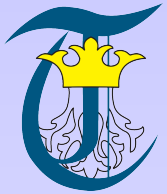


Universitatea *Transilvania* Brașov
Facultatea de Inginerie Electrică și Știința Calculatoarelor
Departamentul de Electronică și Calculatoare

Sisteme de operare

– curs 7 –

2012.04.10

ș.l. dr. ing. Kertész Csaba-Zoltán

Capitolul 3.
Managementul memoriei

- Partea sistemului de operare care se ocupă cu modul de utilizare a memoriei este managerul de memorie
- managerul va trebui să țină evidența zonelor de memorie utilizată și neutilizată, să aloce / dealoce memorie proceselor când este necesar, să ofere protecția diferitelor zone de memorie și să realizeze swappingul între memoria principală și disk
- există o serie de scheme de management al memoriei

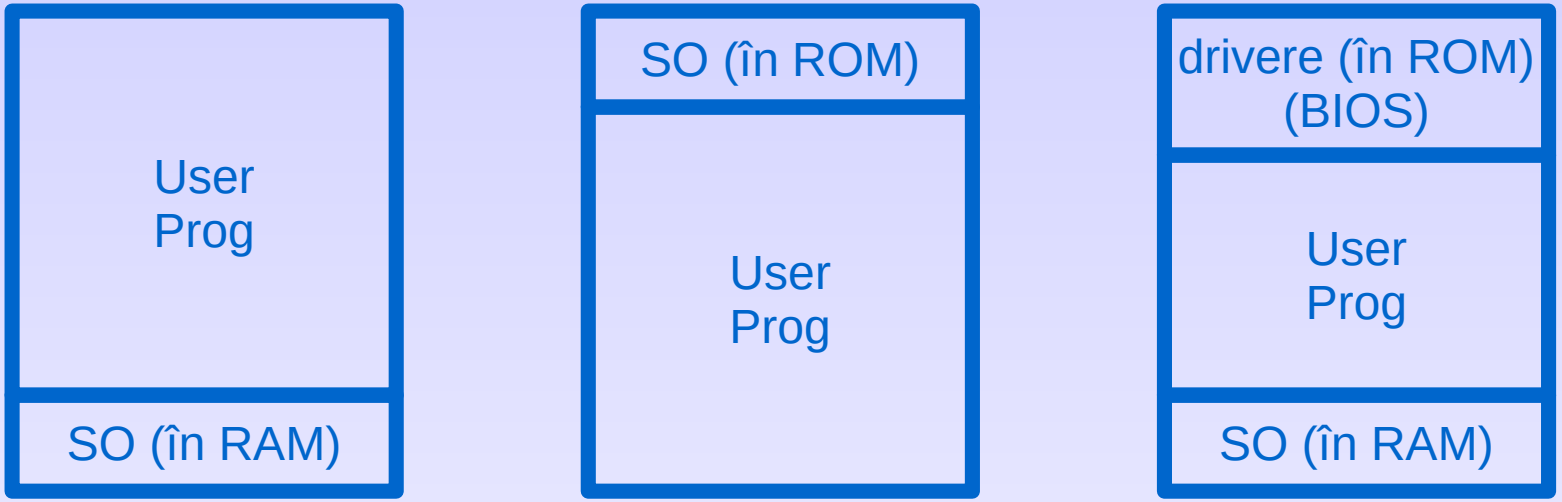
3.1. Managementul memoriei fără utilizarea paginării și swappingului

- există manager de memorie:
  - care transferă procesele din memorie pe harddisk și invers în timpul execuției (paging, swapping)
  - care nu transferă procese
- paginarea și swappingul se utilizează datorită mărimii insuficiente a memoriei principale

3.1.1. Monoprogramare fără paginare și swapping

- cea mai simplă schemă de MM este de a avea un singur proces în memorie la un moment dat, iar procesul respectiv poate să utilizeze toată memoria disponibilă
  - această metodă s-a folosit până pe la '60
- procesul trebuie să conțină câte un driver pentru fiecare dispozitiv I/O utilizat

