UNIVERSITATEA POLITEHNICA BUCURESTI

Facultatea de Electronica, Telecomunicatii si Tehnologia Informatiei

GESTIUNEA MEMORIEI

Coordonator :Dr. Ing. Ștefan Stancescu

Student1 :Simion Bogdan Mihai

Student2:Chelu Marius-Florin

Gestiunea Memoriei

1.Introducere

Memoria reprezinta o resursa fizica fundamentala pentru orice sistem de calcul. Ea se caracterizeaza prin capacitate, mod si timp de acces la informatie, durata ciclului, viteza de transfer etc. Putem vorbi despre o impartire a echipamentelor, ce pot inregistra, stoca si reda informatia, in doua categorii: memoria interna(operativa, principala, centrala) si memoria externa. Memoria principala este cel mai adesea insuficienta pentru programele ce se afla in executie. Memoria interna este o componenta principala a sistemelor de calcul, ea avand rolul de a pastra date si programe, cand acestea urmeaza sa fie folosite in urma executarii unui proces dde catre CPU. Memoria primara alaturi de registrii CPU si memoria cache alcatuiesc memoria executabila(componentele memoriei implicate in executia unui proces). Memoria externa(secundara) este folosita pentru stocarea datelor pe o perioada lunga de timp.

Timpul de acces la memoria interna trebuie sa fie cat mai mic posibil(realizabil prin proiectarea corespunzatoare atat partii hardware cat si al software-ului folosit pentru gestiunea memoriei). Pentru aceasta este folosita memoria cache, un timp de memoria cu timp scazut de acces ce contine informatiile cele mai recent utilizate de CPU. Pe de alta parte capacitatea memoriei adresabile trebuie sa fie cat mai mare posibil; realizabil cu ajutorul conceptului de memorie virtuala.

Subsistemul de gestiune al memoriei din cadrul unui sistem de operare este folosit de toate celelalte subsisteme. Memoria este o resursa importanta, de aceea algoritmii de utilizare si gestiune a acesteia trebuie sa fie cat mai eficienti.

Rolul subsistemului de gestiune a memoriei este de :

- a contabiliza zonele de memorie fizica (ocupate sau libere);
- a aloca memorie proceselor sau celorlalte subsisteme;
- a dealoca memoria de care procesele nu au/nu mai au nevoie;
- a mapa paginile de memorie virtuala ale unui proces (pages) peste paginile fizice (frames)
- a calcula translatarea adresei(realocare)
- a gestiona memoria secundara
- a proteja memoria.

Sistemul de operare ofera un set de interfete (apeluri de sistem) care permit alocarea/dealocarea de memorie, maparea unor regiuni de memorie virtuala peste fisiere, partajarea zonelor de memorie.

Necunoasterea acestor interfete sau folosirea lor intr-un mod neadecvat poate duce la o multitudine de probleme cum ar fi: memory leak-uri, accese nevalide, suprascrieri, buffer overflow, corupere de zone de memorie.

Prin urmare este esentiala cunoasterea contextului in care lucreaza subsistemul de gestiune a memoriei si intelegerea interfetei care ne este pusa la dispozitie de catre sistemul de operare .

