Laborator 07 - Memoria virtuală

Materiale ajutătoare

- lab07-slides.pdf
- lab07-refcard.pdf

Nice to read

- TLPI - Chapter 49, Memory mappings
- TLPI - Chapter 50, Virtual memory operations

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 aplicaţiile folosesc în mod curent memoria virtuală, ele nu fac acest lucru în mod explicit. Există însă câteva cazuri în care aplicaţiile folosesc memoria virtuală în mod explicit.

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

- **Maparea fişierelor** în memorie este folosită în unele sisteme de operare pentru a implementa mecanisme de memorie partajată. De asemenea, acest mecanism face posibilă implementarea paginării la cerere şi a bibliotecilor partajate.
- **Maparea memoriei** în spaţiul de adresă este folositoare atunci când un proces doreşte să aloce o cantitate mare de memorie.
- **Maparea dispozitivelor** este folositoare atunci când un proces doreşte să folosească direct memoria unui dispozitiv cum ar fi placa video.

Linux

Funcţiile cu ajutorul cărora se pot face cereri explicite asupra memoriei virtuale sunt funcţiile din familia `mmap(2)`. Funcţiile 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 fişierelor

În urma mapării unui fişier în spaţiul de adresă al unui proces, accesul la acest fişier se poate face similar cu accesarea datelor dintr-un vector. Eficienţa 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 depăşesc dimensiunea efectivă a memoriei fizice). Mai mult, partea de `I/O` este realizată de către kernel, programatorul scriind cod care doar preia/stochează valori în regiunea mapată. Astfel nu se mai apelează `read`, `write`, `lseek` - ceea ce adesea simplifică scrierea codului.

Observaţie

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

mmap

Prototipul funcţiei mmap ce permite maparea unui fişier în spaţiul de adresă al unui proces este următorul:

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

Funcţia va întoarce în caz de eroare `MAP_FAILED`. Dacă maparea s-a făcut cu succes, va întoarce un pointer spre o zonă de

memorie din spaţiul de adresă al procesului, zonă în care a fost mapat fişierul 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 preferinţe în ceea ce priveşte 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 fişierul la adresa cerută, atunci îl va mapa la altă adresă. Adresa la care se mapează fişierul este întoarsă de funcţie.

Parametrul `prot` specifică tipul de acces care se doreşte; 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 doreşte; poate lua următoarele valori (combinate prin SAU pe biţi; trebuie să existe cel puţin una: fie `MAP_PRIVATE`, fie `MAP_SHARED`):

- `MAP_PRIVATE` - se foloseşte o politică de tip copy-on-write; zona va conţine iniţial o copie a fişierului, dar scrierile nu sunt făcute în fişier; modificările nu vor fi vizibile în alte procese dacă există mai multe procese care au făcut `mmap` pe aceeaşi zonă din acelaşi fişier
- `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 şi 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 eşua
- `MAP_LOCKED` - se va bloca paginarea pe această zonă
- `MAP_ANONYMOUS` - se mapează memorie (argumentele `fd` şi `offset` sunt ignorate)

Este de remarcat că folosirea `MAP_SHARED` permite partajarea memoriei între procese care nu sunt înrudite. În acest caz, conţinutul fişierului devine conţinutul iniţial al memoriei partajate, şi orice modificare făcută de procese în această zonă este copiată apoi în fişier, asigurând persistenţă prin sistemul de fişiere.

msync

Pentru a declanşa în mod explicit sincronizarea fişierului cu maparea din memorie este disponibilă funcţia msync:

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

unde `flags` poate fi:

- `MS_SYNC` - datele vor fi scrise în fişier şi abia apoi funcţia se va termina.
- `MS_ASYNC` - este iniţiată secvenţa de salvare, dar nu se aşteaptă terminarea ei.
- `MS_INVALIDATE` - se invalidează mapările zonei din alte procese, pentru a forţa recitirea paginii în toate celelalte procese la următorul acces.

