Laborator 06 - Memoria virtuală

Materiale ajutătoare

- lab06-slides.pdf [http://elf.cs.pub.ro/so/res/laboratoare/lab06-slides.pdf]
- lab06-refcard.pdf [http://elf.cs.pub.ro/so/res/laboratoare/lab06-refcard.pdf]

Nice to read

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

Link-uri către secțiuni utile

Linux

- Maparea fișierelor
- Alocare de memorie în spațiul de adresă al procesului
- Maparea dispozitivelor
- Demaparea unei zone din spațiul de adresă
- Redimensionarea unei zone mapate
- Schimbarea protecției unei zone mapate
- Blocarea paginării
- ElectricFence

Windows

- Maparea fișierelor
- Alocare de memorie în spațiul de adresă al procesului
- Demaparea unei zone din spațiul de adresă
- Schimbarea protecției unei zone mapate
- Interogarea zonelor mapate
- Blocarea paginării

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

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

- **Maparea fișierelor** 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 și a bibliotecilor partajate.
- **Maparea memoriei** în spațiul de adresă este folositoare atunci când un proces dorește să aloce o cantitate mare de memorie.
- **Maparea dispozitivelor** este folositoare atunci când un proces dorește să folosească direct memoria unui dispozitiv (precum placa video).

Concepte teoretice

Dimensiunea spațiului de adresă virtual al unui proces depinde de dimensiunea registrelor procesorului. Astfel, pe un sistem de 32 biți un proces va putea accesa 2^32 = 4GB spațiu de memorie (pe de altă parte, pe un sistem de 64 biți va accesa teoretic 2^64 B). Spațiul de memorie al procesului este împărțit în spațiu rezervat pentru adresele virtuale de kernel - acest spațiu este comun tuturor proceselor - și spațiul virtual (propriu) de adrese al procesului. De cele mai multe ori, împărțirea între cele două este de 3/1 (3GB user space vs 1GB kernel space).

Memoria fizică (RAM) este împărțită între procesele active în momentul respectiv și sistemul de operare. Astfel că, în funcție de câtă memorie avem pe mașina fizică, este posibil să epuizăm toate resursele și să nu mai putem porni un proces nou. Pentru a evita acest scenariu s-a introdus mecanismul de memorie virtuală. În felul acesta, chiar dacă spațiul virtual (compus din segmentul de text, data, heap, stivă) al unui proces este mai mare decât memoria fizică disponibilă pe sistem, procesul va putea rula încărcându-și în memorie doar paginile de care are nevoie în timpul execuției (on demand paging).

Spațiul virtual de adrese este împărțit în *pagini virtuale* (page). Corespondentul pentru memoria fizică este *pagina fizică* (frame). Dimensiunea unei pagini virtuale este egală cu cea a unei pagini fizice. Dimensiunea este dată de hardware (în majoritatea cazurilor o pagină are 4KB pe un sistem de 32 biți sau 64 biți).

Atât timp cât un proces în timpul rulării accesează numai pagini rezidente în memorie, se execută ca și când ar avea tot spațiul mapat în memoria fizică. În momentul în care un proces va dori să acceseze o anumită pagină virtuală, care nu este mapată în memorie, se va genera un *page fault*, iar în urma acestui page fault pagina virtuală va fi mapată la o pagină fizică. Două procese diferite au spațiu virtual diferit, însă anumite pagini virtuale din aceste procese se pot mapa la aceeași pagină fizică. Astfel că, două procese diferite pot partaja o aceeași pagină fizică, dar nu partajează pagini virtuale.

malloc

Așa cum am aflat la laboratorul de gestiunea memoriei [http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-05#alocareadealocarea_memoriei], `malloc` alocă memorie pe heap, deci în spațiul virtual al procesului. Funcția `malloc` poate fi implementată fie folosind apeluri de sistem `brk`, fie apeluri `mmap` (mai multe detalii găsiți aici [http://ocw.cs.pub.ro/courses/so/cursuri/curs-06#alocarea_de_memorie_virtuala]). Despre funcția `mmap` vom vorbi în următoarele paragrafe din laboratorul curent.

Alocarea memoriei virtuale se face la nivel de pagină, astfel că `malloc` va aloca de fapt cel mai mic număr de pagini virtuale ce cuprinde spațiul de memorie cerut. Fie următorul cod:

```
char *p = malloc(4150);
DIE(p == NULL, "malloc failed");
```

Considerând că o pagină virtuală are 4KB = 4096 octeți, atunci apelul `malloc` va aloca 4096 octeți + 54 octeți = 4KB + 54 octeți, spațiu care nu este cuprins într-o singură pagină virtuală, astfel că se vor aloca 2 pagini virtuale. În momentul alocării cu `malloc` nu se vor aloca (tot timpul) și pagini fizice; acestea vor fi alocate doar atunci când sunt accesate datele din zona de memorie alocată cu `malloc`. De exemplu, în momentul accesării unui element din p se va genera un *page fault*, iar pagina virtuală ce cuprinde acel element va fi mapată la o pagină fizică.