2.Structura memoriei

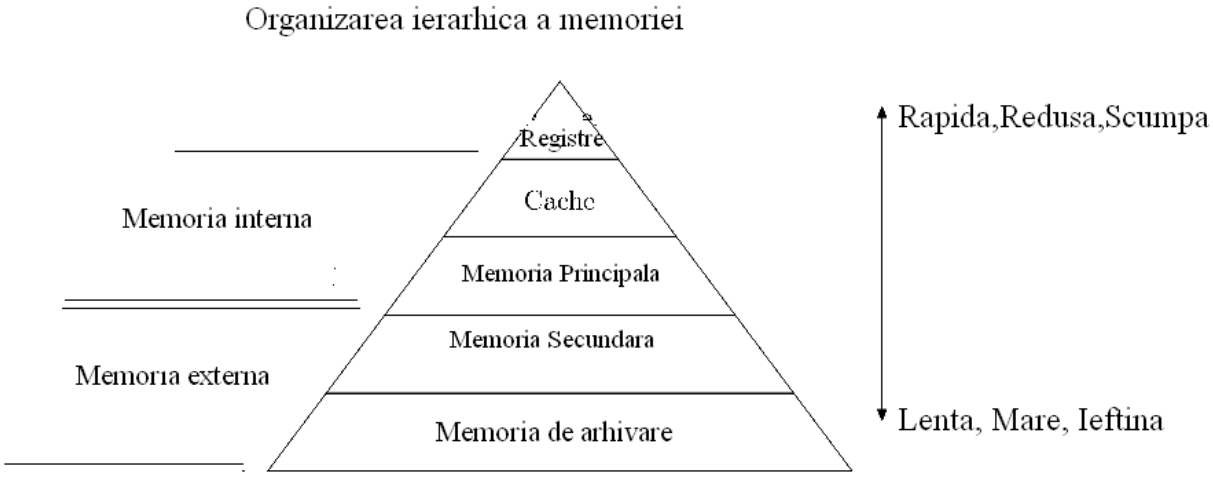


Figura 1 :organizarea ierarhica a memoriei

Memoria cache contine informatiile cel mai recent utilizate de CPU. Este de dimensiune redusa dar are o viteza foarte mare. La fiecare accesare, CPU verifica intai memoria cache pentru informatiile dorite apoi celelalte memorii.

Memoria principala contine instructiunile si datele pentru toate procesele sistemului. Aceasta are o viteza de acces destul de mare.

Memoria secundara este intalnita la sistemele ce contin si mecanisme de memorie virtuala. Aceasta este considerata o extensie a memoriei principale la care accesul se face mult mai lent.

Memoria de arhivare este memoria gestionata de utilizator si contine fisiere, baze de date etc.

Memoria cache si memoria principala formeaza memoria interna si sunt accesate de catre CPU in mod direct. Datele din memoriile secundare si de arhivare trebuie mai intai mutate in cea interna inainte de a fi accesate de CPU.

3.Memoria virtuala

Memoria virtuala este o tehnica ce permite executia proceselor fara a impune necesitatea ca intreg fisierul executabil sa fie incarcat in memorie sau capacitatea de a adresa un spatiu de memorie mai mare decat este cel din memoria interna a unui sistem de calcul. Utilizarea memoriei virtuale ofera avantajul ca poate fi executat un program de dimensiune oricat de mare (mai mare decat dimensiunea memoriei fizice); spatiul de adrese al procesului este divizat in parti care pot fi incarcate in memoria interna atunci cand executia procesului necesita acest lucru si transferate inapoi in memoria secundara, cand nu mai este nevoie de ele. Spatiul de adrese al unui program se imparte in segmentul de cod, segmentul de date de date si segmentul de stiva.

Partea (segmentul) de cod are evident un numar de componente mai mare, contine codul executabil al programului. Este de tip read only.

Segmentul de date contine variabilele programului(poate contine si tablouri, siruri de caractere etc.). Este de tip read write. Are doua parti: date initializate(primesc valori la compilare) si neinitializate. Programele contin o faza necesara initializarii structurilor de date

utilizate in program, alta pentru citirea datelor de intrare, una (sau mai multe) pentru efectuarea unor calcule, altele pentru descoperirea erorilor si una pentru iesiri. Programul executabil contine datele initializate, textul (codul) si un header care ii spune SO-ului numarul de octeti ce trebuie alocata pentru datele neinitializate (astfel se evita pastrarea efectiva în programul executabil a spatiului necesar datelor neinitializate). Segmentul de date isi poate modifica in timpul rularii unui program atat continutul cat si dimensiunea.

Segmentul de stiva incepe de la adresa cea mai mare a spatiului de adrese virtuale a procesului si creste spre adresele inferioare. Programele nu isi gestioneaza explicit segmentul de stiva. La lansarea in executie a unui program, stiva nu este goala. Ea contine toate variabilele de mediu precum si linia de comanda prin care s-a lansat in executie.

“Cand o anumita parte a programului este executata, este utilizata o anumita portiune din spatiul sau de adrese, adica este realizata o localizare a referintelor. Cand se trece la o alta faza a calculului, corespunzatoare logicii programului, este referentiata o alta parte a spatiului de adrese si, deci se schimba aceasta localizare a referintelor. De exemplu, in figura 2, spatiul de adrese este divizat in 5 parti.

