Memoria virtuala

Contents

Memoria virtuală

Mecanismul de memorie virtuală este folosit de către nucleul sistemului de operare pentru a implementa o politică eficientă de gestiune a memoriei. Astfel, cu toate că aplicaiile folosesc în mod curent memoria virtuală, ele nu fac acest lucru în mod explicit. Există însă câteva cazuri în care aplicaiile folosesc memoria virtuală în mod explicit.

Sistemul de operare oferă primitive de mapare a fiierelor, a memoriei sau a dispozitivelor în spaiul de adresă al unui proces.

- **Maparea fiierelor** în memorie este folosită în unele sisteme de operare pentru a implementa mecanisme de memorie partajată. Acest mecanism face de asemenea posibilă implementarea paginării la cerere i a bibliotecilor partajate.

- **Maparea memoriei** în spaiul de adresă este folositoare atunci când un proces dorete să aloce o cantitate mare de memorie.
- **Maparea dispozitivelor** este folositoare atunci când un proces dorete să folosească direct memoria unui dispozitiv cum ar fi placa video.

Linux

Funciile cu ajutorul cărora se pot face cereri explicite asupra memoriei virtuale sunt funciile din familia `mmap(2)`. Funciile folosesc ca unitate minimă de alocare pagina (adică se poate aloca numai un număr întreg de pagini, iar adresele trebuie să fie aliniate corespunzător).

Maparea fiierelor

În urma mapării unui fiier în spaiul de adresă al unui proces, accesul la acest fiier se poate face similar cu accesarea datelor dintr-un vector. Eficiena metodei vine din faptul că zona de memorie este gestionată similar cu memoria virtuală, supunându-se regulilor de evacuare pe disc atunci când memoria devine insuficientă (în felul acesta se poate lucra cu mapări care depăesc dimensiunea efectivă a memoriei fizice).

Observaie

> Nu orice descriptor de fiier poate fi mapat în memorie. Socket-urile, pipe-urile, majoritatea dispozitivelor nu permit decât accesul secvenial i sunt incompatibile din această cauză cu conceptele de mapare. Există cazuri în care fiiere obinuite nu pot fi mapate (spre exemplu, dacă nu au fost deschise pentru a putea fi citite; pentru mai multe informaii: **man mmap**).

mmap

Prototipul funciei ce permite maparea unui fiier în spaiul de adresă al unui proces este următorul:

```
void *mmap(void *start, size_t length, int prot,
           int flags, int fd, off_t offset);
```

Funcia va întoarce un pointer spre o zonă de memorie din spaiul de adresă al procesului, zonă în care a fost mapat fiierul descris de descriptorul *fd*, începând cu offset-ul *offset* . Folosirea parametrului *start* permite propunerea unei anumite zone de memorie la care să se facă maparea; Folosirea valorii *NULL* pentru parametrul *start* indică lipsa vreunei preferine în ceea ce privete zona în care se va face alocarea. Adresa precizată prin parametrul *start* trebuie să fie multiplu de *dimensiunea unei pagini*. Dacă sistemul de operare nu poate să mapeze fiierul la adresa cerută, atunci îl va mapa la altă adresă. Adresa la care se mapează fiierul este întoarsă de funcie

Parametrul *prot* specifică tipul de acces care se dorete; poate fi `PROT_READ`, `PROT_WRITE`, `PROT_EXEC` sau `PROT_NONE`; dacă zona e folosită altfel decât s-a declarat se va genera un semnal `SIGSEGV`

Parametrul *flags* permite stabilirea tipului de mapare ce se dorete; poate lua următoarele valori (combinate prin SAU pe bii; trebuie să existe cel puin una: fie `MAP_PRIVATE`, fie `MAP_SHARED`):

- `MAP_PRIVATE` - se folosete o politică de tip copy-on-write; zona va conine iniial o copie a fiierului, dar scrierile nu sunt făcute în fiier; modificările nu vor fi vizibile în alte procese dacă există mai multe

procese care au făcut `mmap` pe aceeai zonă din acelai fiier

- `MAP_SHARED` - scrierile sunt actualizate imediat în toate mapările existente (în acest fel toate procesele care au realizat mapări vor vedea modificările); pentru ca modificările să fie vizibile i pentru un proces ce utilizează read/write se poate folosi `msync`; altfel actualizarea va avea loc la un moment de timp nespecificat
- `MAP_FIXED` - dacă nu se poate face alocarea la adresa specificată de *start* apelul va eua
- `MAP_LOCKED` - se va bloca paginarea pe această zonă
- `MAP_ANONYMOUS` - se mapează memorie (argumentele *fd* i *offset* sunt ignorate)

În caz de succes, funcia întoarce un pointer către zona din spaiul de adresă al procesului unde a fost mapat fiierul. În caz de insucces se întoarce `MAP_FAILED`, eroarea fiind semnalată în `errno`. De exemplu, pentru alocare de memorie se poate utiliza `MAP_PRIVATE | MAP_ANONYMOUS`.