Scheme de monoprogramare



- un singur proces poate fi rulat la un moment dat
- utilizatorul introduce o comandă la un terminal, SO va încărca programul în memorie și va executa
- la terminarea programului SO revine și așteaptă următoarea comandă

3.1.2. Multiprogramarea și utilizarea memoriei

- avantajele multiprogramării:
  - o aplicație va fi mai ușor de programat dacă este împărțită în mai multe procese
  - se asigură servirea mai multor utilizatori simultan (pot exista mai multe procese în memorie la un moment dat)
  - multe procese își petrec o mare parte din timp așteptând terminarea unor operații I/O
    - de exemplu procesul citește date pe care trebuie să opereze de pe disk

Modelarea multiprogramării

- se poate îmbunătăți utilizarea CPU prin multiprogramare
  - ex. ideal: 5 procese în memorie, fiecare cu 20% calcule și 80% IO => CPU va fi utilizat tot timpul (presupunând că procesele nu așteaptă la IO toate deodată)
- vom obține un model mai bun dacă privim utilizarea CPU d.p.d.v. probabilist
  - un proces petrece o fracțiune p din timpul total pentru a aștepta terminarea unei operații I/O

Modelul probabilist

- având n procese în memorie la un moment dat, probabilitatea ca toate cele n procese să aștepte terminare unei operații I/O este p^n
- utilizarea CPU: $1 - p^n$

  n = grad de multiprogramare
- de exemplu: p = 0.8
  - 1 proces: CPU utilizat = 1 – 0.8 = 0.2
  - 2 procese: CPU utilizat = 1 – 0.64 = 0.36
  - ...
  - 10 procese: CPU utilizat ≈ 0.94

Modelul probabilist

- modelul prezentat este doar o aproximare pentru că s-a presupus că procesele sunt independente
- în realitate procesele nu sunt independente (ele vor trebui să aștepte unul pe celălalt)
- un model mai precis se obține utilizând teoria cozilor
- totuși cu modelul probabilist putem realiza o predicție bună a performanțelor

Analiza performanțelor pe baza modelului probabilist

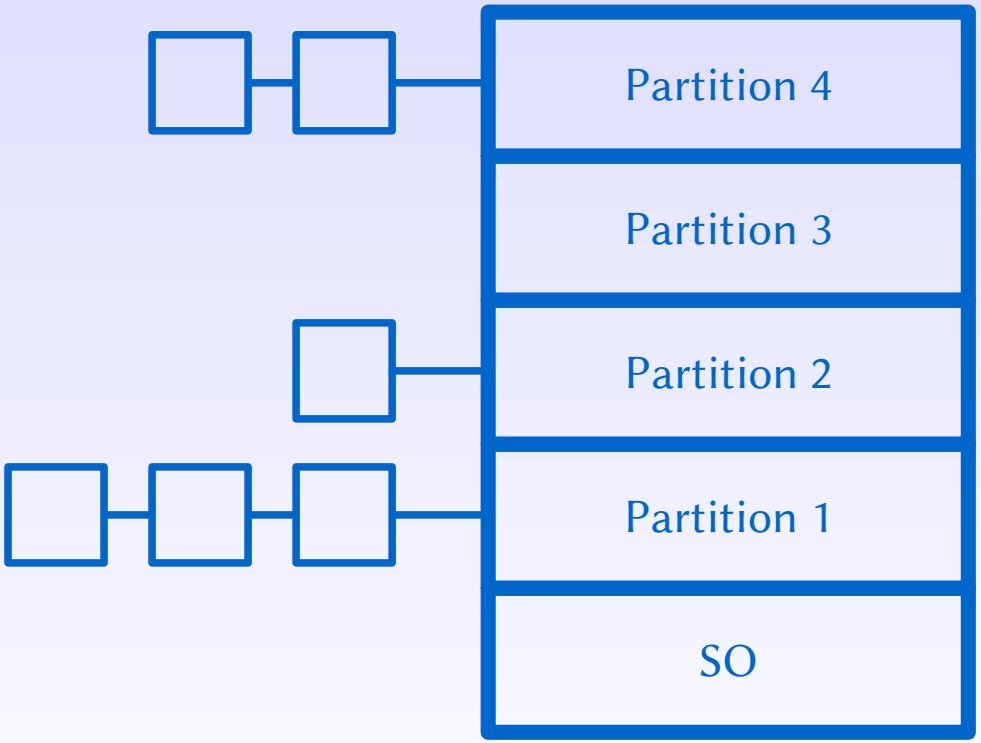
- dacă avem 1MB memorie:
  - 200 kB SO
  - 4 x 200 KB procese cu timpul mediu în IO = 80%
  - CPU utilizat = $1 - 0.8^4 \approx 1 - 0.4 \approx 0.6$
- adăugând încă 1MB:
  - CPU utilizat = $1 - 0.8^9 \approx 1 - 0.12 \approx 0.88$
  - creștere cu 45% (investiție bună)
- adăugând încă 1MB:
  - CPU utilizat = $1 - 0.8^{14} \approx 1 - 0.04 \approx 0.96$
  - creștere cu 10% (investiție mai puțin bună)