Alocare de memorie în spaţiul de adresă al procesului

În UNIX, tradiţional, pentru alocarea *memoriei dinamice*, se foloseşte apelul de sistem brk. Acest apel creşte sau descreşte zona de heap asociată procesului. Odată cu oferirea către aplicaţii 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 spaţiul de adresă memorie, nu fişiere.

Procesele pot cere alocarea unei zone de memorie de la o anumită adresă din spaţiul de adresare, chiar şi cu o anumită politică de acces (citire, scriere sau execuţie). În UNIX, acest lucru se face tot prin intermediul funcţiei mmap. Pentru acest lucru parametrul de flag-uri trebuie să conţină flag-ul `MAP_ANONYMOUS`.

Maparea dispozitivelor

Există chiar şi posibilitatea ca aplicaţiile să mapeze în spaţiul de adresă al unui proces un dispozitiv de intrare-ieşire. Acest lucru este util de exemplu pentru plăcile video: o aplicaţie poate mapa în spaţiul de adresă memoria plăcii video. În UNIX, dispozitivele fiind reprezentate prin fişiere, pentru a realiza acest lucru nu trebuie decât să deschidem fişierul asociat dispozitivului şi să-l folosim într-un apel `mmap`. Atenţie 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 fişierul `/dev/zero` pentru a face mapări de memorie, ca şi când s-ar folosi flag-ul `MAP_ANONYMOUS`.

Demaparea unei zone din spaţiul de adresă

Dacă se doreşte demaparea unei zone din spaţiul de adresă al procesului se poate folosi funcţia 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 conţina bucăţi deja demapate. Se pot astfel demapa mai multe zone în acelaşi timp.

Redimensionarea unei zone mapate

Pentru a executa operaţii de redimensionare a zonei mapate se poate utiliza funcţia mremap:

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

Zona pe care `old_adress` şi `old_size` o descriu trebuie să aparţină unei singure mapări. O singură opţiune este disponibilă pentru `flags`: `MREMAP_MAYMOVE` care arată că este în regulă ca pentru obţinerea noii mapări să se realizeze o nouă mapare într-o altă zonă de memorie (vechea zona fiind dezalocată).

Schimbarea protecţiei 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 funcţia mprotect:

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

Funcţia primeşte ca parametri intervalul de adrese [`addr`, `addr + len - 1`] şi noile drepturi de access (`PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, `PROT_NONE`). Ca şi la munmap, `addr` trebuie să fie multiplu de *dimensiunea unei pagini*. Funcţia va schimba protecţia pentru toate paginile care conţin cel puţin un octet în intervalul specificat.

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) despre modul în care zona va fi folosită.

Blocarea paginării

Există o categorie de procese care trebuie să execute anumite acţiuni la momente de timp bine determinate, pentru a se păstra calitatea execuţiei. Pentru exemplificare, putem considera un player audio/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 dispoziţie apelurile mlock şi mlockall.

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

Există, bineînţeles, şi funcţii ce readuc lucrurile la normal:

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

Astfel, funcţia munlock va reporni mecanismul de paginare al tuturor paginilor din intervalul [`addr`, `addr + len - 1`], iar funcţia munlockall face acelaşi lucru pentru toate paginile procesului, atât curente cât şi 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.

Excepţii

Atunci când se detectează o încălcare a protecţiei 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 funcţii pe care aici o să le denumim `signal` şi sigaction. Funcţia de tip sigaction va primi ca parametru o structură `siginfo_t`. În cazul semnalelor ce tratează excepţii 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 spaţiul 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 excepţia

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. Soluţia folosită de Electric Fence este înlocuirea apelurilor standard `malloc` şi `free` cu implementări proprii. Electric Fence va plasa zona de memorie alocată în spaţiul de adrese al procesului, astfel încât ea să fie mărginită de pagini neaccesibile (protejate la scriere şi citire).

