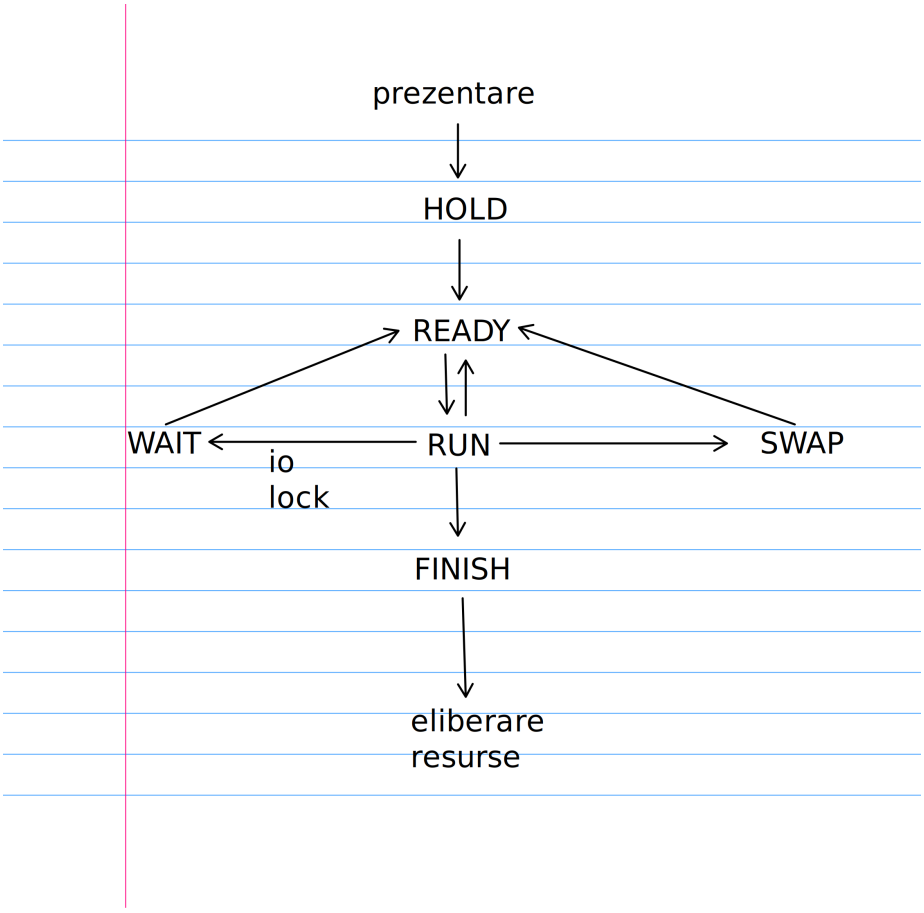
Procese

program = caracter static proces = program în execuție, caracter dinamic

Fazele unui proces



Deadlock

```
// p1      // p2
lock(X) | lock(Y)
lock(Y) | lock(X)
```

- iesire din deadlock
- detectare de deadlock
- prevenire

Iesire

- alege un proces (thread) victima si opreste-l
- daca am avea posibilitatea de a salva un savepoint, am putea cere unui proces sa revina la starea anterioara (fara sa-l oprim)
  - risc de livelock

Prevenire

- ce il face posibil
  1. Mutual exclusion
  2. Hold (lock) and wait
  3. Non-preemption
  4. Asteptarea circulara (dezactivata prin blocarea resurselor in aceeasi ordine)

Cum eviti deadlock???

Prin blocarea resurselor in aceeasi ordine.

Planificarea proceselor (scheduling)

- FCFS (First Come First Served)
- SJF (Shortest Job First)
  - clientul trebuie sa dea o estimare a duratei programului
  - risc de starvation pentru taskurile mari
- prioritati
- deadline scheduling
  - mai multe task-uri, fiecare cu o durata si un termen
- round robin
  - se aloca fiecarui proces cate o cuanta de timp

Gestiunea memoriei

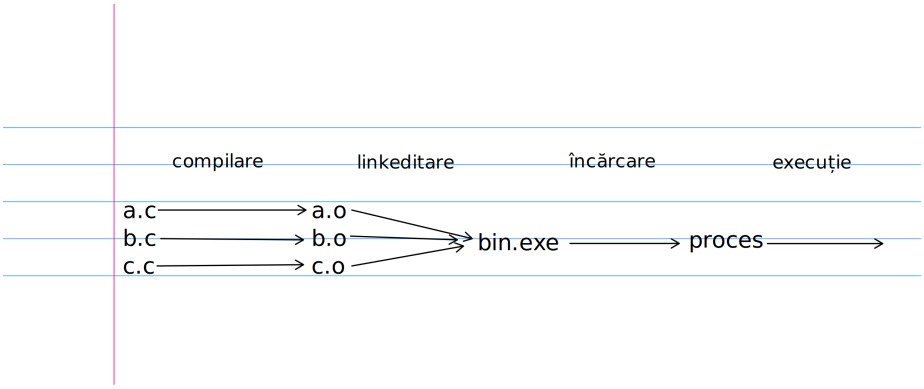
- alocare
- inlocuire
- plasare

Alocare

- alocare reală
  - sisteme single-tasking (1)
  - sisteme multi-tasking
    - partiții fixe
      - absolute (2)
      - relocabile (3)