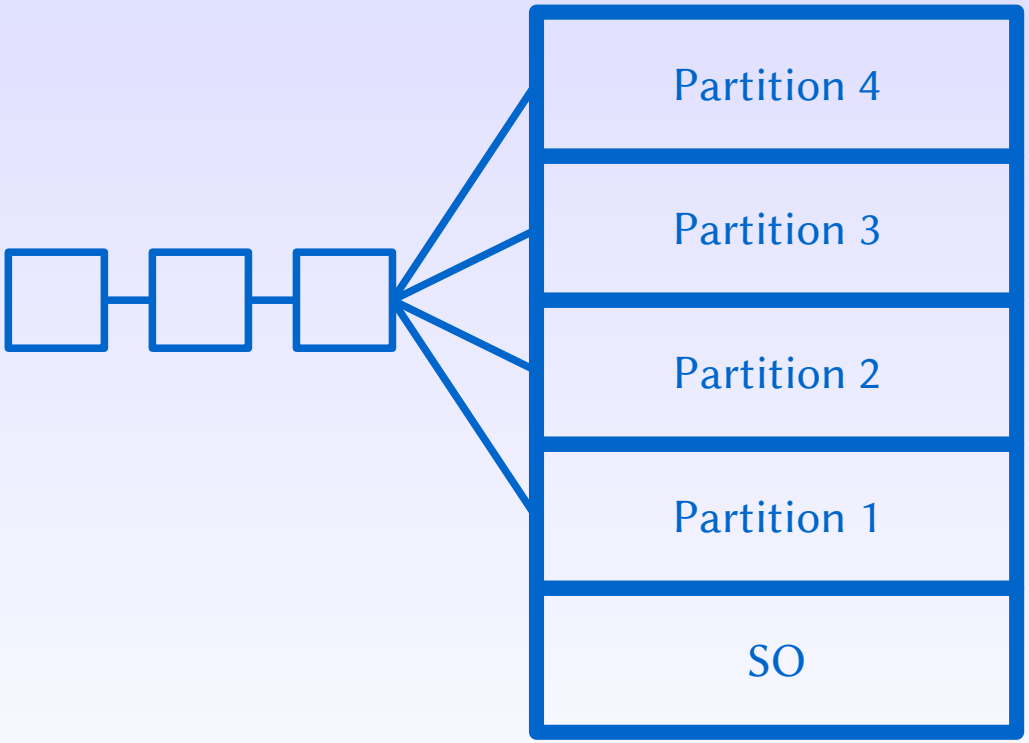
3.1.3. Multiprogramarea utilizând partiții fixate

- cum putem organiza memoria pentru a putea avea mai multe procese în memorie la un moment dat?
- o modalitate simplă este de a împărți memoria în n partiții (posibil inegale)
  - acest lucru se poate realiza de exemplu de către operator la bootarea sistemului
- deoarece partițiile sunt fixe, spațiul neutilizat dintr-o partiție nu va putea fi utilizat de către alte joburi

