru de ieșire, unde funcția wait scrie felul în care s-a terminat procesul copil.

```
int *st
wait(st)
```
} greșit ⇒ nu e inițializat st.

```
int *st;
st = malloc(sizeof(int));
wait(st);
```
} ok, dar dacă apelăm malloc, se face free st
⇒ malloc - funcție costisitoare.

```
int st;
wait(&st);
```
} ok.

```
wait(NULL);      // ok, când nu ne interesează codul de retur
```

```
int st;
wait(&st);
WIFEXITED(st); // dacă programul s-a terminat cu exit
WEXITSTATUS(st); // - codul de retur
if (WIFEXITED(st))
        ne = WEXITSTATUS(st);
WIFSIGNALED(st) // macrou care testează dacă programul
                   s-a terminat în urma unui semnal
WTERMSIG(st)    // cu ce semnal s-a terminat programul
```

wait : dezavantaje :

- apel blocant, sistemul rămâne în sleeping
- dacă se creează mai multe procese copil, iar nouă ne tb. unul anume.

```
int waitpid(int pid, int *st, int opt)
```
↓
pid - ul așteptat

pid > 0 ⇒ aștept după un pid anume

pid = -1 ⇒ aștept după orice proces copil ; exact ca la wait

st : - returnează pid
- se interpretează aproape la fel ca la wait

opt 0 = nicio opțiune

opt WNOHANG - apel neblocant
- nu stă să aștepte, ne scoate automat afară cu cod de retur -1, iar errno ia valoarea EAGAIN

opt WUNTRACED → cod de retur > 0 ⇒ pt. procesele stopate

Starea stopped : indiferent de comandă se oprește / împiedică comanda

WIFSTOPPED(st) - dacă a fost stopat

WSTOPSIG(st) - în urma cărui semnal a fost stopat.