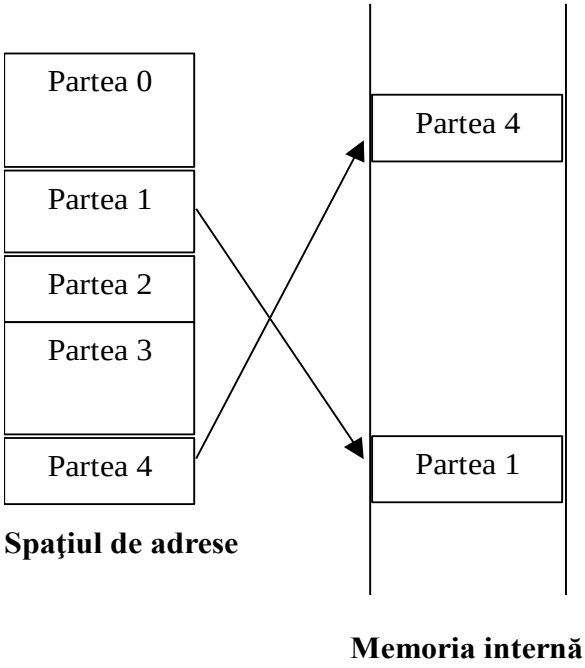


Figura 2. Memoria fizica si cea virtuala

Numai partile 1 si 4 din spatiul de adrese corespund unor faze ale programului care se executa la momentul respectiv, deci numai acestea vor fi incarcate in memoria interna. Parti diferite ale programului vor fi incarcate in memoria primara la momente diferite, in functie de localizarea in cadrul procesului. Sarcina administratorului memoriei este de a deduce localizarea programului si de a urmari incarcarea in memorie a partitiilor din spatiul de adrese corespunzatoare, precum si de a tine evidenta acestora in memoria interna, atata timp cat sunt utilizate de catre proces.

Administratorul memoriei virtuale aloca portiuni din memoria interna care au aceeasi dimensiune cu cele ale partitiilor din spatiul de adrese si, deci incarca imaginea executabila a

partii corespondente din spatiul de adrese intr-o zona din memoria primara. Acest lucru are ca efect utilizarea de catre proces a unei cantitati de memorie mult mai reduse. “ [1]

4.Segmentarea

Un segment este o zona de memorie utilizat pentru a retine date sau instructiuni. Adresa unui octet dintr-un segment este de forma: (adresa_baza_segment, deplasament_in _cadrul_segmentului).

Procesul de segmentare presupune impartirea programelor in mai multe segmente pentru care nu este necesara stocarea in zone apropiate de memorie, desi toate trebuie incarcate intr-un bloc compact de memorie. Acest lucru diminueaza efectele fragmentarii, facand posibila utilizarea si a unor zone de memorie de mai mici dimensiuni.

Segmentele corespund unor submultimi logice ale programului. Se poate considera, spre exemplificare, segmentul de cod si segmentul de date. Segmentarea poate fi mai pronuntata decit atat, putind exista mici segmente compuse din una sau mai multe rutine sau din una sau mai multe structuri de date.

Fiecarui segment i se asociaza o structura de date ce contine informatii ce descriu segmentul cum ar fi: adresa de inceput a segmentului; dimensiunea segmentului; drepturi de acces la segment. Pe baza acestor informatii se face conversia adreselor in procesul de segmentare.

In momentul in care un proces incearca sa acceseze o adresa de memorie au loc urmatoarele operatii:

- se verifica in descriptorul segmentului drepturile de acces, pentru a se decide daca procesul are dreptul de a accesa adresa dorita
- se verifica daca deplasamentul nu depaseste dimensiunea segmentului
- daca se produce o eroare la unul din pasii anteriori, se genereaza o intrerupere, a carei rutina de tratare va anunta procesul asupra erorii sau il va termina in mod fortat
- daca nu s-a produc nici-o eroare, se calculeaza adresa fizica si se realizeaza accesul propriu-zis;

Este esential sa existe un algoritm cu ajutorul caruia sa se decida care zona liberea este cea care va fi potrivita pentru a fi ocupata de un nou segment.

Exemple de algoritmi:

- first fit: se parcurge memoria in ordine crescatoare si segmentul se stocheaza in prima zona suficient de mare;
- next fit: la fel ca first fit doar ca se utilizeaza a doua zona libera;
- best fit: se parcurge memoria si se plaseaza in cea mai mica zona suficient de mare;
- worst fit: se plaseaza in cea mai mare zona libera;

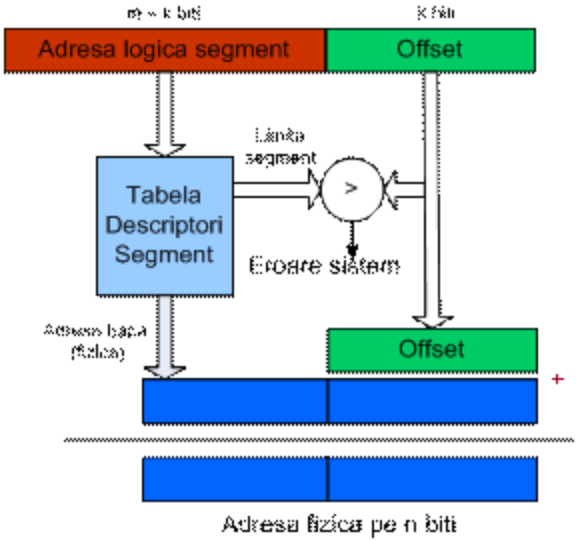


Fig. 3. Schema de conversie a adresarii ceruta de segmentare. Mecanismul de segmentare

"In figura 3. se prezinta algoritmul utilizat pentru conversia adreselor. Identificarea segmentului este facilitata de existenta unei submultimi de adrese virtuale. Descriptorul acestuia este extras din tabela de descriptori ai segmentelor. Dupa aceasta, conversia de adresa se face similar cazului tehnicii swapping.