Alocarea proceselor în partiții

- multiple input queues
- single input queues





Relocatarea și protecția

- multiprogramarea conduce la apariția a 2 probleme ce trebuie rezolvate
  - relocatarea
  - protecția
- joburi diferite vor rula la adrese diferite
- link-editorul va trebui să cunoască la ce adresă de memorie va începe programul
  - dacă prima instrucțiune în program este un salt la adresa 0x100:
    - în partiția 1 va trebui să sară la 100k + 0x100
    - în partiția 2 va trebui să sară la 200k + 0x100

Relocatarea și protecția

- pentru relocatare trebuie modificate instrucțiunile programului la încărcarea acestuia
  - programele încărcate în partiția 1 vor avea 100k adăugat la fiecare adresă
  - programele încărcate în partiția 2 vor avea 200k adăugat la fiecare adresă
- relocatarea în timpul încărcării nu rezolvă problema protecției
  - programele pot întotdeauna construi orice adresă din memorie deci pot citi sau scrie orice cuvânt din memorie

Relocatarea și protecția

- nu este de dorit ca un proces să poată accesa memoria unui alt proces
- soluția: folosirea a două registre
  - bază
  - limită
- la planificarea unui proces se încarcă registrul bază cu adresa de început a partiției, iar registrul limită cu sfârșitul partiției

Relocatarea și protecția

- fiecare adresă folosită de către program va fi relativă la adresa de început a partiției (registrul bază)
- adresele vor fi verificate să nu depășească limita partiției
- numai SO poate modifica registrele bază și limită
- avantajul acestei abordări că putem muta programul oriunde în memorie