Observaie

Operaia `mmap` incrementează contorul de utilizări ale fiierului deschis în *fd*. La închiderea lui *fd* maparea va supravieui. Acest lucru are ca efect menținerea unui fiier chiar după ce el este ters, dacă fiierul este mapat. Fiind asociată spaiului de adresă al unui proces, maparea va fi distrusă atunci când se termină procesul.
Adresa întoarsă este multiplu de *dimensiunea unei pagini*.

`msync`

Pentru a declana în mod explicit sincronizarea fiierului cu maparea din memorie este disponibilă următoarea funcie:

```
int msync(void *start, size_t length, int flags);
```

unde *flags* poate fi

`MS_SYNC`
: datele vor fi scrise în fiier i abia apoi funcia se va termina

`MS_ASYNC`
: este iniiată secvena de salvare dar nu se ateaptă terminarea ei

`MS_INVALIDATE`
: se invalidează mapările zonei din alte procese, pentru a fora recitirea paginii în toate celelalte procese la următorul acces.

Dacă operaia are succes se întoarce 0 i se actualizează poriunea din fiier care corespunde zonei de memorie `start` de lungime `length`. Dacă `msync` euează se întoarce valoarea `-1`. Erorile sunt specificate în `errno`.

Alocare de memorie în spaiul de adresă al procesului

În UNIX, tradiional, pentru alocarea *memoriei dinamice* se folosete apelul de sistem `brk`. Acest apel crete sau descrete zona de heap asociată procesului. Odată cu oferirea către aplicaii a unor apeluri de sistem de gestiune a memoriei virtuale (`mmap`), a existat posibilitatea ca procesele să aloce memorie folosind aceste noi apeluri

de sistem. Practic, procesele pot mapa în spaiul de adresă... memorie, nu fiiere.

Procesele pot cere alocarea unei zone de memorie de la o anumită adresă din spaiul de adresare, chiar i cu o anumită politică de acces (citire, scriere sau execuie). În UNIX acest lucru se face tot prin intermediul funciei `mmap`. Pentru acest lucru parametrul de flag-uri trebuie să conină flag-ul `MAP_ANONYMOUS`. În acest caz, descriptorul de fiier i offset-ul sunt ignorate. În rest, funcia primete argumente cu aceeai semnificaie ca cele discutate în seciunea anterioară.

Maparea dispozitivelor

Există chiar i posibilitatea ca aplicaiile să mapeze în spaiul de adresă al unui proces un dispozitiv de intrare-ieire. Acest lucru este util de exemplu pentru plăcile video: o aplicaie poate mapa în spaiul de adresă memoria plăcii video. În UNIX, dispozitivele fiind reprezentate prin fiiere, pentru a realiza acest lucru nu trebuie decât să deschidem fiierul asociat dispozitivului i să-l folosim într-un apel `mmap`. Atenie însă, nu toate dispozitivele pot fi mapate în memorie, iar atunci când pot fi mapate, ce înseamnă acest lucru depinde de dispozitiv.

Un alt exemplu de dispozitiv care poate fi mapat este chiar... memoria. În Linux se poate folosi fiierul /dev/zero pentru a face mapări de memorie, ca i când s-ar folosi flag-ul `MAP_ANONYMOUS`.

Demaparea unei zone din spaiul de adresă

Dacă se dorete demaparea unei zone din spaiul de adresă al procesului se poate folosi funcia `munmap`:

```
int munmap(void *start, size_t length);
```

start este adresa primei pagini ce va fi demapate (trebuie sa fie multiplu de *dimensiunea unei pagini*). Dacă *length* nu este o dimensiune care reprezintă un număr întreg de pagini, va fi rotunjit superior. Zona poate să conina bucăi deja demapate. Se pot astfel demapa mai multe zone în acelai timp.

Redimensionarea unei zone mapate

Pentru a executa operaii de redimensionare a zonei mapate se poate utiliza următoarea funcie:

```
void *mremap(void *old_address, size_t old_size, size_t new_size, unsigned long flags);
```

Zona pe care *old_adress* i *old_size* o descriu trebuie să aparină unei singure mapări. O singură opiune este disponibilă pentru *flags*: `MREMAP_MAYMOVE` care arată că este în regulă ca pentru obinerea noii mapări să se realizeze o noua mapare într-o altă zonă de memorie (vechea zona fiind dezalocată). Dacă totul se termină cu bine atunci funcia întoarce un pointer spre noua zonă. In caz de eroare se întoarce `MAP_FAILED`.