Relocarea este critica din punctul de vedere al performantei sistemului, intrucit ea trebuie efectuata la fiecare ciclu memorie. Deci, daca tabela descriptorilor segmentului ar fi pastrata in memoria principala, apare necesitatea unui ciclu suplimentar de acces la memorie, ceea ce dubleaza timpul cerut de efectuarea unui ciclu memorie.

Depasirea acestei dificultati se face stocand tabela descriptorilor segmentelor intr-o, mica dar foarte rapida, memorie RAM cu care este prevazuta MMU. Dimensiunea limitata a acestei memorii speciale necesita folosirea tehnicii de swapping intre aceasta si memoria principala a sistemului.

Un alt tip de problema apare in cazul in care numarul de segmente depaseste cantitatea ce poate fi stocata in memoria RAM a MMU. In acest caz, memoria rapida din cadrul MMU se poate utiliza ca memorie cache, in care se pastreaza cei mai recent utilizati descriptori de segment. Cind se initializeaza procesul de conversie a adreselor, MMU cauta in memoria sa interna descriptorul de segment; daca el nu este gasit, descriptorul este citit din memoria primara si copiat in memoria interna. Principiul localitatii programelor indica o probabilitate ridicata ca descriptorul sa se afle in memoria interna a MMU, cel de al doilea ciclu de memorie este doar arareori necesar , deci nu influenteaza intr-un mod important performanta de ansamblu.

Un alt punct important, referitor la intirzierea de relocare, este viteza cu care se face adunarea indicata in partea de jos a fig. 3. Adresa fizica de m biti se obtine prin adunarea portiunii de k biti (k =< m), adresa baza, cu offsetul de h biti (h =< m). h determina dimensiunea maxima a segmentului, intrucit nu este posibil, intr-un segment dat, sa se adreseze o informatie cu un offset mai mare decit 2^h-1. Cei mai semnificativi m-k biti nu sint adunati, de fapt, ci doar copiati din adresa virtuala in adresa fizica, asa ca adunarea are loc doar pentru k biti (daca h+k > m). Deci, cu cit numarul k este mai mare,cu atit intirzierea provocata de adunare creste, ceea ce recomanda utilizarea de valori mici pentru k. Pe de alta parte , adresa baza permite fixarea unui numar de 2^{m-k} frontiere de segment, asa ca valori mici ale lui k vor augmenta procesul de fragmentare. Valorile tipice pentru situatii practice ale diferentei m-k sint, adesea, intre 4 si 8."[2]

5.Memorie virtuala segmentata

Cind s-a discutat despre localitatea programelor s-a aratat ca executia lor evolueaza printr-o serie de faze, iar spatiul de adresare la care se face acces in cadrul unei faze este mai mic decat intregul spatiu de adresare al programului. Daca segmentele la care se face acces in cadrul unei faze sint stocate in memoria primara, programul este executat aproape la fel de repede ca in cazul in care toate segmentele sale s-ar gasi in memoria primara, deoarece practic toate operatiile de acces au loc la segmente existente in aceasta. Cind se genereaza o referire la un segment absent, el este adus in memoria principala, inlocuind, eventual, alte segmente continute de memoria amintita.

Implementarea mecanismului presupune ca MMU sa verifice prezenta segmentului adresat, ceea ce presupune testarea unui anumit bit in descriptorul segmentului: acest bit este 1 cind segmentul este incarcat si zero cind el este evacuat din memoria primara. Evacuarea se face mai rapid daca segmentul nu a fost modificat de la incarcarea in memorie, deoarece nu mai este necesara copierea sa in memoria de masa (de regula, acest lucru esteadevarat in cazul segmentelor de program). Acesta este motivul pentru care fiecare descriptor de segment are un fanion indicind starea de modificat / nemodificat a segmentului in discutie. Initial, fanionul are valoarea 0 si este inscris cu 1 la fiecare operatie de inscriere a unei locatii din codul segmentului.

