14.10.13
curs

Funcțiile unui S.O.

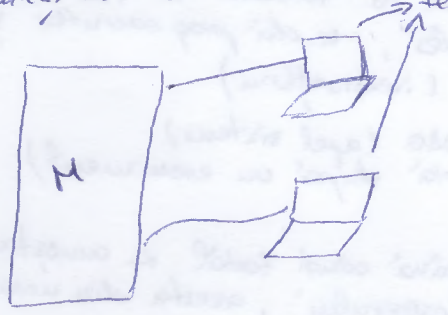
obligatorii pt un S.O.
1. Interfața cu utilizatorul
2. Gestiunea perifericelor
3. Gestiunea fișierelor
4. Apelurile sistem

5. Gestiunea proceselor (multitasking)
6. Gestiunea și protecția memoriei

- cu MSDos:
```
int *x;
while(1){
    *x = 33;
    x++;
}
```
// umple memoria cu 33
// scrie peste toate datele; executabile, S.O.
// S.O evoluate nu permit acest lucru.

7. Gestiunea utilizatorilor (multiuser)

- O mașină cu mai multe terminale



8. Lucrul în rețea

- Interfața cu utilizatorul
  - rezolvată în 2 moduri:
    - linie de comandă (shell) → windows cur.
    - mod grafic (fereastră)

(sub examen) ⊛ Terminale — alfanumerice
— grafic

- Gestiunea proceselor

→ prin întreruperi
- salvez context proces întrerupt,
- aleg proces pentru a fi continuat.

→ funcționează pe echitate
- restaurarea ~~procesului~~ contextului ~~ales~~ procesului ales
- programare ceas pentru întrerupere după slot.
- salt la programul ales

• Caracteristicile proceselor (generale și cele mai importante)
- identificator unic (PID)
- UID (Unix-Only) PID proces părinte (PPID) - fiecare proces este lansat de un alt proces (are un părinte)
- context (al procesului)
- lista fișierelor deschise
- prioritate (procesului)
- directorul curent
- pot adresa fișiere absolut prin cale: C:\x\y\z\f - în Windows
/x/y/z/f - în Unix
- ./x/y/z/f > prin adresare relativă .. - director părinte
- determină succesul sau eșecul lor adresarea relativă
- Stare proces
→ READY (procesul este în tabela de procese active și așteaptă)
→ RUNNING (a fost ales, i se dă prog counter și rulează)
- USER MODE (instrucțiuni)
- KERNEL MODE (apel sistem)
→ SLEEPING (așteaptă după un eveniment)
→ STOPPED (UO)
→ ZOMBIE (UO) (până când tatăl ia cunoștință de terminarea procesului, acesta din urmă este zombie)
- variabile de mediu și valorile lor

° Apelurile de sistem UNIX pt gestionarea proceselor

• Principii de funcționare a apelurilor de sistem

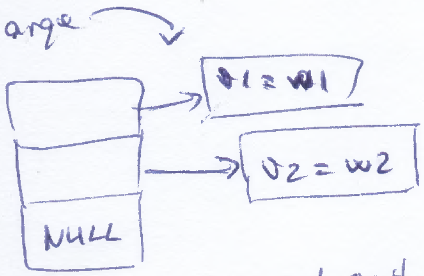
Exemplu
P testare cod de retur apel de sistem.

→ Se clasifică după tipurile de retur
° int ← eșec - returnează -1
succes - returnează ≥0
° void
° pointer ← eșec - returnează null
succes - returnează un pointer.

→ int errno; // poziționată pe 0 (apel sist a reușit)
pe alte val (apel sist. nu a reușit)
if (errno == ENOENT) // errno se modifică după fiecare apel sist.
errno ↳ errno.h
void perror(char *s) // afișează un mesaj specific lui errno urmat de un mesaj dat de noi

început procese → int main (int argc, char ** argv, char ** argc) → environment

argc ↳ câte argumente; argv ↳ valori (şir de şiruri de caract.)



ultimul pointer are val NULL şi la argc şi la argv

terminarea proceselor ↗ void exit (int x) // sau când a ajuns la ultima „}" a lui main

- apelul care permite unui proces să apeleze un alt proces fiu : int fork()

pid_t – tip de date | int, char, short, long | compatibil

```
a = b+2;
fork(); // intră în kernel mode şi când iese moşteneşte aproape tot de la procesul tată.
b = c+3;
```

↳ cod de retur:
- -1 – eşec
- >0 – cod de retur primit de tată
- =0 – cod de retur primit de fiu

```
if ((pid = fork()) == 0)
  {
    /* fiu */
  }
else
  {
    /* tată */
  }
```