În general, la apelul malloc de dimensiuni mici (când se apelează în spate apelul de sistem `brk`) biblioteca standard C parcurge paginile alocate, se generează page fault-uri, iar la revenirea din apel paginile fizice

vor fi deja alocate. Putem spune că pentru dimensiuni mici, apelul `malloc`, așa cum este văzut el din aplicație (din afara bibliotecii standard C), alocă și pagini fizice și pagini virtuale.

Mai mult, alocarea efectivă de pagini virtuale și fizice are loc în momentul apelului de sistem `brk`. Acesta prealocă un spațiu mai mare, iar viitoarele apeluri `malloc` vor folosi acest spațiu. În acest fel, următoarele apeluri `malloc` vor fi eficiente: nu vor face apel de sistem, nu vor face alocare efectivă de spațiu virtual sau fizic, nu vor genera *page fault-uri*.

Apelul `malloc` este mai eficient decât apelul `calloc` pentru că nu parcurge spațiul alocat pentru a-l umple cu zero-uri. Acest lucru înseamnă că `malloc` va întoarce zona alocată cu informațiile de acolo; în anumite situații, acest lucru poate fi un risc de securitate - dacă datele de acolo sunt private.

Linux

Funcțiile cu ajutorul cărora se pot face cereri explicite asupra memoriei virtuale sunt funcțiile din familia `mmap(2)`. Funcțiile folosesc ca unitate minimă de alocare *pagina*.

Se poate aloca numai un număr întreg de pagini, iar adresele trebuie să fie aliniate corespunzător.

Maparea fișierelor

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

Nu orice descriptor de fișier poate fi mapat în memorie. Socket-urile, pipe-urile, dispozitivele care nu permit decât accesul secvențial (ex. char device) sunt incompatibile cu conceptele de mapare. Există cazuri în care fișiere obișnuite nu pot fi mapate (spre exemplu, dacă nu au fost deschise pentru a putea fi citite; pentru mai multe informații: **man mmap**).

mmap

Prototipul funcției mmap [http://linux.die.net/man/2/mmap] ce permite maparea unui fișier în spațiul de adresă al unui proces este următorul:

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

Funcția 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 spațiul de adresă al procesului, zonă în care a fost mapat fișierul 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 preferințe în ceea ce privește 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 fișierul la adresa cerută, atunci îl va mapa la o adresă apropiată și multiplu de dimensiunea unei pagini. Cea mai corespunzatoare adresa este si intoarsa.

Parametrul `prot` specifică tipul de acces care se dorește:

- `PROT_READ` (citire)
- `PROT_WRITE` (scriere)
- `PROT_EXEC` (execuție)

- PROT_NONE.

Cand zona e folosită altfel decât s-a declarat, este generat un semnal SIGSEGV.

Parametrul flags permite stabilirea tipului de mapare ce se dorește; poate lua următoarele valori (combinate prin SAU pe biți; trebuie să existe cel puțin MAP_PRIVATE sau MAP_SHARED):

- MAP_PRIVATE - Se folosește o politică de tip *copy-on-write*. Zona va conține inițial o copie a fișierului, dar scrierile nu sunt făcute în fișier. Modificările *nu vor fi vizibile în alte procese*, dacă există mai multe procese care au făcut mmap pe aceeași zonă din același fișier.
- MAP_SHARED - Scrierile sunt actualizate imediat în toate mapările existente. In acest fel toate procesele care au realizat mapări vor vedea modificările. Lucru datorat faptului că mapările MAP_SHARED se fac peste paginile fizice din page cache, iar apelurile r/w folosesc paginile fizice din page cache pentru a reduce numărul de citiri/scrieri de pe disc. Actualizările pe disc vor avea loc la un moment de timp ulterior, nespecificat.
- MAP_FIXED - Dacă nu se poate face alocarea la adresa specificată de start, apelul va eșua.
- MAP_LOCKED - Se va bloca paginarea pe această zonă, în maniera mlock [http://linux.die.net/man/2/mlock].
- MAP_ANONYMOUS - Se mapează memorie RAM (argumentele fd și offset sunt ignorate).

Este de remarcat faptul că folosirea MAP_SHARED permite partajarea memoriei între procese care nu sunt înrudite. În acest caz, conținutul fișierului devine conținutul inițial al memoriei partajate și orice modificare făcută de procese în această zonă este copiată apoi în fișier, asigurând persistență prin sistemul de fișiere.

msync

Pentru a declanșa în mod explicit sincronizarea fișierului cu maparea din memorie este disponibilă funcția msync [http://linux.die.net/man/2/msync]:

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

unde flags poate fi:

- MS_SYNC - Datele vor fi scrise în fișier și se așteaptă până se termină.
- MS_ASYNC - Este inițiată secvența de salvare, dar nu se așteaptă terminarea ei.
- MS_INVALIDATE - Se invalidează mapările zonei din alte procese, astfel incât procesele își vor face update cu datele noi înscrise.

Apelul msync este util pentru a face scrierea paginilor modificate din page cache pe disc, cu scopul de a evita pierderea modificărilor în cazul unei căderi a sistemului.