O alta caracteristica importanta, chiar esentiala, pentru arhitectura hardware a unui sistem folosind tehnici de segmentare este data de posibilitatea intreruperii de catre CPU a executiei in mijlocul unei instructiuni si de reluare a instructiunii exact de la punctul la care a survenit intreruperea. In aceste cazuri mecanismele clasice pentru intreruperi si capcane nu mai sint necesare, deoarece ele sint activate doar la sfirsitul instructiunii, asa ca ele nu pot fi utilizate in tratarea unor defecte ale segmentelor , defecte care pot aparea oricind pe durata desfasurarii unei instructiuni. Instructiunea nu poate fi terminata daca a aparut un defect de segment , deoarece executia ei ar conduce la o eroare ireparabila.

Desi ultima varianta de tehnica de segmentare are mai multe avantaje (cum sint cele legate de posibilitatea de rulare a programelor al caror spatiu de adresare este mai mare decit cel al memoriei fizice disponibile), ea are si o serie de neajunsuri. Cel mai important este legat de dimensiunile neuniforme ale segmentelor, fapt ce complica gestionarea spatiului de memorie, deoarece, cind se pune problema incarcarii in memoria primara a unui segment de program, s-ar putea sa nu existe spatiul necesar, intrucit segmentul evacuat avea dimensiune mai mica, deci memoria eliberata este insuficienta. Rezulta fie necesitatea unor algoritmi complecsi de inlocuire a segmentelor,fie mentinerea permanenta a unui spatiu neutilizat de memorie. Una din tehnicile de rezolvare a acestei probleme este paginarea, care divizeaza spatiul programului in submultimi de aceleasi dimensiuni.

6.Spatiul de adresa al unui proces

Spatiul de adrese al unui proces reprezinta zona de memorie virtuala ce poate fi utilizata. Fiecare proces are un spatiu de adresa propriu. In cazurile in care doua procese partajeaza o zona de memorie, spatiul virtual este distinct, dar se mapeaza peste aceeasi zona de memorie fizica.

In sistemele de operare moderne, in spatiul virtual al fiecarui proces se mapeaza memoria nucleului, aceasta poate fi mapata fie la inceput, fie la sfarsitul spatiului de adresa.

“Cele 4 zone importante din spatiul de adresa al unui proces sunt zona de date, zona de cod, stiva si heap-ul. Dupa cum se observa si din figura, stiva si heap-ul sunt zonele care pot

creste. De fapt, aceste doua zone sunt dinamice si au sens doar in contextul unui proces. De partea cealalta, informatiile din zona de date si din zona de cod sunt descrise in executabil. “[1]

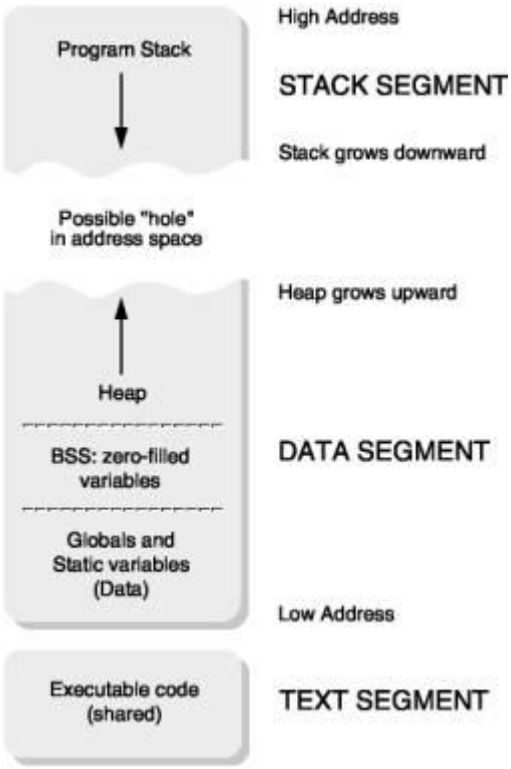


Fig 1.Spatiul de adrese pentru procese

Zona de cod

Segmentul de cod (denumit si `text segment`) reprezinta instructiunile in limbaj masina ale programului. Registrul de tip `instruction pointer` (IP) va referi adrese din zona de cod. Se citeste instructiunea indicata de catre IP, se decodifica si se interpreteaza, dupa care se incrementeaza contorul programului si se trece la urmatoarea instructiune.
Zona de cod este, de obicei, o zona read-only pentru ca procesul sa nu poata modifica propriile instructiuni prin folosirea gresita a unui pointer. Zona de cod este partajata intre toate procesele care ruleaza acelasi program. Astfel, o singura copie a codului este mapata in spatiul de adresa virtual al tuturor proceselor.

Zone de date

Zonele de date contin variabilele globale definite intr-un program si variabilele de tipul read-only. In functie de tipul de date exista mai multe subtipuri de zone de date.