Schimbarea proteciei unei zone mapate

Uneori este nevoie ca modul (drepturile de acces) în care a fost mapată o zonă să fie schimbat. Pentru acest lucru se poate folosi funcia `mprotect`:

```
int mprotect(const void *addr, size_t len, int prot);
```

Funcia primete ca parametri intervalul de adrese [*addr*, *addr* + *len* - 1] i noile drepturi de access (`PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, `PROT_NONE`). Ca i la `munmap` *addr* trebuie sa fie multipla de *dimensiunea unei pagini*. Funcia va schimba protecia pentru toate paginile care conin cel puin un byte în intervalul specificat. Funcia întoarce 0 dacă operaia s-a încheiat cu succes sau -1 în caz contrar, erorile fiind descrise de `errno`.

Exemplu

```
int fd = open("fisier", O_RDWR);
void *p = mmap(NULL, 2*getpagesize(), PROT_NONE, MAP_SHARED, fd, 0);
// *(char*)p = 'a'; // segv fault
mprotect(p, 2*getpagesize(), PROT_WRITE);
*(char*)p = 'a';
munmap(p, 2*getpagesize());
```

Optimizări

Pentru ca sistemul de operare să poată implementa cât mai eficient accesele la o zona de memorie mapată, programatorul poate să informeze kernel-ul (prin apelul de sistem `madvise(2)`) despre modul în care zona va fi folosită. `madvise(2)` e utilă mai ales la când în spatele memoriei virtuale se află un dispozitiv fizic (de ex. când se mapează fiiere de pe hard-disk, kernelul poate citi în avans pagini de pe disk, reducând latena datorată poziionării capului de citire). Prototipul funciei este următorul:

```
int madvise(void *start, size_t length, int advice);
```

unde valorile acceptate pentru *advice* sunt:

`MADV_NORMAL`
: regiunea este una obinuită i care nu are nevoie de un tratament special

`MADV_RANDOM`
: regiunea va fi accesată în mod aleator; sistemul de operare nu va citi în avans pagini

`MADV_SEQUENTIAL`
: regiunea va fi accesată în mod secvenial; sistemul de operare ar putea citi în avans pagini

`MADV_WILLNEED`
: regiunea va fi utilizată undeva în viitorul apropiat (nucleul poate decide să preîncarce paginile în memorie)

`MADV_DONTNEED`
: regiunea nu va mai fi utilizată; nucleul poate să elibereze zona alocată din memorie, dar zona nu este demapată; nu se garantează păstrarea datelor la accesări ulterioare

Dacă operaia are loc cu succes se întoarce 0. Dacă apare o eroare se întoarce -1, erorile fiind descrise de `errno`.

Blocarea paginării

Există o categorie de procese care trebuie să execute anumite aciuni la momente de timp bine determinate, pentru a se păstra calitatea execuiei. Pentru exemplificare putem considera un player audio sau video. Sau un program ce controlează mersul unui robot biped. Problema cu acest gen de procese este dată de faptul că dacă o anumită pagină nu este prezentă în memorie, va dura un timp până ce ea va fi adusă. Pentru a contracara aceste probleme, sistemele UNIX pun la dispoziie apelurile `mlock` i `mlockall`.

```
int mlock(const void *addr, size_t len);
int mlockall(int flags);
```

Funcia `mlock` va bloca paginarea (nu se va mai face swapout) paginilor incluse în intervalul [*addr*, *addr* + *len* - 1]. Funcia `mlockall` va bloca paginarea tuturor paginilor procesului, în funcie de flag-uri:

`MCL_CURRENT`
: se va bloca paginarea tuturor paginilor mapate în spaiul de adresă al procesului la momentul apelului

`MCL_FUTURE`
: se va bloca paginarea noilor pagini mapate în spaiul de adresă al procesului; aici intră noi mapări realizate cu funcia `mmap` dar i paginile de stivă mapate automat de sistem

Notă
: Flag-ul `MCL_FUTURE` nu garantează faptul că paginile de stivă vor fi automat mapate în sistem. Dacă procesul depăete limita de memorie impusă de sistem, va primi semnalul SIGSEGV. Pentru a nu se ajunge în astfel de situaii, programul trebuie să folosească `mlockall(MCL_CURRENT | MCL_FUTURE)` i apoi să aloce dimensiunea maximă a stivei pe care urmează să o folosească (prin declararea unei variabile locale, un vector de exemplu, i accesarea completă a acesteia).