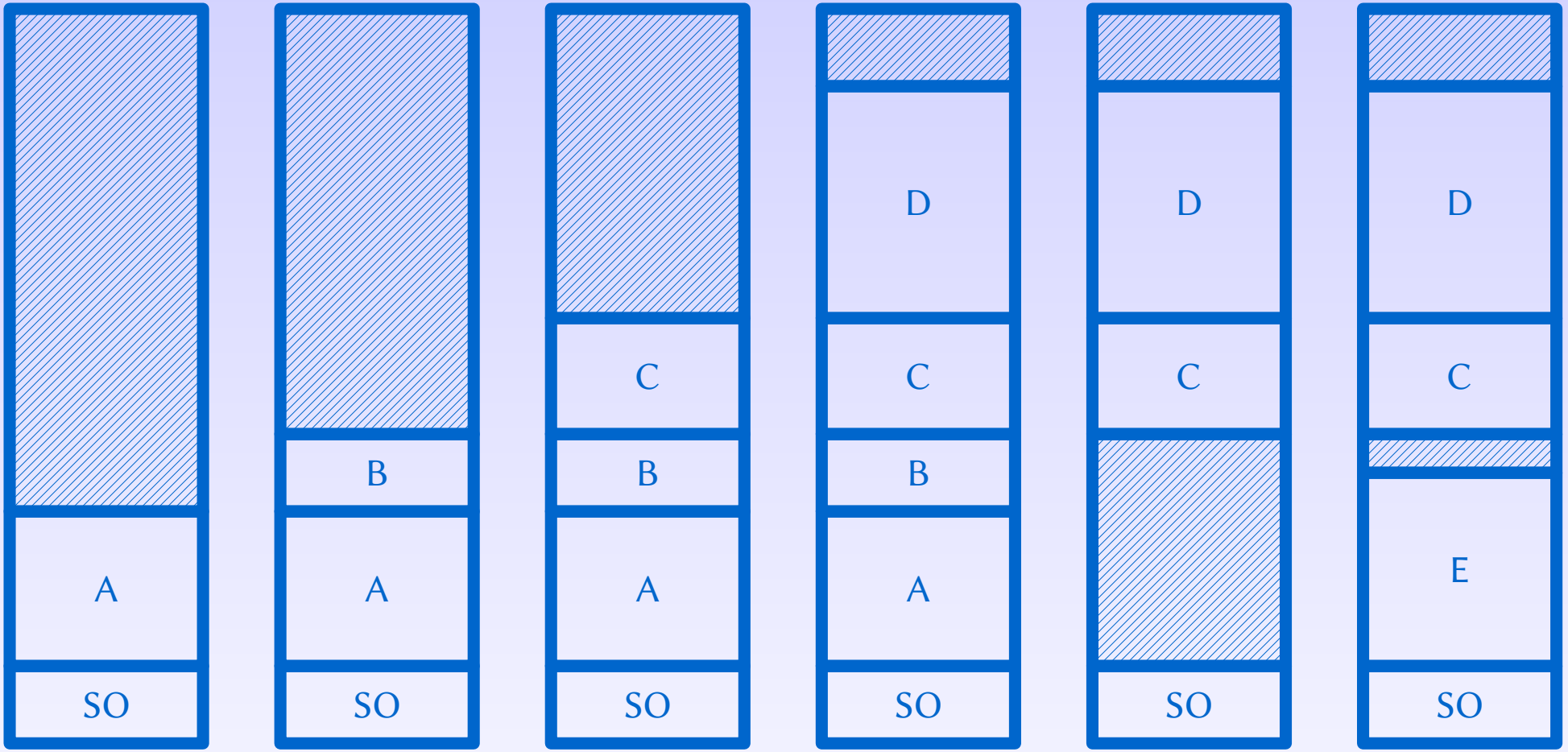
3.2. Swapping

- în cazul sistemelor time-sharing există mai multe procese care trebuie să se execute
- deseori memoria nu este suficientă pentru a memora toate aceste procese
- unele procese vor trebui mutate pe disc, iar în momentul rulării acestora trebui aduse înapoi în memorie:
- transferarea proceselor între memoria și disc se numește swapping

3.2.1. Multiprogramarea utilizând partiții variabile

- în principiu swappingul poate fi folosit și cu partiții fixe, dar această abordare nu este eficientă (rămânând zone neutilizate de memorie)
- astfel în cazul în care se folosește swapping se abordează o partiționare variabilă a memoriei:
  - numărul, poziția și mărimea partițiilor variază dinamic în funcție de procesele care vor fi plasate sau eliberate din memorie
  - se îmbunătățește folosirea memoriei

Partiții variabile



- se complică foarte mult procesul de alocare și dealocare a memoriei

Probleme la alocare și dealocare

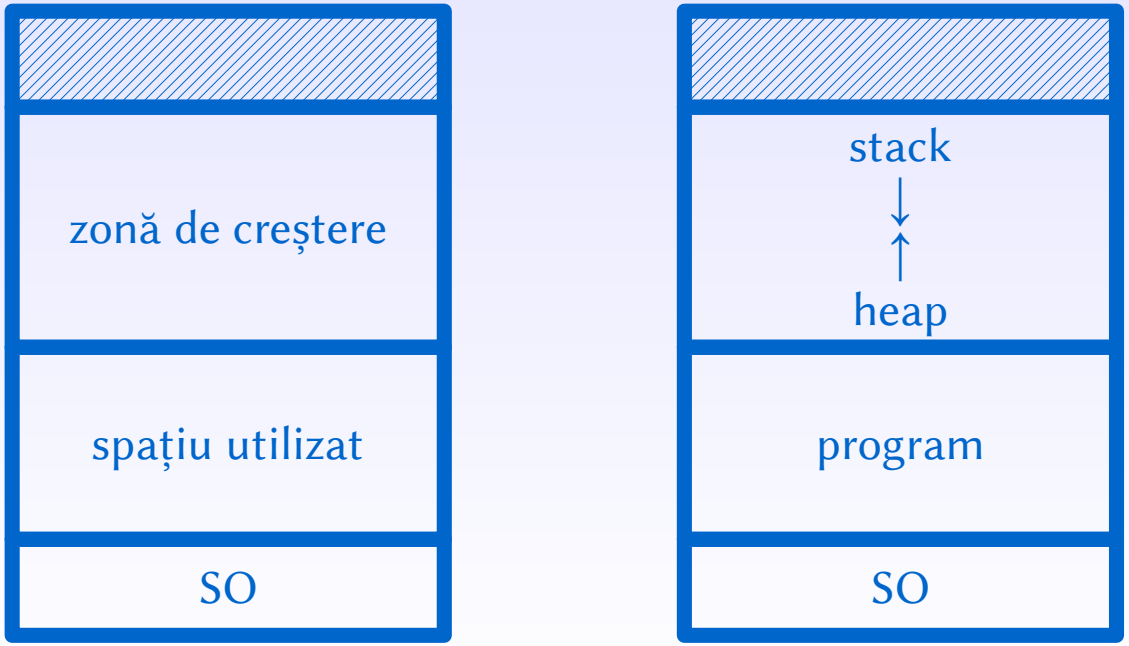
- scopul este ca procesele să fie aranjate astfel încât să folosească cât mai bine memoria
- o metodă ar fi mutarea proceselor astfel încât să nu existe găuri:
  - compactarea memoriei
  - nu prea se folosește pentru că consumă mult timp
- o problemă care mai apare este: câtă memorie va trebui alocat pentru un proces
  - cele mai multe ori segmentele de date alocate procesului cresc în timp (heap, stack)