.data

Zona `.data` contine variabilele globale si statice *initializate* la valori nenule ale unui program. De exemplu:

```
static int a = 3;
char b = 'a';
```

.bss

Zona `.bss` contine variabilele globale si statice *neinitializate* ale unui program. Inainte de executia codului, acest segment este initializat cu 0. De exemplu:

```
static int a;
char b;
```

In general aceste variabile nu vor fi prealocate in executabil, ci in momentul crearii procesului. Alocarea zonei .bss se face peste pagini fizice zero (zeroed frames).

.rodata

Zona .rodata contine informatie care poate fi doar citita, nu si modificata. Aici sunt stocate *constantele*:

```
const int a;
const char *ptr;
```

si *literalii*:

```
"Hello, World!"
"En Taro Adun!"
```

7.Stiva

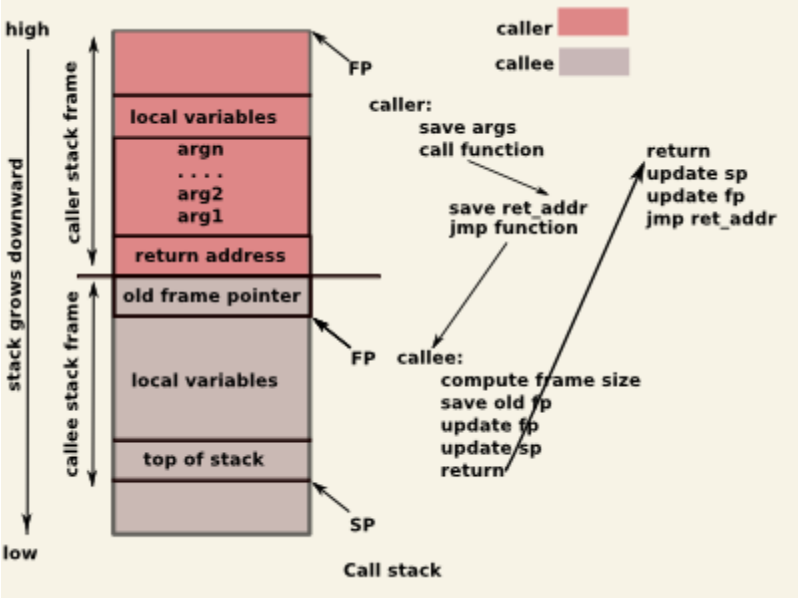
Stiva este o regiune dinamica în cadrul unui proces care este gestionata automat de compilator.

Aceasta este folosita pentru a stoca “stack frame-uri”. Fiecare apel al functiei creaza un nou ‘stack frame ‘, care contine :

1. variabile locale
2. argumentele functiei
3. adresa de retur

In marea majoritate a arhitecturilor stiva creste în jos si heap-ul în sus. Stiva crește la fiecare apel de functie si scade la fiecare revenire din functie.

In figura urmatoare este prezentata o vedere conceptuala asupra stivei in momentul apelului unei functii.



7.1.Segmentul de mapare a memoriei

Sub stiva se afla segmentul de mapare al memoriei. Aici kernel-ul mapeaza continutul fisierelor direct in memorie. Orice aplicatie poate sa solicite o astfel de mapare cu ajutorul apelului de sistem Linux *mmap()* sau *CreateFileMapping()/ MapViewOfFile()* in cazul sistemelor de operare Windows. Maparea de memorie este un mod performant si convenabil sa se creeze fisiere I/O, si astfel este folosit pentru incarcarea librariilor dinamice. Este posibil de asemenea sa se creeze mapari anonime de memorie care sa nu corespunda nici unui fisier, dar care sa fie folosit in schimb pentru date de program. In Linux, daca se solicita un bloc mare de memorie utilizand *malloc(),* libraria C va crea o mapare anonima in loc sa foloseasca memoria heap. Bloc mare desemneaza un bloc mai mare decat MMAP_THRESHOLD bytes, implicit 128 kB si ajustabil prin mallopt().

7.2.Heap-ul

Heap-ul este zona de memorie care este dedicata alocarii dinamice a memoriei. Este folosit pentru a aloca regiuni de memorie a caror dimensiune este determinata la runtime.

O proprietate comuna si stivei este faptul ca heap-ul este o regiune dinamica ce isi modifica dimensiunea. Ce o deosebeste de stiva,insa,este faptul ca heap-ul nu este gestionat de compilator. Datoria de a stii cata memorie trebuie alocata si de a retine cat a alocat si cat trebuie sa dealoce ii revine in totalitate compilatorului. Problemele ce apar frecvent in majoritatea programelor au legatura cu pierderea referintelor la zonele alocate (memory leaks) sau referirea de zone nealocate sau insuficient alocate (accese nevalide).