Există bineînes i funcii ce readuc lucrurile la normal:

```
int munlock(const void *addr, size_t len);
int munlockall(void);
```

Astfel funcia `munlock` va reporni mecanismul de paginare al tuturor paginilor din intervalul [*addr*, *addr* + *len* - 1], iar funcia `munlockall` face acelai lucru pentru toate paginile procesului, atât curente cât i viitoare. Trebuie notat faptul că dacă s-au efectuat mai multe apeluri `mlock` sau `mlockall` este suficient un singur apel `munlock` sau `munlockall` pentru a reactiva paginarea.

Excepii

Atunci când se detectează o încălcare a proteciei la accesul la memorie se va trimite semnalul `SIGSEGV` sau `SIGBUS` procesului. După cum am văzut atunci când am discutat despre semnale, semnalul poate fi tratat cu două tipuri de funcii pe care aici o să le denumim `signal` i `sigaction`. Funcia de tip `sigaction` va primi ca parametru o structură `siginfo_t`. În cazul semnalelor ce tratează excepii cauzate de un acces incorect la memorie, următoarele câmpuri din această structură sunt setate:

`si_signo`
: setat la `SIGSEGV` sau `SIGBUS`

`si_code`
: pentru `SIGSEGV` poate fi `SEGV_MAPPER` pentru a arăta că zona accesată nu este mapată în spaiul de adresă al procesului, sau `SEGV_ACCERR` pentru a arăta că zona este mapată dar a fost accesată necorespunzător; pentru `SIGBUS` poate fi `BUS_ADRALN` pentru a arăta că s-a făcut un acces nealiniat la memorie, `BUS_ADRERR` pentru a arăta că s-a încercat accesarea unei adrese fizice inexistente sau `BUS_OBJERR` pentru a indica o eroare hardware

`si_addr`
: adresa care a generat excepia

ElectricFence

ElectricFence este un pachet ce ajută programatorii la depanarea problemelor de tipul buffer overrun. Aceste probleme sunt cauzate de faptul că anumite date sunt suprascrise fiindcă nu se fac verificări când se modifică date adiacente. Soluia folosită de ElectricFence este înlocuirea apelurilor standard malloc i free cu implementări proprii. ElectricFence va plasa zona de memorie alocată în spaiul de adrese al procesului, astfel încât ea să fie mărginită de pagini neaccesibile (protejate la scriere i citire).

Din păcate, sistemul de operare i arhitectura procesorului limiteaz dimensiunea paginii la cel putin 1-4K, astfel încât dac zona de memorie alocată nu este multiplu de această dimensiune, există posibilitatea ca programul să poată citi sau scrie i în zone în care nu ar trebui, fără ca sistemul de operare să oprească executia programului. Pentru a preveni situaii de acestă natură alocarea zonelor de memorie de către ElectricFence se face astfel încât ele sunt la limita superioară a unei pagini, după care urmează o pagină neaccesibilă. Problema cu o astfel de abordare este că nu previne situaiile de *buffer underrun*, în care datele sunt citite sau scrise peste limita inferioar.

Pentru a putea verifica i astfel de situaii utilizatorul trebuie să definească variabila de mediu `EF_PROTECT_BELOW` înainte de rula programul. În acest caz, ElectricFence va plasa zona de memorie alocată la începutul unei pagini, pagină care la rândul ei este plasată după o pagină inaccesibilă procesului.

De ce este importantă detectarea situaiilor de buffer overrun? Asa cum am explicat i în seciunea precedentă, astfel de situaii vor produce în cele din urmă erori, dar la un moment de timp ulterior, astfel încât este imposibil să determinăm cauza erorii cu mijloace de depanare obinuite. În plus, situaiile buffer overrun pot suprascrie nu numai variabile, ci i alte date importante pentru stabilitatea programului cum ar fi datele de control folosite de rutinele malloc i free. Pachetul ElectricFence poate determina erorile de buffer overrun doar dacă acestea apar în memoria alocată dinamic (adică în zona *heap*) cu rutinele *malloc* i *free*. Pentru a folosi pachetul ElectricFence utilizatorul trebuie să folosească la compilarea programului biblioteca `libefence`. Pentru a vedea utilitatea acestui pachet să analizăm programul de mai jos:

Exemplu 15. exemplul-8.c

```
#include <stdio.h>
#include <malloc.h>

int main()
{
int i;
int *data_1, *data_2;

      =mdata_1alloc(11*sizeof(int));

for(i=0; i<=11; i++)
      [i]=i;    data_1
```

```
      =malloc(11*sizeof(int));

for(i=0; i<=11; i++)
      [i]=11-i; data_2

for(i=0; i<=11; i++)
printf("%d %d\n", data_1[i], data_2[i]);

free(data_1); free(data_2);

return 0;
}
```

