Gestiunea & securitatea memoriei

I. Securitatea memoriei

Funcționalitatea ei este indispensabilă într-un sistem de operare multitasking;

Un proces este compus dintr-un executabil și din date. Cele două componente sunt plasate în zone diferite de memorie: prima este RO (ReadOnly), a doua RW.

Amintim de semnalul SIGSEF primit în urma încercărilor unui proces de a scrie într-o zonă în care nu are acces.

Zona de date a unui proces este compusă din trei subzone:

A. Zona de date statice
B. Zona de heap
C. Zona de stiva (stack)

A. Zona de date statice

Executabilul plasează variabilele globale ș pe cele locale statice în această zonă, la o adresă dată de compilator. (Variabilele sunt inițializate automat cu 0)

B. Zona de heap

Aici sunt plasate variabilele alocate dinamic cu funcțiile **malloc,calloc** și **realloc.**

(vezi mai jos *Alocare dinamică în C*)

C. Zona de stivă

În momentul apariției unei funcții, *contextul dinamic* al acesteia este pus pe stivă.
Contextul dinamic constă în:
- adresa de retur;
- cod de retur;
- parametrii;
- spațiu alocat pe stivă pentru variabilele locale automatice.

II. Gestiunea memoriei

Pentru ca un proces să intre în stare **running,** întreg contextul său trebuie să fie încărcat în memorie.

Problema insuficienței spațiului: → apare fenomenul de sort

memorie → pagini → segmente

Contextul oricărui proces trebuie încărcat în totalitate în memorie și ocupă un număr întreg de pagini.

ex.: P1 → memorie
P2 → memorie
...
Pn → memorie
Se ajunge astfel ca la un moment x toată memoria să fie încărcată.
Pn+1 vrea k pagini contigue (i.e. Așezate una după alta)

Atunci, sistemul de operare sacrifică unul sau mai multe procese contigue, care însumate ocupă cel puțin k pagini. Se salvează conținutul zonelor de memorie ale acestora pe disc și se alocă k pagini noului proces. Apoi, procesele salvate se vor reîncărca de pe disc în memorie ș.a.m.d.

Acesta este fenomenul de sort, sau memoria virtuală.

Scopurile urmărite în momentul alegerii proceselor ce vor fi sacrificate

1. Minimizarea numărului de accese la disc. (Deoarece discul este un periferic, accesele la el duc la o scădere a performanței).
2. Întârzierea pe cât posibil a fragmentării memoriei. (Dacă nu se găsește cel puțin un proces cu zonă continuă de k pagini, se caută pentru cel puțin k pagini. Se obține astfel o zonă mică rămasă goală. În timp apare o multitudine de astfel de zone mici libere e nu pot fi folosite ca atare, deci apare o fragmentare a memoriei. Sistemul de operare face din când în când o defragmentare a memoriei și se obține o zonă continuă liberă.)

Strategii posibile

1. FIFO (nu se aplică)
2. Best fit (Cea mai bună potrivire)
– vreau să eliberez cel puțin k pagini, iau toate combinațiile posibile și caut cel mai apropiat număr mai mare decât k de pagini. Astfel obținem cea mai bună potrivire.
– Dezavantaj: conduce rapid la fragmentare.
3. LRU (Last Recent User)
   - se iau în considerare și prioritățile proceselor și experiența cu acele procese și sacrifică procesele, cărora li se dau controlul mai rar (cele mai vechi pagini utilizate).

Alocare dinamică în C

Apelurile sunt definite în biblioteca standard C (stdlib.h). (Observăm că ele nu sunt apeluri sistem, dar în interiorul lor se pot face apeluri sistem.)

1. **void* malloc (int dim)**

   - **dim** este dat cu ajutorul operatorului **sizeof()** și reprezintă dimensiunea în bytes a zonei de memorie pe care dorim să o rezervăm
   - se va realiza cea mai bună aliniere posibilă în memorie
   - returneaza un pointer către zonă alocată sau NULL în caz de eroare (aceasta se produce dacă nu există suficient spațiu liber)

2. **void* calloc(int size, int dim)**

   - alocare contiguă a **dim** obiecte de dimensiunea **size**
   - **size** este dat cu ajutorul operatorului **sizeof();**
   - returnează un pointer către zona alocată sau NULL în caz de eroare (aceasta se produce dacă nu există suficient spațiu liber)
   - se observă că apelul *calloc(x,y)* este echivalent cu apelul *malloc(x·y)*. Diferența constă în faptul că funcția *calloc* inițializează memoria cu 0;

3. **void* realloc(void* p, int dim)**

   - realizează o alocare dinamică;
   - **dim** reprezintă dimensiunea anterioară + un increment (ex.: 1000+2)
   - pointerul **p** pasat trebuie să fie NULL sau un pointer obținut prin *malloc* sau *calloc*
   - modul de funcționare: se pune întâi problema spațiului liber, dacă acesta este găsit, atunci se alocă memoria și se returnează **p**; dacă, însă, *increment x octeți* sunt ocupați se caută o altă zonă de memorie de dimensiunea *dim_anterioară+increment*; dacă o găsește, o alocă, copiază conținutul și eliberează zona de memorie indicată de **p**, și returnează un nou **p**.

4. **void free(void*p)**

   - eliberează zona de memorie
   - pointerul **p** trebuie să fie NULL (nu are niciun efect) sau un pointer obținut prin *malloc*, *calloc* sau *realloc*.