In limbaje precum Java, Lisp etc. unde nu există “pointer freedom”, eliberarea spațiului alocat se face automat prin intermediul unui garbage collector. Aici este prevenita problema pierderii referintelor, insa inca ramane nerezovata problema referirii zonelor nealocate.

8.Alocarea/Dealocarea memoriei

Alocarea memoriei se realizeaza static de compilator sau dinamic, in timpul executiei. *Alocarea statică* se realizeaza in segmentele de date pentru variabilele globale sau pentru literali.

In timpul procesului de executie, variabilele se aloca pe stiva sau in heap. Procesul de alocare pe stiva se realizează automat de compilator pentru variabilele locale unei functii (mai putin variabilele locale prefixate de identificatorul **static**).

Alocarea dinamica se realizează in heap. Alocarea dinamica are loc in momentul in care, in momentul compilarii, nu se stie cata memorie va fi necesara pentru o variabila/ structura/vector. Pentru a prevenii erori frecvent aparute in cazul alocarii dinamice, se recomanda alocarea ei statica in cazul in care se stie deja cata memorie ocupa o variabila.Pentru o fragmentare cat mai redusa a spatiului de adrese al procesului, in urma alocarilor si dealocarilor unor zone de diverse dimensiuni, alocatorul de memorie organizeaza segmentul de date alocate dinamic sub forma de *heap*, de unde si numele segmentului.

Dealocarea memoriei este echivalenta cu eliberarea zonei de memorie (este marcata ca fiind libera) alocate dinamic anterior.

In cazul in care se omite dealocarea unei zone de memorie, aceasta va ramane alocata pe intreaga durata de rulare a procesului. De fiecare data cand nu mai este nevoie de o zona de memorie, ea trebuie dealocata pentru a creste eficienta utilizarii spatiului de memorie.
Nu este neaparata nevoie sa se realizeze dealocarea diverselor zone inainte de un apel exit sau de finalizarea programului deoarece acestea sunt automat eliberate de sistemul de operare.

!!! Este posibil sa apara dificultati daca se incearca dealocarea aceleiasi regiuni de memorie in randuri repetate si sunt corupte datele interne de management al zonelor alocate dintr-un heap **!!!**

8.1.Alocarea memoriei în Linux

In Linux procesul de alocare a memoriei este realizat prin functiile de biblioteca malloc, calloc si realloc, iar dealocarea ei prin intermediul functiei free. FUnctiile respective reprezinta apeluri de biblioteca si rezolva cererile de alocare si dealocare de memorie, pe cat posibil.

Implementarea funcției malloc

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

Memoria alocata trebuie intotdeauna eliberata (free). Memoria alocata de proces se elibereaza automat odata cu terminarea procesului, insa, de exemplu, in cazul unui proces server cu o durata foarte mare de rulare si care nu elibereaza memoria alocata, acesta va ajunge sa ocupe toata memoria disponibila în sistem, avand astfel consecinte negative.

!!! Nu trebuie eliberata de doua ori aceeasi zona de memorie deoarece acest lucru va avea drept urmare coruperea tabelelor din malloc ceea ce va avea din nou consecințe negative. Intrucat functia free se intoarce imediat daca primeste ca parametru un pointer `NULL`, este recomandat ca după un apel free, pointer-ul să fie resetat la `NULL`.

Exemple de alocare folosin malloc :

```
int n = atoi(argv[1]);
char *str;

/* usually malloc receives the size argument like:
   num_elements * size_of_element */
str = malloc((n + 1) * sizeof(char));
if (NULL == str) {
        perror("malloc");
        exit(EXIT_FAILURE);
}

[...]

free(str);
str = NULL;
/* Creating an array of references to the arguments received by a program */
char **argv_no_exec;

/* allocate space for the array */
argv_no_exec = malloc((argc - 1) * sizeof(char*));
if (NULL == argv_no_exec) {
        perror("malloc");
        exit(EXIT_FAILURE);
}

/* set references to the program arguments */
for (i = 1; i < argc; i++)
        argv_no_exec[i-1] = argv[i];

[...]

free(argv_no_exec);
argv_no_exec = NULL;
```

Modificarea spatiului de memorie alocat cu un appel malloc sau calloc se realizeaza cu apelul realloc :

```
int *p;

p = malloc(n * sizeof(int));
if (NULL == p) {
        perror("malloc");
        exit(EXIT_FAILURE);
}

[...]

p = realloc(p, (n + extra) * sizeof(int));
```

```
[...]

free(p);
p = NULL;
```