Alocare de memorie în spațiul de adresă al procesului

În UNIX, tradițional, pentru alocarea *memoriei dinamice*, se folosește apelul de sistem brk [http://linux.die.net/man/2/brk]. Acest apel crește sau descrește zona de heap asociată procesului. Odată cu oferirea către aplicații a unor apeluri de sistem de gestiune a memoriei virtuale (mmap [http://linux.die.net/man/2/mmap]), a existat posibilitatea ca procesele să aloce memorie folosind aceste noi apeluri de sistem. Practic, procesele pot mapa memorie în spațiul de adresă, nu fișiere.

Procesele pot cere alocarea unei zone de memorie de la o anumită adresă din spațiul de adresare, chiar și cu o anumită politică de acces (citire, scriere sau execuție). În UNIX, acest lucru se face tot prin intermediul funcției mmap [http://linux.die.net/man/2/mmap]. Pentru acest lucru parametrul flags trebuie să conțină flag-ul MAP_ANONYMOUS.

Maparea dispozitivelor

Există chiar și posibilitatea ca aplicațiile să mapeze în spațiul de adresă al unui proces un dispozitiv de intrare-ieșire. Acest lucru este util, de exemplu, pentru plăcile video: o aplicație poate mapa în spațiul de adresă memoria fizica a plăcii video. În UNIX, dispozitivele fiind reprezentate prin fișiere, pentru a realiza acest lucru nu trebuie decât să deschidem fișierul asociat dispozitivului și să-l folosim într-un apel `mmap`.

Nu toate dispozitivele pot fi mapate în memorie, însă atunci când pot fi mapate, semnificația acestei mapări depinde strict de dispozitiv.

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

Demaparea unei zone din spațiul de adresă

Dacă se dorește demaparea unei zone din spațiul de adresă al procesului se poate folosi funcția munmap [http://linux.die.net/man/3/munmap]:

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

`start` reprezintă adresa primei pagini ce va fi demapată (trebuie să 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 conțină bucăți deja demapate. Se pot astfel demapa mai multe zone în același timp.

Redimensionarea unei zone mapate

Pentru a executa operații de redimensionare a zonei mapate se poate utiliza funcția mremap [http://linux.die.net/man/2/mremap]:

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

Zona pe care `old_address` și `old_size` o descriu trebuie să aparțină unei singure mapări. O singură opțiune este disponibilă pentru `flags`: `MREMAP_MAYMOVE` care arată că este în regulă ca pentru obținerea noii mapări să se realizeze o nouă mapare într-o altă zonă de memorie (vechea zona fiind dealocată).

Schimbarea protecției 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 funcția mprotect [http://linux.die.net/man/2/mprotect]:

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

Funcția primește ca parametri intervalul de adrese [`addr`, `addr` + `len` - 1] și noile drepturi de access (`PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, `PROT_NONE`). Ca și la munmap [http://linux.die.net/man/2/munmap], `addr` trebuie să fie multiplu de *dimensiunea unei pagini*. Funcția va schimba protecția pentru toate paginile care conțin cel puțin 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());
```

Apelul `getpagesize` va returna dimensiunea unei pagini in bytes.

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 [http://linux.die.net/man/2/madvise]) despre modul în care zona va fi folosită.

madvise [http://linux.die.net/man/2/madvise] e utilă mai ales atunci când în spatele memoriei virtuale se află un dispozitiv fizic (de ex., când se mapează fișiere de pe hard-disk, kernel-ul poate citi în avans pagini de pe disc, reducând latența datorată poziționării capului de citire). Prototipul funcției este următorul:

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

unde valorile acceptate pentru `advice` sunt:

- `MADV_NORMAL` - regiunea este una obișnuită și 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 secvențial; 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.

Blocarea paginării

Paginarea se referă la evacuarea paginilor pe disc (swap out) si restaurarea lor (swap in) atunci când sunt folosite. Există o categorie de procese care trebuie să execute anumite acțiuni la momente de timp bine determinate, pentru a se păstra calitatea execuției. 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 de pe disc. Pentru a contracara aceste probleme, sistemele UNIX pun la dispoziție apelurile mlock [http://linux.die.net/man/2/mlock] și mlockall [http://linux.die.net/man/2/mlockall].

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

Funcția mlock [http://linux.die.net/man/2/mlock] va bloca paginarea (nu se va mai face swap out) paginilor incluse în intervalul [`addr`, `addr` + `len` - 1]. Funcția mlockall [http://linux.die.net/man/2/mlockall] va bloca paginarea tuturor paginilor procesului, în funcție de flag-uri:

- `MCL_CURRENT` - se va bloca paginarea tuturor paginilor mapate în spațiul de adresă al procesului la momentul apelului
- `MCL_FUTURE` - se va bloca paginarea noilor pagini mapate în spațiul de adresă al procesului (noi mapări realizate cu funcția `mmap`, dar și 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 depășește limita de memorie impusă de sistem, va primi semnalul SIGSEGV. Pentru a nu se ajunge în astfel de situații, programul trebuie să folosească `mlockall(MCL_CURRENT | MCL_FUTURE)` și apoi să aloce dimensiunea maximă a stivei pe care urmează să o folosească (prin declararea unei variabile locale, un vector de exemplu, și accesarea completă a acesteia).

Există, bineînțeles, și funcții ce readuc lucrurile la normal:

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

Astfel, funcția munlock [http://linux.die.net/man/2/munlock] va reporni mecanismul de paginare al tuturor paginilor din intervalul [`addr`, `addr + len - 1`], iar funcția munlockall [http://linux.die.net/man/2/munlockall] face același lucru pentru toate paginile procesului, atât curente, cât și viitoare. Trebuie notat faptul că, dacă s-au efectuat mai multe apeluri mlock [http://linux.die.net/man/2/mlock] sau mlockall [http://linux.die.net/man/2/mlockall], este suficient un singur apel munlock [http://linux.die.net/man/2/munlock] sau munlockall [http://linux.die.net/man/2/munlockall] pentru a reactiva paginarea.

Excepții

Atunci când se detectează o încălcare a protecției 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 funcții: `sa_handler` și `sa_sigaction`. Funcția de tip `sa_sigaction` va primi ca parametru o structură `siginfo_t`. În cazul semnalelor ce tratează excepții 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 spațiul 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 excepția

ElectricFence

ElectricFence [http://linux.die.net/man/3/efence] 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**. Soluția folosită de Electric Fence [http://linux.die.net/man/3/efence] este înlocuirea apelurilor standard `malloc` și `free` cu implementări proprii. Electric Fence [http://linux.die.net/man/3/efence] va plasa zona de memorie alocată în spațiul de adrese al procesului, astfel încât ea să fie mărginită de pagini neaccesibile (protejate la scriere și citire).

Din păcate, sistemul de operare și arhitectura procesorului limitează dimensiunea paginii la cel puțin 1-4KB, astfel încât dacă zona de memorie alocată nu este multiplu de această dimensiune, există posibilitatea ca programul să poată citi sau scrie și în zone în care nu ar trebui, fără ca sistemul de operare să oprească executia programului. Pentru a preveni situații de acestă natură, Electric Fence [http://linux.die.net/man/3/efence] 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 sub limita inferioară.

Pentru a putea verifica și astfel de situații, utilizatorul trebuie să definescă variabila de mediu `EF_PROTECT_BELOW` înainte de rula programul. În acest caz, Electric Fence [http://linux.die.net/man/3/efence] 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 situațiilor de *buffer overrun*? Așa cum am explicat și în secțiunea precedentă, astfel de situații vor produce în cele din urmă erori, dar la momente de timp ulterioare, când va fi mai greu să se determine cauza erorilor cu mijloace de depanare obișnuite. În plus, în situațiile de *buffer overrun* se pot suprascrie nu numai variabile, ci și alte date importante pentru stabilitatea programului cum ar fi datele de control folosite de rutinele `malloc` și `free`. Biblioteca Electric Fence [http://linux.die.net/man/3/efence] poate determina erorile de *buffer overrun* doar dacă acestea apar în memoria alocată dinamic (adică în zona *heap*) cu rutinele `malloc` și `free`. Pentru a folosi Electric Fence [http://linux.die.net/man/3/efence] utilizatorul trebuie să folosească la link-editare 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++)
                printf("%d %d\n", data_1[i], data_2[i]);

        free(data_1);
        free(data_2);

        return 0;
}
```

Aparent totul pare în regulă. La execuția programului însă obținem 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 libefence și 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
$1 = 11
(gdb)
```

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

Pentru mai multe informații despre Electric Fence [http://linux.die.net/man/3/efence] consultați pagina de manual (**man efence**).

Windows

În Windows funcțiile de control al memoriei virtuale sau mai bine zis al spațiului de adresă al unui proces nu mai sunt grupate, ca în cazul Unix, într-o singură primitivă oferită de sistemul de operare. Avem funcții pentru maparea fișierelor în memorie și funcții pentru alocarea de memorie fizică în spațiul de adresă al unui proces.

Maparea fișierelor

Pentru a mapa un fișier în spațiul de adresă al unui proces trebuie mai întâi creat un handle către un obiect de tipul FileMapping [http://msdn.microsoft.com/en-us/library/aa366556%28VS.85%29.aspx] și apoi realizată efectiv maparea.

Pentru a crea un obiect de tip `FileMapping` se folosește funcția CreateFileMapping [http://msdn.microsoft.com/en-us/library/aa366537%28v=VS.85%29.aspx]:

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

Funcția primește ca parametri handle-ul fișierului care se dorește 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) și dimensiunea maximă care poate fi mapată cu ajutorul funcției `MapViewOfFile`. Opțional se poate specifica și un șir care să identifice obiectul `FileMapping` creat. Dacă mai există un obiect de acest tip, funcția `CreateFileMapping` nu va crea unul nou, ci îl va folosi pe cel existent. Atenție 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 funcția OpenFileMapping [http://msdn.microsoft.com/en-us/library/aa366791%28VS.85%29.aspx]:

```
HANDLE OpenFileMapping(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName
);
```

Maparea în spațiul de adrese al procesului se face folosind funcția MapViewOfFile [http://msdn.microsoft.com/en-us/library/aa366761%28VS.85%29.aspx]:

```
LPVOID MapViewOfFile(
   HANDLE hFileMappingObject,
   DWORD dwDesiredAccess,
   DWORD dwFileOffsetHigh,
   DWORD dwFileOffsetLow,
```

```
  SIZE_T dwNumberOfBytesToMap
);
```

Funcția primește 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 fișier de unde începe maparea și numărul de octeți de mapat. Funcția va întoarce un pointer în spațiul de adresă al procesului, la zona mapată.

Puteți urmări o prezentare mai detaliată a funcțiilor CreateFileMapping [http://msdn.microsoft.com/en-us/library/aa366537%28v=VS.85%29.aspx] și MapViewOfFile [http://msdn.microsoft.com/en-us/library/aa366761%28VS.85%29.aspx].

Alocare de memorie în spațiul de adresă al procesului

Pentru alocarea de memorie în spațiul de adresă al procesului se pot folosi funcțiile VirtualAlloc [http://msdn.microsoft.com/en-us/library/aa366887%28VS.85%29.aspx] sau VirtualAllocEx [http://msdn.microsoft.com/en-us/library/aa366890%28v=VS.85%29.aspx]:

```
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 funcția VirtualAllocEx [http://msdn.microsoft.com/en-us/library/aa366890%28v=VS.85%29.aspx] se poate aloca memorie în spațiul de adresă al unui proces arbitrar, specificat în parametrul `hProcess`. Procesul curent trebuie să aibă drepturi corespunzătoare asupra procesului pe care se încearcă operația (`PROCESS_VM_OPERATION`). Funcțiile întorc un pointer către adresa de start, iar parametrii așteptați de funcții sunt descriși în spoiler:

- `lpAddress` - adresa de unde începe alocarea; trebuie să fie multiplu de 4KB pentru alocare și 64KB pentru rezervare; dacă parametrul este `NULL`, sistemul va furniza automat o adresă
- `dwSize` - dimensiunea zonei
- `fAllocationType` - specifică tipul operației: rezervare (`MEM_RESERVE`), alocare (`MEM_COMMIT`) sau renunțare la zonă (`MEM_RESET`); rezervarea unei zone înseamnă de fapt "punerea deoparte" a unui interval din spațiul de adrese virtuale al procesului, fără a se aloca însă memorie fizică; dacă se folosește `MEM_COMMIT`, se alocă efectiv memorie (dar doar dacă în prealabil zona vizată a fost rezervată); atunci când se renunță la zonă nucleul poate face discard la paginile din zonă, fără a face însă dezalocarea lor; după această operație datele nu se păstrează
- `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 excepție `STATUS_GUARD_PAGE`. `PAGE_GUARD` și `PAGE_NOCACHE` se pot folosi împreună cu celelalte moduri.

Demaparea unei zone din spațiul de adresă

Pentru demaparea unei fișier mapat în memorie se folosește funcția UnmapViewOfFile [http://msdn.microsoft.com/en-us/library/aa366882%28VS.85%29.aspx]:

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

Funcția primește adresa de început a zonei.

Pentru dezalocarea unei zone de memorie din spațiul de adresă se folosesc funcțiile VirtualFree [http://msdn.microsoft.com/en-us/library/aa366892%28VS.85%29.aspx] și VirtualFreeEx [http://msdn.microsoft.com/en-us/library/aa366894%28v=VS.85%29.aspx]:

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

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

Funcția VirtualFreeEx [http://msdn.microsoft.com/en-us/library/aa366894%28v=VS.85%29.aspx] va dezaloca o zonă de memorie din spațiul de adresă al unui proces arbitrar, specificat în parametrul `hProcess`. Procesul curent trebuie să aibă drepturi corespunzătoare asupra procesului pe care se încearcă operația (`PROCESS_VM_OPERATION`).

Parametrii `lpAddress` și `dwSize` identifică zona de dezalocat. `dwFreeType` specifică tipul operației: `MEM_DECOMMIT`, `MEM_RELEASE`. Prima operație va demapa paginile din spațiul de adresă, dar ele vor rămâne rezervate. Cea de-a doua operație 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 protecției unei zone mapate

În Windows, schimbarea drepturilor de acces a unei zone mapate se poate face cu ajutorul funcțiilor VirtualProtect [http://msdn.microsoft.com/en-us/library/aa366898%28VS.85%29.aspx] și VirtualProtectEx [http://msdn.microsoft.com/en-us/library/aa366899%28v=VS.85%29.aspx]:

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

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

Funcțiile vor schimba protecția 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`.