Din păcate, sistemul de operare şi arhitectura procesorului limitează dimensiunea paginii la cel puţin 1-4K, astfel încât dacă zona de memorie alocată nu este multiplu de această dimensiune, există posibilitatea ca programul să poată citi sau scrie şi în zone în care nu ar trebui, fără ca sistemul de operare să oprească executia programului. Pentru a preveni situaţii de acestă natură, Electric Fence alocă zonele de memorie la limita superioară a unei pagini, mapând o pagină neaccesibilă după aceasta. Această abordare nu previne *buffer underrun*-ul, în care datele sunt citite sau scrise peste limita inferioară.

Pentru a putea verifica şi astfel de situaţii utilizatorul trebuie să definească variabila de mediu `EF_PROTECT_BELOW` înainte de rula programul. În acest caz, Electric Fence 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 situaţiilor de *buffer overrun*? Aşa cum am explicat şi în secţiunea precedentă, astfel de situaţii vor produce în cele din urmă erori, dar la un moment de timp ulterior, când va fi mai greu să determine cauza erorii cu mijloace de depanare obişnuite. În plus, în situaţiile de *buffer overrun* se pot suprascrie nu numai variabile, ci şi alte date importante pentru stabilitatea programului cum ar fi datele de control folosite de rutinele `malloc` şi `free`. Biblioteca Electric Fence poate determina erorile de *buffer overrun* doar dacă acestea apar în memoria alocată dinamic (adică în zona *heap*) cu rutinele `malloc` şi `free`. Pentru a folosi Electric Fence utilizatorul trebuie să folosească la link-editarea programului biblioteca `libefence`. Pentru a vedea utilitatea acestui pachet, să analizăm programul de mai jos:

ef_example.c

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

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

        data_1 = malloc(11 * sizeof(int));

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

        data_2 = malloc(11 * sizeof(int));

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

        for (i = 0; i <= 11; i++)
                  <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("%d
%d\n", data_1[i], data_2[i]);

        free(data_1);
        free(data_2);
```

```
        return 0;
    }
```

Aparent totul pare în regulă. La execuţia programului însă obţinem următorul output:

```
 so@spook$ gcc -Wall -g ef_example.c
 so@spook$ ./a.out
ff: malloc.c:3074: sYSMALLOc: Assertion `(old_top == (((mbinptr) (((char *)
&((av)->bins[((1) - 1) * 2])) - __builtin_offsetof (struct malloc_chunk, fd))))
&& old_size == 0) || ((unsigned long)(old_size) >= (unsigned long)
((((__builtin_offsetof (struct malloc_chunk, fd_nextsize))+((2 * (sizeof(size_t)))
 - 1)) & ~((2 * (sizeof(size_t))) - 1))) && ((old_top)->size & 0x1) &&
((unsigned long)old_end & pagemask) == 0)' failed.
```

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

```
 so@spook$ gcc -Wall -g ef_example.c -lefence
 so@spook$ gdb ./a.out
 Reading symbols from /home/so/a.out...done.
 (gdb) run
 Starting program: /home/so/a.out
 [Thread debugging using libthread_db enabled]

   Electric Fence 2.1 Copyright (C) 1987-1998 Bruce Perens.