Apelul calloc se foloseste pentru a aloca zone de memorie cu continut nul (care are numai valori de zero). Deosebirea de malloc este faptul ca apelul va primi două argumente: numărul de elemente și dimensiunea unui element.

```
list_t *list_v; /* list_t could be any C type ( except void ) */

list_v = calloc(n, sizeof(list_t));
if (NULL == list_v) {
        perror("calloc");
        exit(EXIT_FAILURE);
}

[...]

free(list_v);
list_v = NULL;
```

Conform standardului C, este redundant sa se faca *cast* la valoarea întoarsa de malloc.

```
int *p = (int *)malloc(10 * sizeof(int));
```

malloc intoarce *void* * care în C se converteste automat la tipul dorit. Asadar, odata ce se face cast, iar headerul `stdlib.h` util pentru functia malloc nu este inclus, nu se va da eroare! Pe unele arhitecturi, acest caz poate conduce la un comportament nedefinit. Deosebirea lui C++ fata de C conta in faptul ca in C++ este nevoie de cast.

8.2.Alocarea memoriei în Windows

In Windows un proces isi poate creea mai multe obiecte `Heap` exceptand `Heap`-ul implicit al procesului. Acest lucru este foarte util in momentul in care o aplicatie aloca și dealoca foarte multe zone de memorie cu cateva dimensiuni fixe. Aplicatia isi poate creea cate un `Heap` pentru fiecare dimensiune și, in interiorul fiecarui `Heap`, poate aloca zone de dimensiuni egale micsorand la maxim fragmentarea heap-ului.

Pentru a creea si a distruge un `Heap` se folosesc functiile HeapCreate și HeapDestroy:

```
HANDLE HeapCreate(
        DWORD flOptions,
        SIZE_T dwInitialSize,
        SIZE_T dwMaximumSize
);

BOOL HeapDestroy(
        HANDLE hHeap
);
```

Pentru a obtine un descriptor al heap-ului implicit al procesului (daca nu dorim crearea altor heap-uri) se foloseste functia GetProcessHeap. Daca dorim sa obtinem descriptorii tuturor heap-urilor din proces, utilizam functia GetProcessHeaps.

Există si functii care enumera toate blocurile alocate intr-un heap, valideaza unul sau toate blocurile alocate intr-un heap sau intorc dimensiunea unui bloc pe baza descriptorului de heap si a adresei blocului: HeapWalk, HeapValidate, HeapSize.

Pentru alocarea, dealocarea sau redimensionarea unui bloc de memorie din `Heap`, Windows dispune de functiile HeapAlloc, HeapFree, respectiv HeapReAlloc, cu formele de mai jos:

```
LPVOID HeapAlloc(
        HANDLE hHeap,
        DWORD dwFlags,
        SIZE_T dwBytes
);

BOOL HeapFree(
        HANDLE hHeap,
        DWORD dwFlags,
        LPVOID lpMem
);

LPVOID HeapReAlloc(
        HANDLE hHeap,
        DWORD dwFlags,
        LPVOID lpMem,
        SIZE_T dwBytes
);
```

Exemplu alocare, dealocare, redimensionare (HeapAlloc, HeapFree, HeapReAlloc)

```
#include <windows.h>
#include "utils.h"

/* Example of matrix allocation */

int main(void)
{
        HANDLE processHeap;
        DWORD **mat;
        DWORD i, j, m = 10, n = 10;

        processHeap = GetProcessHeap();
        DIE (processHeap == NULL, "GetProcessHeap");

        mat = HeapAlloc(processHeap, 0, m * sizeof(*mat));
        DIE (mat == NULL, "HeapAlloc");

        for (i = 0; i < n; i++) {
                mat[i] = HeapAlloc(processHeap, 0, n * sizeof(**mat));
                if (mat[i] == NULL) {
                        PrintLastError("HeapAlloc failed");
                        goto freeMem; /* free previously allocated memory */
                }
        }
```

```
        /* do work */

freeMem:
        for (j = 0; j < i; j++)
                HeapFree(processHeap, 0, mat[j]);
        HeapFree(processHeap, 0, mat);

        return 0;
}
```

Pe arhitecturile care ruleaza Windows putem folosi si functiile bibliotecii standard C pentru gestiunea memoriei: malloc, realloc, calloc, free, insa apelurile de sistem specifice Windows ofera functionalitati suplimentare fara sa implice legarea bibliotecii standard C în executabil.

BIBLIOGRAFIE:

1.Platforme laborator Facultatea de Automatica(Bucuresti,Brasov)

2.Curs SO

3.www.scritube.com

4.www.wikipedia.com

5.Radu Radescu-Arhitectura Sistemelor de calcul

Cuprins

Simion Bogdan: capitolele 1-6

Chelu Marius: capitolele 7-8