Aparent totul pare în regulă. La execuia programului însă obinem următorul output:

```
[tavi@dhcp-48 intro]$ gcc -Wall -g exemplul-8.c
[tavi@dhcp-48 intro]$ ./a.out
 0 11
 1 10
 2  9
 3  8
 4  7
 5  6
 6  5
 7  4
-8  3
 8  2
 8  1
-7  0
```

Ceva este clar în neregulă. Dacă folosim biblioteca efence i GDB eroarea va fi vizibilă imediat :

```
[tavi@dhcp-48 intro]$ gcc -Wall -g exemplul-8.c -lefence
[tavi@dhcp-48 intro]$ gdb a.out
(gdb) run
Starting program: /home/tavi/cursuri/so/lab/draft/intro/a.out
[New Thread 1024 (LWP 20752)]

  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens
        <bruce@perens.com>

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 1024 (LWP 20752)]
0x08048594 in main () at exemplul-8.c:13
13                      data_1[i]=i;
(gdb) print i
$1 = 11
(gdb)
```

Se observă că eroarea apare în momentul în care încercăm să iniializăm al 12-lea element al vectorului, dei vectorul nu are decât 11 elemente.

Pentru mai multe informatii despre pachetul ElectricFence consultai pagina de manual (**man efence**).

Windows

În Windows funciile de control al memoriei virtuale sau mai bine zis al spaiului de adresă al unui proces nu mai sunt grupate ca în cazul Unix într-o singură primitivă oferită de sistemul de operare. Avem funcii pentru maparea fiierelor în memorie i funcii pentru alocarea de memorie fizică în spaiul de adresă al unui proces.

Maparea fiierelor

Pentru a mapa un fiier în spaiul de adresă al unui proces trebuie mai întâi creat un handle către un obiect de tipul `FileMapping` i apoi realizată efectiv maparea. Funciile `CreateFileMapping` i `MapViewOfFile` au mai fost prezentate atunci când s-a discutat despre memoria partajată. Le prezentăm din nou pe scurt aici:

```
HANDLE CreateFileMapping(
   HANDLE hFile,
   LPSECURITY_ATTRIBUTES lpAttributes,
   DWORD flProtect,
   DWORD dwMaximumSizeHigh,
   DWORD dwMaximumSizeLow,
   LPCTSTR lpName
 );
```

Funcia primete ca parametri handle-ul fiierului care se dorete a fi mapat, atribute de securitate care controlează accesul la hande-ul obiectului `FileMapping` creat, tipul mapării (`PAGE_READONLY`, `PAGE_READWRITE`, `PAGE_WRITECOPY` pentru copy-on-write) i dimensiunea maximă care poate fi mapată cu ajutorul funciei `MapViewOfFile`. Opional se poate specifica i un ir care să identifice obiectul `FileMapping` creat. Dacă mai există un obiect de acest tip, funcia `CreateFileMapping` nu va crea unul nou, ci îl va folosi pe cel existent. Atenie însă, obiectul trebuie să fi fost creat cu drepturi care să permită procesului apelant să îl deschidă. Pentru deschiderea unui obiect de tip `FileMapping` deja creat se mai poate folosi funcia `OpenFileMapping`, funcie care a fost de asemenea prezentată atunci când s-a discutat despre memoria partajată.

```
LPVOID MapViewOfFile(
   HANDLE hFileMappingObject,
   DWORD dwDesiredAccess,
   DWORD dwFileOffsetHigh,
   DWORD dwFileOffsetLow,
   SIZE_T dwNumberOfBytesToMap
 );
```

Funcia primete ca parametri un handle către un obiect de tip `FileMapping`, modul de acces la zona mapată (`FILE_MAP_READ`, `FILE_MAP_WRITE`, `FILE_MAP_COPY` pentru copy-on-write), offset-ul în fiier de unde începe maparea i numărul de octei de mapat. Funcia va întoarce un pointer în spaiul de adresă al procesului, la zona mapată.