 Program received signal SIGSEGV, Segmentation fault.
 0x08048536 in main () at ef.c:12
 12                      data_1[i] = i;
 (gdb) print i
  class="code bash" = 11
 (gdb)
```

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

Pentru mai multe informaţii despre Electric Fence consultaţi pagina de manual (**man efence**).

Windows

În Windows funcţiile de control al memoriei virtuale sau mai bine zis al spaţiului de adresă al unui proces nu mai sunt grupate, ca în cazul Unix, într-o singură primitivă oferită de sistemul de operare. Avem funcţii pentru maparea fişierelor în memorie şi funcţii pentru alocarea de memorie fizică în spaţiul de adresă al unui proces.

Maparea fişierelor

Pentru a mapa un fişier în spaţiul de adresă al unui proces trebuie mai întâi creat un handle către un obiect de tipul FileMapping şi apoi realizată efectiv maparea. Funcţiile CreateFileMapping şi MapViewOfFile au mai fost prezentate atunci când s-a discutat despre memoria partajată.

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

Funcţia primeşte ca parametri handle-ul fişierului care se doreşte a fi mapat, atribute de securitate care controlează accesul la handle-ul obiectului `FileMapping` creat, tipul mapării (`PAGE_READONLY`, `PAGE_READWRITE`, `PAGE_WRITECOPY` pentru copy-on-write) şi dimensiunea maximă care poate fi mapată cu ajutorul funcţiei `MapViewOfFile`. Opţional se poate specifica şi un şir care să identifice obiectul `FileMapping` creat. Dacă mai există un obiect de acest tip, funcţia `CreateFileMapping` nu va crea unul nou, ci îl va folosi pe cel existent. Atenţie 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 funcţia OpenFileMapping

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

Funcţia primeşte 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 fişier de unde începe maparea şi numărul de octeţi de mapat. Funcţia va întoarce un pointer în spaţiul de adresă al procesului, la zona mapată.

Alocare de memorie în spaţiul de adresă al procesului

Pentru alocarea de memorie în spaţiul de adresă al procesului se pot folosi funcţiile 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 funcţia VirtualAllocEx se poate aloca memorie în spaţiul de adresă al unui proces arbitrar, specificat în parametrul `hProcess`. Procesul curent trebuie să aibă drepturi corespunzătoare asupra procesului pe care se încearcă operaţia (`PROCESS_VM_OPERATION`). Funcţiile întorc un pointer către adresa de start, iar parametrii aşteptaţi de funcţii sunt descrişi în spoiler:

Demaparea unei zone din spaţiul de adresă

Pentru demaparea unei fişier mapat în memorie se foloseşte funcţia UnmapViewOfFile:

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

Funcţia primeşte adresa de început a zonei.

Pentru demaparea unei zone de memorie din spaţiul de adresă se folosesc funcţiile VirtualFree şi VirtualFreeEx:

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

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

Parametrii `lpAddress` şi `dwSize` identifică zona de dezalocat. `dwFreeType` specifică tipul operaţiei: `MEM_DECOMMIT`, `MEM_RELEASE`. Prima operaţie va demapa paginile din spaţiul de adresă, dar ele vor rămâne rezervate. Cea de-a doua operaţie 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 0.

Schimbarea protecţiei unei zone mapate

În Windows, schimbarea drepturilor de access a unei zone mapate se poate face cu ajutorul funcţiilor VirtualProtect şi 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
);
```

Funcţiile vor schimba protecţia 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`.

Atenţie! - Toate paginile din intervalul specificat trebuie să fie din aceeaşi regiune rezervată cu apelul `VirtualAlloc` sau `VirtualAllocEx` folosind `MEM_RESERVE`. Paginile nu pot fi localizate în regiuni adiacente rezervate prin apeluri separate ale `VirtualAlloc` sau `VirtualAllocEx` folosind `MEM_RESERVE`.

Interogarea zonelor mapate

Pentru a afla informaţii despre o zonă mapată în spaţiul de adresă al unui proces se pot folosi funcţiile VirtualQuery şi VirtualQueryEx. Ele vor oferi informaţii apelantului despre adresa de start a zonei, protecţie, 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
);
```

Funcţiile primesc ca parametri o adresă din cadrul zonei ce se doreşte a fi interogată, un pointer către un buffer alocat ce va primi informaţii despre zonă şi întorc numărul de octeţi scrişi în buffer. Dacă funcţia întoarce 0 înseamnă că nici o informaţie nu a fost furnizată. Acest lucru se întâmplă dacă funcţiei îi este pasată o adresă din spaţiul kernel.

Informaţiile primite vor descrie două zone: zona alocată (cu `VirtualAlloc`) în care este inclusă adresa dată, şi zona care conţine pagini de acelaşi fel (cu aceeaşi protecţie şi 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` şi `AllocationProtect` se referă la zona alocată, iar `BaseAddress`, `RegionSize`, `Type` şi `Protect` la zona ce conţine pagini de acelaşi fel. `State` indică starea paginilor din zonă: `MEM_COMMIT` pentru zonă alocată, `MEM_RESERVED` pentru zonă rezervată şi `MEM_FREE` pentru zonă nealocată. `Type` indică dacă în zonă este mapat un fişier (`MEM_IMAGE` sau `MEM_MAPPED`) sau nu, şi 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 consecinţă apelurile ulterioare nu mai produc page fault), sistemul de operare Windows pune la dispoziţia utilizatorilor funcţia VirtualLock:

```
BOOL VirtualLock(
   LPVOID lpAddress,
   SIZE_T dwSize
);
```

Funcţia primeşte 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.

Funcţia pentru reactivarea paginării este VirtualUnlock:

```
BOOL VirtualUnlock(
   LPVOID lpAddress,
   SIZE_T dwSize
);
```

Excepţii

Atunci când sistemul de operare detectează accese incorecte la memorie, va genera o excepţie către procesul care a efectuat accesul. Pentru tratarea excepţiei se pot folosi construcţii `__try` şi `__except`, pentru care este necesar suport din partea compilatorului, sau se poate folosi funcţia AddVectoredExceptionHandler.

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

```
ULONG RemoveVectoredExceptionHandler(
   PVOID VectoredHandlerHandle
);
```

Funcţia AddVectoredExceptionHandler va adăuga pe lista funcţiilor de executat atunci când se generează o excepţie, pe cea primită ca parametru în `VectoredHandler`. Parametrul `FirstHandler` indică dacă funcţia doreşte să fie adăugată la începutul listei sau la sfârşit. Funcţia de tratare a excepţiilor 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 excepţii cauzate de un acces invalid la memorie, `ExceptionCode` va fi setat la `EXCEPTION_ACCESS_VIOLATION` sau `EXCEPTION_DATATYPE_MISALIGNMENT`, iar `ExceptionAddress` la adresa instrucţiunii care a cauzat excepţia; `NumberParameters` va fi setat pe 2, iar prima intrare în `ExceptionInformation` va fi `0` dacă s-a efectuat o operaţie de citire sau `1` dacă s-a efectuat o operaţie de scriere. A doua intrare din `ExceptionInformation` va conţine adresa virtuală la care s-a încercat accesarea fără drepturi, fapt care a dus la generarea excepţiei. Aşadar, corespondentul câmpului `si_addr` din structura `siginfo_t` de pe Linux este `ExceptionInformation[1]` pe Windows, NU `ExceptionAddress`.

Funcţia de tratare a excepţiei înregistrată cu AddVectoredExceptionHandler trebuie să întoarcă `EXCEPTION_CONTINUE_EXECUTION`, dacă excepţia a fost tratată şi se doreşte continuarea execuţiei, sau `EXCEPTION_CONTINUE_SEARCH` pentru a continua parcurgerea listei de funcţii de tratare a excepţiilor, în caz că au fost înregistrate mai multe astfel de funcţii.

Exerciţii

În rezolvarea laboratorului, folosiţi arhiva de sarcini lab07-tasks.zip

Observaţii: Pentru a vă ajuta la implementarea exerciţiilor din laborator, în directorul `utils` din arhivă există un fişier `utils.h` cu funcţii utile.

Platforma este la alegerea voastră. Punctajul maxim se poate obţine fie pe Linux, fie pe Windows

Linux

1. (**0.5 punct**) Investigarea mapării folosind pmap
   - Intraţi în directorul `1-intro` şi compilaţi sursa `intro.c`
   - Rulaţi programul `intro`. Folosiţi ENTER pentru a continua programul.
   - Folosiţi comanda watch -d pmap $(pidof intro) pentru a urmări modificările asupra memoriei procesului.
   - Urmăriţi modificările care apar în urma diferitelor tipuri de mapare din cod.
   - De ce unele biblioteci sunt mapate cu drept de scriere ?
   - Analizaţi mapările făcute de procesul `init` folosind comanda: sudo pmap 1
2. (**1 punct**) Scrierea în fişier - write vs mmap
   - Intraţi în directorul `2-compare` şi inspectaţi sursele `write.c` şi `mmap.c`, apoi compilaţi