        * partiții variabile (4)
- alocare virtuală
  - paginată (5)
  - segmentată (6)
  - paginat-segementată (7)

Calcul de adresă



(3) -> alocare reală -> sisteme multi-tasking -> partiții fixe -> relocabile

La compilare nu știm partiția în care va rula programul. => la executare trebuie făcut un calcul de adresă

AE (adresa din executabil) => offset in executabil AF (adresa fizică) => început partiție + AE

(4) -> alocare reala -> sisteme multi-tasking -> partitii variabile

- creează fragmentare

(5) -> alocare virtuala -> paginată

- memoria RAM se împarte în bucăți (pagini)
- programul se împarte în pagini virtuale
- fragmentarea e rezolvată, dar calculul de adresă e mai complex
- memoria se folosește mai eficient, paginile de cod ale unui program pot fi încărcate odată, și folosite de mai multe procese
- calculul de adresă
  - tabelă de pagini
  - AE -> adresă virtuala
    * PV (pagină virtuala)
    * offset

(6) -> alocare virtuala -> segmentată

- nu adresează fragmentarea
- grupeaza codul si datele în segmente cu protecție la acces
- calculul de adresă
  - tabelă de segmente
  - AE -> adresa virtuală
    - segment
    - offset

(7) -> alocare virtuala -> paginat-segementată

- calculul de adresă
  - tabelă de pagini
  - tabelă de segmente
  - AE -> adresă virtuala
    - segment
    - PV
    - offset

Politici de incarcare

- ce și când încărcăm în RAM la pornirea unui proces?
- ce = pagini
- când =
  1. încărcăm toate paginile de la inceput
     - pornire lentă
     - memorie ocupată de pagini care nu vor fi folosite
     - odată încărcat, merge repede
  2. încărcăm fiecare pagină când devine necesar
     - rulare mai lentă
     - pornire rapidă
     - paginile nefolosite nu ajung în memorie
  3. **principiul vecinătății** (dacă un proces referă o pagină, e probabil să refere curând paginile învecinate) => când incărcăm o pagină referită, aducem și câteva pagini de lângă ea

Politici de înlocuire

- cum alegem o pagină, pentru a fi mutată în swap, când memoria e plină?
  1. FIFO
  2. NRU (Not Recently Used)
     - fiecare pagina fizica are 2 biti, care periodic sunt resetati la 00
     - definim 4 clase de pagini:
       - 0 00 necitite si nescrise recent
       - 1 01 scrise recent
       - 2 10 citite recent

        * 3 11 citite si scrise recent
    - alegem o victima din cea mai mica clasa nevida
3. LRU (Least Recently Used)
    - considerand ca avem N pagini in RAM, intretinem o matrice de NxN biti astfel:
        * cand pagina k este accesata, populam cu 1 linia k, si apoi cu 0 coloana k
    - alegem ca victima pagina cu cea mai mica suma pe linie

Exemplu LRU

```
0000   0111   0110   0100   0000
0000 0 0000 3 0000 2 0000 1 1011
0000 > 0000 > 0000 > 1101 > 1001
0000   0000   1110   1100   1000
```

Politici de plasare

- cum tinem contabilitatea spatiilor alocate / libere in heap?
    - doua liste inlantuite
        * una cu toate blocurile de memorie alocata
        * una cu toate blocurile de memorie libera
- unde plasam o cerere de alocare?
    - FirstFit: plasam cererea de alocare in prima pozitie libera suficient de mare (rapida)
    - BestFit: alegem cel mai mic spatiu suficient de mare (mai lenta, fragmentare cu spatii foarte mici)
    - WorstFit: alegem cel mai mare spatiu suficient de mare
    - BuddyFit: alocam doar puteri ale lui doi, cat cea mai apropiata putere a lui 2
        * intretinem liste ale spatiilor goale, dupa dimensiuni
        * se bazeaza pe: $2^n = 1 + 1 + 2 + 2^2 + \ldots + 2^{n-2} + 2^{n-1}$

Cache-uri

- registri
- L1
- L2
- L3
- RAM
- HDD / SSD
- cum punem chestii in cache?
    - cache direct: pagina k merge in cache direct pe pozitia k%c

        * coliziuni, care conduc la thrashing
    - cache set: pagina k e pusa pe prima pozitie libera
    - cache set-asociativ: organizam cache-ul in grupuri de pagini, determinam grupul cu %, si apoi cautam in grup

Sisteme de fisiere

UNIX: - ext2 - ext3 - ext4 - xfs - hfs - reiserfs - zfs - btrfs

Windows: - FAT - NTFS

Structura discului (ext3)

- 0: boot
- 1: superblock (configurari)
- 2-n: i-node-uri
- n+1-. . . : date

i-node: punct de intrare in fisier

Structura unui fisier:

- idk
- adrese:
    - 1
    - 2
    - 3
    - . . .
    - 9
    - 10
    - 11 (spre alt block) (indirectare simpla)
    - 12 (spre alte block-uri) (indirectare dubla)
    - 13 (indirectare tripla)
- cat de mare poate fi un fisier?
    - depinde de:
        * dimensiune adresa A
        * cate adrese incap intr-un block N
    - $(10 + N + N^2 + N^3) \cdot N \cdot A$