Problema creșterii memoriei necesare pentru un proces

- dacă zona de memorie adiacentă procesului este liber, se poate extinde în această zonă
- altfel va trebui mutat procesul într-o altă zonă de memorie mai mare care permite creșterea dimensiunii
- dacă nu e loc suficient în memoria, va trebui trimise unele procese din memorie pe disc
- dacă nici pe disc nu mai e loc atunci procesul trebuie suspendat sau omorât

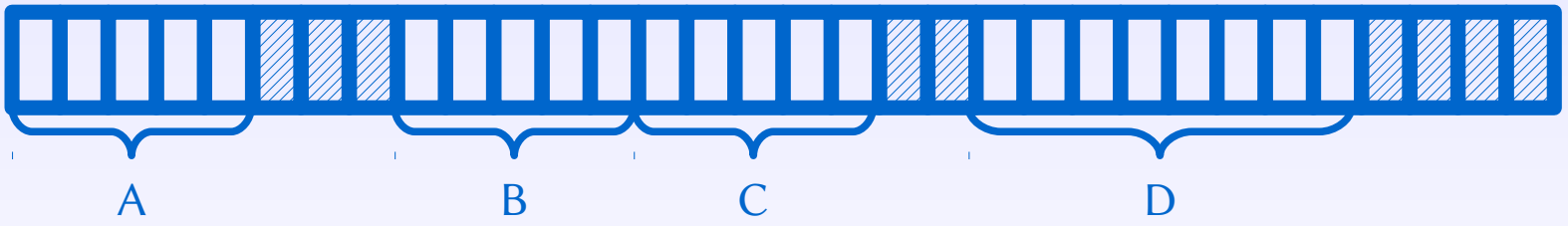
Alocare/dealocare

- pentru a reduce overhead-ul datorat swappingului se poate aloca o zonă mică de memorie goală în care procesele adiacente pot să crească
  - diminuează eficiența folosirii memoriei



3.2.2. Managementul memoriei utilizând bit maps

- memoria este divizată în unități de alocare (zeci de B – zeci de kB)
- fiecărei unități de alocare îi va corespunde un bit în bit map
  - 0: dacă unitatea nu a fost alocată
  - 1: dacă a fost alocată



11 111 000
11 111 111
11 001 111
11 110 000

Bit maps

- o problemă importantă este alegerea dimensiunii unității de alocare
  - unități mici: bit map necesar mare (de ex. pentru o unitate de 4B vom avea nevoie de un bit pentru fiecare 32 de biți => 1/33 din memorie ocupat pentru bit map)
  - unități mari: bit map mic, dar poate conduce la utilizarea ineficientă a memoriei
- o altă problemă este faptul că la alocare unui proces, managerul de memorie va trebui să caute un număr de biți de 0 consecutivi suficient pentru procesul respectiv