   - Obţineţi timpul de execuţie al celor două programe folosind comanda `time`: time ./write; time ./mmap

- Care timp este mai mare şi de ce?
- Hint:
  - folosiţi strace pentru a vedea câte apeluri de sistem se realizează pentru rularea fiecărui program
- Ce se întâmplă dacă în programul `mmap.c` schimbaţi flagul de creare al memoriei partajate din MAP_SHARED în MAP_PRIVATE? Cum explicaţi?
- Hint:
  - man mmap, în special secţiunea despre MAP_PRIVATE
  - Ce conţine fişierul 'test_mmap' în ambele cazuri?

3. (**1 punct**) Detectare buffer underrun utilizând ElectricFence.
   - Intraţi în directorul `3-efence` şi urmăriţi sursa `bug.c`
   - Compilaţi şi rulaţi executabilul `bug`
   - Folosiţi ElectricFence pentru a prinde situaţia de 'buffer underrun'
   - Hint:
     - Creaţi şi rulaţi programul `ef_bug` utilizând makefile-ul `Makefile_efence`
     - Setaţi în bash export EF_PROTECT_BELOW=1
   - Explicaţi de ce bug-ul nu s-a manifestat anterior.
   - Hint: Urmăriţi exemplul din secţiunea ElectricFence.
4. (**2 puncte**) Copierea fişierelor folosind mmap
   - Intraţi în directorul `4-cp`.
   - Completaţi sursa `mycp.c` astfel încât să realizeze copierea unui fişier primit ca argument
   - Pentru aceasta, mapaţi ambele fişiere în memorie
   - Realizaţi copierea ca o simplă copiere de vectori
   - Hints:
     - Înainte de mapare, aflaţi dimensiunea fişierului sursă folosind fstat
     - Nu uitaţi să trunchiaţi fişierul destinaţie la dimensiunea fişierului sursă
     - Urmăriţi comentariile cu `TODO`
     - Revedeţi secţiunea maparea fişierelor
   - Puteţi testa în felul următor: ./mycp Makefile /tmp/Makefile; diff Makefile /tmp/Makefile
   - Verificaţi cum realizează utilitarul cp copierea de fişiere (folosind mmap sau read/write).
     - Hint: folosiţi strace
   - De ce credeţi că se foloseşte această variantă ?
5. (**4 puncte**) Tipuri de acces pentru pagini.
   1. (**2 puncte**)
      - Intraţi în directorul `5-prot` şi inspectaţi sursa `prot.c`
      - Să se creeze trei zone de memorie în spaţiul de adresă, cu drepturi de citire, scriere, respectiv nici un drept
      - Zonele vor avea dimensiunea de o pagină
      - Să se testeze comportamentul programului când se fac accese de citire şi scriere în aceste zone.
        - Hint: Urmăriţi comentariile cu `TODO 1`
   2. (**2 puncte**)
      - Adăugaţi un handler de tratare a excepţiilor care să remapeze zonele cu protecţie de citire şi scriere la generarea excepţiilor.
      - Hint: Urmăriţi comentariile cu `TODO 2`
6. (**0.5 puncte**) Page-Faulturi.
   - Intraţi în directorul `6-faults` şi urmăriţi conţinutul fişierului `fork-faults.c`.
   - **Câte page-fault-uri** credeţi că se realizează la rulare?
   - Compilaţi fişierul
   - Folosiţi utilitarul `pidstat` din pachetul `sysstat` care permite monitorizarea page fault-urilor unui proces (prin intermediul argumentului `-r`).
   - Rulaţi programul `fork-faults`. Folosiţi `ENTER` pentru a continua programul.
   - Folosiţi comanda pidstat -r -T ALL -p $(pidof fork-faults) pentru a urmări page fault-urile. Rulaţi comanda pentru fiecare secvenţă de program.