Toate paginile din intervalul specificat trebuie să fie din aceeași 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 informații despre o zonă mapată în spațiul de adresă al unui proces se pot folosi funcțiile VirtualQuery [http://msdn.microsoft.com/en-us/library/aa366902%28VS.85%29.aspx] și VirtualQueryEx [http://msdn.microsoft.com/en-us/library/aa366907%28v=VS.85%29.aspx]. Ele vor oferi informații apelantului despre adresa de start a zonei, protecție, 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
);
```

Funcțiile primesc ca parametri o adresă din cadrul zonei ce se dorește a fi interogată, un pointer către un buffer alocat ce va primi informații despre zonă și întorc numărul de octeți scriși în buffer. Dacă funcția întoarce 0 înseamnă că nicio informație nu a fost furnizată. Acest lucru se întâmplă dacă funcției îi este pasată o adresă din spațiul kernel.

Informațiile primite vor descrie două zone: zona alocată (cu `VirtualAlloc`) în care este inclusă adresa dată, și zona care conține pagini de același fel (cu aceeași protecție și 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` și `AllocationProtect` se referă la zona alocată, iar `BaseAddress`, `RegionSize`, `Type` și `Protect` la zona ce conține pagini de același fel. `State` indică starea paginilor din zonă: `MEM_COMMIT` pentru zonă alocată, `MEM_RESERVED` pentru zonă rezervată și `MEM_FREE` pentru zonă nealocată. `Type` indică dacă în zonă este mapat un fișier (`MEM_IMAGE` sau `MEM_MAPPED`) sau nu, și indică de asemenea dacă zona este partajată (`MEM_PRIVATE`) sau nu.

Blocarea paginării

Pentru blocarea paginării pentru un set de pagini (nu se va mai face swap out - în consecință apelurile ulterioare nu mai produc page fault), sistemul de operare Windows pune la dispoziția utilizatorilor funcția VirtualLock [http://msdn.microsoft.com/en-us/library/aa366895%28VS.85%29.aspx]:

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

Funcția primește prin parametri un interval de pagini (alcătuit din paginile care au măcar un octet în intervalul [`lpAddress`, `lpAddress + dwSize - 1`]) pentru care se vrea blocarea paginării.

Funcția pentru reactivarea paginării este VirtualUnlock [http://msdn.microsoft.com/en-us/library/aa366910%28v=VS.85%29.aspx]:

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

Excepții

Atunci când sistemul de operare detectează accese incorecte la memorie, va genera o excepție către procesul care a efectuat accesul. Pentru tratarea excepției se pot folosi construcții `__try` și `__except`, pentru care este necesar suport din partea compilatorului, sau se poate folosi funcția AddVectoredExceptionHandler [http://msdn.microsoft.com/en-us/library/ms679274%28VS.85%29.aspx].

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

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

Funcția AddVectoredExceptionHandler [http://msdn.microsoft.com/en-us/library/ms679274%28VS.85%29.aspx] va adăuga pe lista funcțiilor de executat atunci când se generează o excepție, pe cea primită ca parametru în `VectoredHandler`. Parametrul `FirstHandler` indică dacă funcția dorește să fie adăugată la începutul listei sau la sfârșit. Funcția de tratare a excepțiilor trebuie să aibă următoarea semnătură:

```
LONG CALLBACK 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 excepții cauzate de un acces invalid la memorie, `ExceptionCode` va fi setat la `EXCEPTION_ACCESS_VIOLATION` sau `EXCEPTION_DATATYPE_MISALIGNMENT`, iar `ExceptionAddress` la adresa instrucțiunii care a cauzat excepția; `NumberParameters` va fi setat pe 2, iar prima intrare în `ExceptionInformation` va fi `0` dacă s-a efectuat o operație de citire sau `1` dacă s-a efectuat o operație de scriere. A doua intrare din `ExceptionInformation` va conține adresa virtuală la care s-a încercat accesarea fără drepturi, fapt care a dus la generarea excepției. Așadar, corespondentul câmpului `si_addr` din structura `siginfo_t` de pe Linux este `ExceptionInformation` pe Windows, NU `ExceptionAddress`.

Funcția de tratare a excepției înregistrată cu AddVectoredExceptionHandler [http://msdn.microsoft.com/en-us/library/ms679274%28VS.85%29.aspx] trebuie să întoarcă EXCEPTION_CONTINUE_EXECUTION, dacă excepția a fost tratată și se dorește continuarea execuției, sau EXCEPTION_CONTINUE_SEARCH pentru a continua parcurgerea listei de funcții de tratare a excepțiilor, în caz că au fost înregistrate mai multe astfel de funcții.

Exerciții

În rezolvarea laboratorului, folosiți arhiva de sarcini lab06-tasks.zip [http://elf.cs.pub.ro/so/res/laboratoare/lab06-tasks.zip]. Platforma este la alegerea voastră. Punctajul maxim se poate obține fie pe Linux, fie pe Windows. Lucrați în mașina virtuală

Pentru a vă ajuta la implementarea exercițiilor din laborator, în directorul `utils` din arhivă există un fișier `utils.h` cu funcții utile.

Linux

Exercițiul 1 - Investigarea mapărilor folosind pmap

Intrați în directorul `1-intro` și compilați sursa `intro.c`. Rulați programul `intro`:

```
./intro
```

Într-o altă consolă, folosiți comanda pmap [http://linux.die.net/man/1/pmap].:

```
 watch -d pmap $(pidof intro)
```

pentru a urmări modificările asupra memoriei procesului.