Alocare de memorie în spaiul de adresă al procesului

Pentru alocarea de memorie în spaiul de adresă al procesului se pot folosi funciile `VirtualAlloc` sau `VirtualAllocEx`:

```
LPVOID VirtualAlloc(
   LPVOID lpAddress,
```

```
   SIZE_T dwSize,
   DWORD flAllocationType,
   DWORD flProtect
 );
 LPVOID VirtualAllocEx(
   HANDLE hProcess,
   LPVOID lpAddress,
   SIZE_T dwSize,
   DWORD flAllocationType,
   DWORD flProtect
 );
```

Cu funcia `VirtualAllocEx` se poate aloca memorie în spaiul de adresă al unui proces arbitrar, specificat în parametrul *hProcess*. Procesul curent trebuie să aibă drepturi corespunzătoare asupra procesului pe care se încearcă operaia (PROCESS_VM_OPERATION). Funciile întorc un pointer către adresa de start, iar parametrii ateptai de funcii sunt descrii mai jos:

fAllocationType

specifică tipul operaiei: rezervare (`MEM_RESERVE`), alocare (`MEM_COMMIT`) sau renunare la zonă (`MEM_RESET`); rezervarea unei zone înseamnă de fapt "punerea deoparte" a unui interval din spaiul de adrese virtuale al procesului, fără a se aloca însă memorie fizică; dacă se folosete MEM_COMMIT, se alocă efectiv memorie (dar doar dacă în prealabil zona vizată a fost rezervată); atunci când se renună la zonă nucleul poate face discard la paginile din zonă, fără a face însă dezalocarea lor; după această operaie datele nu se păstrează

lpAddress

adresa din spaiul de adresă de unde începe alocarea; trebuie să fie multiplu de 4KB pentru alocare i 64KB pentru rezervare; dacă este *NULL* sistemul va furniza automat o adresă

dwSize

dimensiunea zonei

flProtect

specifică modul de acces permis la zona alocată: `PAGE_EXECUTE`, `PAGE_EXECUTE_READ`, `PAGE_EXECUTE_READWRITE`, `PAGE_EXECUTE_WRITECOPY`, `PAGE_READONLY`, `PAGE_READWRITE`, `PAGE_WRITECOPY`, `PAGE_NOACCESS`, `PAGE_GUARD`, `PAGE_NOCACHE`. Modurile `_WRITECOPY` arată că se va folosi mecanismul copy-on-write. Modul `PAGE_GUARD` specifică faptul că la primul acces la o astfel de zonă se va genera o excepie `STATUS_GUARD_PAGE`. `PAGE_GUARD` i `PAGE_NOCACHE` se pot folosi împreună cu celelalte moduri

Demaparea unei zone din spaiul de adresă

Pentru demaparea unei fiier mapat în memorie se folosete funcia `UnmapViewOfFile`:

```
BOOL UnmapViewOfFile(
   LPCVOID lpBaseAddress
 );
```

Funcia primete adresa de început a zonei, i întoarce TRUE dacă operaia s-a încheiat cu succes sau FALSE în caz contrar, caz în care eroarea poate fi aflată cu funcia `GetLastError`.

Pentru demaparea unei zone de memorie din spaiul de adresă se folosesc funciile `VirtualFree` i `VirtualFreeEx`:

```
BOOL VirtualFree(
   LPVOID lpAddress,
   SIZE_T dwSize,
   DWORD dwFreeType
 );
 BOOL VirtualFreeEx(
   HANDLE hProcess,
   LPVOID lpAddress,
   SIZE_T dwSize,
   DWORD dwFreeType
 );
```

Funcia `VirtualFreeEx` va dezaloca o zonă de memorie din spaiul de adresă al unui proces arbitrar, specificat în parametrul *hProcess*. Procesul curent trebuie să aibă drepturi corespunzătoare asupra procesului pe care se încearcă operaia (`PROCESS_VM_OPERATION`).

Parametrii *lpAddress* i *dwSize* identifică zona de dezalocat. *dwFreeType* specifică tipul operaiei: `MEM_DECOMMIT`, `MEM_RELEASE`. Prima operaie va demapa paginile din spaiul de adresă, dar ele vor rămâne rezervate. Cea de-a doua operaie va anula rezervarea întregii zone "puse deoparte" anterior, astfel încât adresa de start trebuie să coincidă cu adresa de start a zonei rezervate, iar dimensiunea trebuie să fie setată pe 0.

Schimbarea proteciei unei zone mapate

În Windows, schimbarea drepturilor de access a unei zone mapate se poate face cu ajutorul funciilor `VirtualProtect` i `VirtualProtectEx`:

```
BOOL VirtualProtect(
   LPVOID lpAddress,
   SIZE_T dwSize,
   DWORD flNewProtect,
   PDWORD lpflOldProtect
 );
 BOOL VirtualProtectEx(
   HANDLE hProcess,
   LPVOID lpAddress,
   SIZE_T dwSize,
   DWORD flNewProtect,
   PDWORD lpflOldProtect
 );
```

Funciile vor schimba protecia paginilor care au măcar un octet în intervalul [*lpAddress*, *lpAddress* + *dwSize* - 1] la cea specificată în *flNewProtect*. Vechile drepturi de acces sunt salvate în *lpfOldProtect*.