   - Urmăriţi evoluţia numărului de page fault-uri pentru cele două procese: părinte şi copil. Page fault-urile care apar în cazul unui copy-on-write în procesul copil vor fi vizibile ulterior şi în procesul părinte.
7. (**1 punct**) Blocarea paginării.
   - Vă aflaţi într-o situaţie în care trebuie să procesaţi în timp real datele dintr-un buffer şi vreţi să evitaţi swaparea paginilor.
     - Intraţi în directorul `7-paging` şi completaţi `TODO-urile` astfel încât paginarea va fi blocată pentru variabila *data* pe parcursul lucrului cu aceasta, iar la final va fi deblocată.
     - Hints:
       - Adresa trebuie aliniata la limita unei pagini
       - Revedeţi secţiunea referitoare la blocarea paginării
     - Cât de mare poate fi *DATA_SIZE*? Încercaţi cu diverse valori şi explicaţi comportamentul.
     - Memoria blocata este prin definiţie memorie rezidentă - nu poate fi trimisă pe swap. Puteţi urmări cum se modifică dimensiunea memoriei rezidente (cea care nu poate fi trimisa pe swap) folosind comanda: ps -p $(pidof paging) -o pid,rss,vsz,comm după fiecare pas al programului.

Bonus Linux

1. (**1 so karma**) Schimbarea tipului de acces pentru pagini din segmentul de cod.
   - Intraţi în directorul `8-hack`.
   - Programul apelează funcţia *foo()*. Având determinată pagina în care se află funcţia în spaţiul de adresă al procesului, i se schimbă drepturile de acces în `PROT_READ|PROT_WRITE|PROT_EXEC` şi se modifică valoarea de retur a funcţiei (se scrie în segmentul de cod). Analizaţi cu atenţie programul.
   - Analizaţi comportamentul cu `gdb`. Având *pid-ul* procesului afişat la `stdout`, folosiţi pmap pentru a observa pagina cu drepturile schimbate.
   - Observaţi tipul de acces pentru celelalte pagini din spaţiul de adresă al procesului.
   - Modificaţi drepturile de acces în `PROT_READ|PROT_EXEC` şi recompilati sursa. Ce se întâmplă şi de ce?

Windows

1. (**0.5 puncte**) Maparea memoriei
   - Deschideţi proiectul `1-intro`
   - Inspectaţi şi compilaţi sursa `intro.c`
   - Rulaţi proiectul, iar în paralel urmăriţi comportamentul programului `intro` în Task Manager - în special coloanele `Mem Usage` şi `Page Faults`
   - Hints:
     - Pentru a vedea o listă completă cu coloanele care pot fi activate, Task ManagerViewSelect Columns
2. (**1 punct**) Crearea unor rutine în mod dinamic.
   - Deschideţi proiectul `2-dyn` şi urmăriţi sursa `dyn.c`
   - Programul alocă memorie în spaţiul de adresă al procesului pt a stoca o rutină, de forma `dyncode`.
     - Rutina va incrementa parametrul primit şi va întoarce această valoare. Urmăriţi conţintul lui `code`.
     - Deşi în acest caz conţinutul rutinei este definit direct în program prin `code`, el ar putea fi primit în orice alt mod (fişier, etc).
3. (**2 puncte**) Mapare fişiere în memorie.
   - Să se scrie un program care copiază un fişier. Programul primeşte ca argumente numele fişierului sursă, numele fişierului desţinatie, mapează în memorie cele două fişiere şi copiază conţinutul primului fişier folosind `memcpy(3)`.
     - Folosiţi proiectul `3-copy`.
     - **Hint**: pentru aflarea lungimii unui fişier folosiţi GetFileAttributesEx.
     - **Atenţie**: fişierul destinaţie trebuie trunchiat la dimensiunea fişierului sursă. Folosiţi SetFilePointer şi SetEndOfFile.