În prima consolă, folosiți ENTER pentru a continua programul. În cea de-a doua consolă urmăriți modificările care apar în urma diferitelor tipuri de mapare din cod.

Analizați mapările făcute de procesul `init` folosind comanda:

```
sudo pmap 1
```

Puteți observa că pentru bibliotecile partajate (de exemplu, `libc`) sunt mapate trei zone: zona de cod (read-execute), zona .rodata (read-only) și zona .data (read-write).

Exercițiul 2 - Scrierea în fișier - write vs. mmap

Intrați în directorul `2-compare` și inspectați sursele `write.c` și `mmap.c`, apoi compilați. Obțineți timpul de execuție al celor două programe folosind comanda `time`:

```
time ./write; time ./mmap
```

Observăm că varianta cu `mmap` este mai rapidă decât varianta cu `write`. Vom folosi strace [http://linux.die.net/man/1/strace] pentru a vedea ce apeluri de sistem se realizează pentru rularea fiecărui program:

```
strace -c ./write
strace -c ./mmap
```

Din output-ul `strace` observăm că programul `write` face foarte multe (100000) de apeluri `write` și din această cauză este mai lent decât programul `mmap`.

În continuare vom analiza cele două moduri de mapare a fișierelor: `MAP_SHARED` și `MAP_PRIVATE`. Observați că fișierul `test_mmap` (creat de programul `mmap` cu `MAP_SHARED`) conține 100000 de linii:

```
cat test_mmap | wc -l
```

În programul `mmap.c` schimbați flagul de creare al memoriei partajate din `MAP_SHARED` în `MAP_PRIVATE`, compilați și rulați din nou:

```
./mmap
cat test_mmap | wc -l
```

Modificările aduse unei zone de memorie mapată cu `MAP_PRIVATE` nu vor fi vizible nici altor procese și nici nu vor ajunge în fișierul mapat de pe disc.

Exercițiul 3 - Detectare 'buffer underrun' folosind ElectricFence

Intrați în directorul `3-efence` și urmăriți sursa `bug.c`. Compilați și rulați executabilul `bug`:

```
make
./bug
```

Folosiți ElectricFence pentru a prinde situația de 'buffer underrun' urmărind pașii:

- Instalați pachetul `electric-fence` în cazul in care biblioteca `libefence.so` nu se găsește pe sistem.
- Setați în bash variabila de mediu `EF_PROTECT_BELOW` la `1`:

  ```
  export EF_PROTECT_BELOW=1
  ```

- Creați și rulați programul `ef_bug` utilizând makefile-ul `Makefile_efence`:

  ```
  make -f Makefile_efence
  ./ef_bug
  ```

Dacă întâmpinați probleme în instalarea pachetului `electric-fence`, descărcați-l de aici pentru x86_64 [http://ro.archive.ubuntu.com/ubuntu/pool/main/e/electric-fence/electric-fence_2.2.4_amd64.deb] și de aici pentru i386 [http://ro.archive.ubuntu.com/ubuntu/pool/main/e/electric-fence/electric-fence_2.2.4_i386.deb] și instalați-l folosind comanda dpkg.

```
$ wget http://ro.archive.ubuntu.com/ubuntu/pool/main/e/electric-fence/electric-fence_2.2.4_i386.deb
$ sudo dpkg -i electric-fence_2.2.4_i386.deb
```

Exercițiul 4 - Copierea fișierelor folosind mmap

Intrați în directorul `4-cp` și completați sursa `mycp.c` astfel încât să realizeze copierea unui fișier primit ca argument. Pentru aceasta, mapați ambele fișiere în memorie și realizați copierea folosind `memcpy`. Urmăriți comentariile cu `TODO` din sursă și următoarele hint-uri:

- Înainte de mapare, aflați dimensiunea fișierului sursă folosind fstat [http://linux.die.net/man/2/fstat].

- Trunchiați fișierul destinație la dimensiunea fișierului sursă folosing ftruncate [https://linux.die.net/man/2/ftruncate].
- Folosiți MAP_SHARED pentru mapare pentru a fi transmise schimbările în fișier: rețineți faptul că apelul mmap folosește una dintre opțiunile MAP_SHARED sau MAP_PRIVATE (una singură)
- Pentru fișierul de intrare protecția trebuie să fie PROT_READ: fișierul a fost deschis read-only.
- Pentru fișierul de ieșire protecția trebuie să fie PROT_READ | PROT_WRITE; anumite arhitecturi/implementări se pot plânge dacă folosiți **doar** PROT_WRITE.
- Argumentele funcției memcpy [http://man7.org/linux/man-pages/man3/memcpy.3.html] sunt, în ordine: destinația, sursa, numărul de octeți care să fie copiați.
- Revedeți secțiunea maparea fișierelor.
- Asigurați persistența datelor pe sistemul de fișiere printr-un apel explicit msync [https://ocw.cs.pub.ro/courses/so/laboratoare/laborator-06#msync]

Puteți testa în felul următor:

```
./mycp Makefile /tmp/Makefile
diff Makefile /tmp/Makefile
```

Verificați cum realizează utilitarul cp [http://linux.die.net/man/1/cp] copierea de fișiere (folosind mmap sau read/write) folosind strace [http://linux.die.net/man/1/strace].