Interogarea zonelor mapate

Pentru a afla informaii despre o zonă mapată în spaiul de adresă al unui proces se pot folosi funciile `VirtualQuery` i `VirtualQueryEx`. Ele vor oferi informaii apelantului despre adresa de start a zonei, protecie, dimensiune etc.

```
DWORD VirtualQuery(
   LPCVOID lpAddress,
   PMEMORY_BASIC_INFORMATION lpBuffer,
   SIZE_T dwLength
 );
 DWORD VirtualQueryEx(
   HANDLE hProcess,
   LPCVOID lpAddress,
   PMEMORY_BASIC_INFORMATION lpBuffer,
   SIZE_T dwLength
 );
```

Funciile primesc ca parametri o adresă din cadrul zonei ce se dorete a fi interogată, un pointer către un buffer alocat ce va primi informaii despre zonă i întorc numărul de octei scrii în buffer. Dacă funcia întoarce 0 înseamnă că nici o informaie nu a fost furnizată. Acest lucru se întâmplă dacă funciei îi este pasată o adresă din spaiul kernel.

Informaiile primite vor descrie două zone: zona alocată (cu `VirtualAlloc`) în care este inclusă adresa dată, i zona care conine pagini de acelai fel (cu aceeai protecie i stare) în care este inclusă adresa dată:

```
typedef struct _MEMORY_BASIC_INFORMATION {
   PVOID BaseAddress;
   PVOID AllocationBase;
   DWORD AllocationProtect;
   SIZE_T RegionSize;
   DWORD State;
   DWORD Protect;
   DWORD Type;
 } MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

Câmpurile `AllocationBase` i `AllocationProtect` se referă la zona alocată, iar `BaseAddress`, `RegionSize`, `Type` i `Protect` la zona ce conine pagini de acelai fel. `State` indică starea paginilor din zonă: MEM_COMMIT pentru zonă alocată, MEM_RESERVED pentru zonă rezervată i MEM_FREE pentru zonă nealocată. `Type` indică dacă în zonă este mapat un fiier (MEM_IMAGE sau MEM_MAPPED) sau nu, i indică de asemenea dacă zona este partajată sau nu (MEM_PRIVATE).

Blocarea paginării

Pentru blocarea paginării pentru un set de pagini (nu se va mai face swapout - în consecină apelurile ulterioare nu mai produc page fault), sistemul de operare Windows pune la dispoziia utilizatorilor funciile `VirtualLock` i `VirtualLockEx`:

```
BOOL VirtualLock(
   LPVOID lpAddress,
   SIZE_T dwSize
 );
 BOOL VirtualLockEx(
   HANDLE hProcess,
   LPVOID lpAddress,
   SIZE_T dwSize
 );
```

Funciile primesc prin parametri un interval de pagini (alcătuit din paginile care au măcar un octet în intervalul [*lpAddress*, *lpAddess* + *dwSize*]) pentru care se vrea blocarea paginării. Dacă reuesc, vor întoare `TRUE`, altfel `FALSE` i eroarea se poate afla cu `GetLastError`.

Funciile pentru repornirea paginării sunt:

```
BOOL VirtualUnlock(
   LPVOID lpAddress,
   SIZE_T dwSize
 );
 BOOL VirtualUnlockEx(
   HANDLE hProcess,
   LPVOID lpAddress,
   SIZE_T dwSize
 );
```

Excepii

Atunci când sistemul de operare detectează accese incorecte la memorie, va genera o excepie către procesul care a efectuat accesul. Pentru tratarea excepiei se pot folosi construcii `__try` i `__except`, pentru care este necesar suport din partea compilatorului, sau se poate folosi funcia `AddVectoredExceptionHandler`. Vom discuta în continuare ultima variantă.

```
PVOID AddVectoredExceptionHandler(
   ULONG FirstHandler,
   PVECTORED_EXCEPTION_HANDLER VectoredHandler
 );
 ULONG RemoveVectoredExceptionHandler(
   PVOID VectoredHandlerHandle
 );
```

Funcia `AddVectoredExceptionHandler` va adăuga pe lista funciilor de executat atunci când se generează o excepie, pe cea primită ca parametru în `VectoredHandler`. Parametrul `FirstHandler` indică dacă funcia dorete sa fie adăugată la începutul listei sau la sfârit. Funcia de tratare a excepiilor trebuie să aibă următoarea semnătură:

```
LONG WINAPI VectoredHandler(
   PEXCEPTION_POINTERS ExceptionInfo
 );

 typedef struct _EXCEPTION_POINTERS {
  PEXCEPTION_RECORD ExceptionRecord;
  PCONTEXT ContextRecord;
 } EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;

 typedef struct _EXCEPTION_RECORD {
  DWORD ExceptionCode;
  DWORD ExceptionFlags;
  struct _EXCEPTION_RECORD* ExceptionRecord;
  PVOID ExceptionAddress;
  DWORD NumberParameters;
  ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
 } EXCEPTION_RECORD, *PEXCEPTION_RECORD;
```

În cazul unor excepii cauzate de un acces invalid la memorie, `ExceptionCode` va fi setat la `EXCEPTION_ACCESS_VIOLATION` sau `EXCEPTION_DATATYPE_MISALIGNMENT`, iar `ExceptionAddress` la adresa instruciunii care a cauzat excepia; `NumberParameters` va fi setat pe 2, iar prima intrare în `ExceptionInformation` va fi 0 dacă s-a efectuat o operaie de citire sau 1 dacă s-a efectuat o operaie de scriere. A doua intrare din `ExceptionInformation` va conine adresa virtuală la

care s-a încercat accesarea fără drepturi, fapt care a dus la generarea excepiei. Aadar, corespondentul câmpului `si_addr` din structura `siginfo_t` de pe Linux este `ExceptionInformation[1]` pe Windows, NU `ExceptionAddress`.

Funcia de tratare a excepiei înregistrată cu `AddVectoredExceptionHandler` trebuie să întoarcă `EXCEPTION_CONTINUE_EXECUTION`, dacă excepia a fost tratată i se dorete continuarea execuiei, sau `EXCEPTION_CONTINUE_SEARCH` pentru a continua parcurgerea listei de funcii de tratare a excepiilor, în caz că au fost înregistrate mai multe astfel de funcii.

Quiz

Pentru auto-evaluare răspundei la întrebările din acest quiz.

Exerciii

Prezentare

Pentru a urmări mai uor noiunile expuse la începutul laboratorului folosii această prezentare (pdf) (odp).

Exerciii de laborator

Folosii arhiva de sarcini a laboratorului.

1. **Linux (1p)** În directorul `efence` din arhivă se găsete un program care conine un bug. Folosii `efence` pentru detectarea bug-ului i corectai-l. Explicai de ce bug-ul nu s-a manifestat anterior.
    - **nu modificai *char mtext[0]* din structura *msgbuf*** - motivul folosirii acestui vector de dimensiune zero în structură.
    - **Hint**: nu uitai de `ipcrm(1)`.
2. **Linux (2p)/Windows (2p)**: Să se scrie un program care copiază un fiier. Programul primete ca argumente numele fiierului sursă, numele fiierului desinatie, mapează în memorie cele două fiiere i copiază coninutul primului fiier folosind `memcpy(3)`.
    - **Hint**: pentru aflarea lungimii unui fiier
        - ◊ în Linux: `stat(2)`
        - ◊ în Windows: `GetFileAttributesEx`
    - **Atenie**: fiierul destinaie trebuie trunchiat la dimensiunea fiierului sursă
        - ◊ în Linux: `ftruncate(2)`
        - ◊ în Windows: `SetFilePointer` i `SetEndOfFile`
    - **Atenție**: în Linux trebuie folosit `MAP_SHARED`, altfel nu se va scrie nimic în fiierul final.
3. **Linux (1p)/Windows (1p)**: Alocai cu `mmap`/`VirtualAlloc` memorie de dimensiune dublul mărimii unei pagini (folosii `NULL` ca adresă de start). Acordai drepturi de scriere pentru primul byte de la adresa intoarsă. Verificai dacă ceilali bytes pot fi scrii/citii.
4. **Linux (2p)/Windows (2p)**: Să se creeze trei zone de memorie în spaiul de adresă, cu drepturi de citire, scriere, respectiv nici un drept. Să se testeze comportamentul programului când se fac accese de citire i scriere în aceste zone.

- **Atenție**: în Windows se poate apela `VirtualProtect` doar pentru o zonă de memorie alocată cu `VirtualAlloc`.

Pentru acasa

(5.) Linux/Windows: Pentru exerciiul anterior adăugai un handler de tratare a excepiilor care să remapeze zonele cu protecie de citire i scriere la generarea excepiilor.

Soluii

- Soluii exerciii laborator 7

Link-uri

1. Wikipedia: Memory Management
2. Memory Management in Linux
3. Opengroup - mmap
4. MSDN: Managing Virtual Memory in Win32
5. MSDN: Managing Memory-Mapped Files in Win32
6. MSDN: Structured Exception Handling