4. (**4 puncte**) Tipuri de acces pentru pagini.
   - Încărcaţi proiectul `4-prot` şi inspectaţi sursa `libvm.c`
   1. (**2 puncte**)
      - Să se creeze trei zone de memorie în spaţiul de adresă, cu drepturi de citire, scriere, respectiv nici un drept
      - Zonele vor avea dimensiunea de o pagină
      - Să se testeze comportamentul programului când se fac accese de citire şi scriere în aceste zone.
      - Hint:
        - Urmăriţi comentariile cu `TODO 1`
        - În Windows se poate apela VirtualProtect doar pentru o zonă de memorie alocată cu VirtualAlloc.
   2. (**2 puncte**)
      - Adăugaţi un handler de tratare a excepţiilor care să remapeze zonele cu protecţie de citire şi scriere la generarea excepţiilor.
      - Hint:
        - Revedeţi secţiunea cu Schimbarea protecţiei unei zone mapate
        - Pentru implementarea handlerului, revedeţi secţiunea referitoare la Excepţii
        - Urmăriţi comentariile cu `TODO 2`
5. (**1 punct**) Detectare buffer overrun - implementare utilitar asemănător cu Electric Fence
   - Încarcaţi proiectul `5-ef`
   - Inspectaţi sursa, ignorând pentru moment funcţia `MyMalloc`
   - Compilaţi şi rulaţi proiectul. Ar trebui să apară erori ?
   - Completaţi funcţia `MyMalloc` astfel încât orice depăşire a bufferului alocat să producă eroare.
   - Hints:
     - Alocaţi cu VirtualAlloc memorie de dimensiunea primită ca parametru + încă o pagină la final (o vom numi `guard page`)
     - Schimbaţi dreptul de acces pentru pagina de final în PAGE_NOACCESS utilizând `VirtualProtect`
     - Întoarceţi un pointer la o zonă de memorie cu dimensiunea egală cu dimensiunea cerută, dar care se termină fix înainte de `guard page`)
     - Urmăriţi commentariile cu `TODO`

- Testaţi din nou folosind de data aceeasta `MyMalloc`, atât în cazul în care iniţializarea vectorului depăşeşte dimensiunea alocată, cât şi în cazul în care nu depăşeşte.

6. (**1.5 puncte**) Blocarea paginării.
   - Vă aflaţi într-o situaţie în care trebuie să procesaţi în timp real datele dintr-un buffer şi vreţi să evitaţi swaparea paginilor.
     - Intraţi în directorul `6-lock` şi completaţi `TODO-urile` astfel încât paginarea să fie blocată pentru variabila *data* pe parcursul lucrului cu aceasta, iar la final să fie deblocată.
     - Hints:
       - Adresa trebuie aliniată la limita unei pagini
       - Revedeţi secţiunea referitoare la blocarea paginării în Windows
     - Cât de mare poate fi *DATA_SIZE*? Încercaţi cu diverse valori şi explicaţi comportamentul.
     - Urmăriţi comportamentul programului în `Task Manager`

EXTRA

Comparaţi timpii de execuţie ai algoritmilor de numărare a liniilor din fişier, aflaţi în această arhivă

- Cât de performantă este metoda cu mapare a fişierului în memorie?
- Care sunt cele mai importante diferenţe între metoda mmap din modulul de Python cu acelaşi nume şi funcţia nativă din Linux?

Soluţii

Soluţii exerciţii laborator 7

Resurse Utile

- Wikipedia: Memory Management
- Memory Management in Linux
- Opengroup - mmap
- MSDN: Managing Virtual Memory in Win32
- MSDN: Managing Memory-Mapped Files in Win32
- MSDN: Structured Exception Handling
- Utilizarea vectorilor de excepţie (Windows)