Utilitarul cp folosește read/write pentru a copia fișiere, în special pentru a limita consumul de memorie în cazul copierii unor fișiere de dimensiuni mari. De asemenea, în cazul mapării fișierului în memorie cu mmap, scrierea efectivă a datelor pe disc se va face într-un timp mai îndelungat, lucru care de cele mai multe ori nu este dorit (urmăriți acest link [http://stackoverflow.com/a/27987994]).

Exercițiul 5 - Tipuri de acces pentru pagini

Intrați în directorul 5-prot și inspectați sursa prot.c.

Creați o zonă de memorie în spațiul de adresă, formată din trei pagini virtuale (folosiți un singur apel mmap). Prima pagină nu va avea vreun drept, a doua va avea drepturi de citire, iar a treia va avea drepturi de scriere (folosiți mprotect pentru a configura drepturile fiecărei pagini). Testați comportamentul programului când se fac accese de citire și scriere în aceste zone. Completați comentariile cu TODO 1.

Adăugați un handler de tratare a excepțiilor care să remapeze incremental zonele cu protecție de citire și scriere la generarea excepțiilor. Astfel, dacă pagina nu are vreun drept, la page fault se va remapa cu drepturi de citire. Dacă pagina are drepturi de citire, la page fault se va remapa cu drepturi de citire + drepturi de scriere. Completați comentariile cu TODO 2.

Trebuie să ștergeți prima linie old_action.sa_sigaction(signum, info, context); pentru a putea rezolva a doua parte a exercițiului.

Exercițiul 6 - Page fault-uri

Intrați în directorul 6-faults și urmăriți conținutul fișierului fork-faults.c.

Vom folosi utilitarul pidstat (tutorial pidstat [http://www.cyberciti.biz/open-source/command-line-hacks/linux-monitor-process-using-pidstat]) din pachetul sysstat pentru a monitoriza page fault-urile făcute de un proces.

Dacă întâmpinați probleme în instalarea pachetului sysstat, descărcați-l de aici [http://ro.archive.ubuntu.com/ubuntu/pool/main/s/sysstat/sysstat_11.2.0-1_i386.deb] și instalați-l folosind comanda dpkg.

```
student@spook:~$ wget http://ro.archive.ubuntu.com/ubuntu/pool/main/s/sysstat/sysstat_11.2.0-1_i386.deb
student@spook:~$ sudo dpkg -i sysstat_11.2.0-1_i386.deb
```

Rulați programul `fork-faults`. Într-o altă consolă executați comanda

```
pidstat -r -T ALL -p $(pidof fork-faults) 5
```

pentru a urmări page fault-urile. Comanda de mai sus vă afișează câte un mesaj la fiecare 5 secunde; ne interesează valorile `minflt-nr`.

Pe rând, apăsați tasta ENTER în consola unde ați rulat programul `fork-faults` și observați output-ul comenzii `pidstat`. Urmăriți evoluția numărului de page fault-uri pentru cele două procese: părinte și copil. Page fault-urile care apar în cazul unui copy-on-write în procesul copil vor fi vizibile ulterior și în procesul părinte (după ce procesul copil își încheie execuția).

Pachetul `sysstat` mai conține și utilitarul `sar` prin care puteți colecta și realiza rapoarte despre activitatea sistemului. Pentru a activa salvarea datelor, trebuie setat flag-ul `ENABLED` din `/etc/default/sysstat`. Cu ajutorul utilitarului `sar` puteți monitoriza informații precum încărcarea CPU-ului, utilizarea memoriei și a paginilor, operațiile de I/O, activitatea proceselor. Detalii puteți afla din tutorial sar [http://www.cyberciti.biz/tips/identifying-linux-bottlenecks-sar-graphs-with-ksar.html].

Exercițiul 7 - Blocarea paginării

Vă aflați într-o situație în care trebuie să procesați în timp real datele dintr-un buffer și vreți să evitați swaparea paginilor. Intrați în directorul `7-paging` și completați `TODO-urile` astfel încât paginarea va fi blocată pentru variabila *data* pe parcursul lucrului cu aceasta, iar la final va fi deblocată. Deși pe Linux adresa va fi aliniată automat la dimensiunea unei pagini, acest lucru nu se întâmplă pe toate sistemele POSIX compliant, prin urmare este o practică bună să o aliniem manual.

Deoarece variabila *data* este o variabilă locală a funcției `main`, aceasta va fi alocată pe stivă. Rulați programul `paging` și folosiți, într-o altă consolă, comanda

```
pmap -X -p $(pidof paging)
```

după fiecare apăsare a tastei ENTER. Veți observa blocarea/deblocarea paginării pentru paginile mapate pe stivă ce conțin cel puțin un byte al variabilei `data`.

Limita maximă pentru care se poate executa cu succes `mlock` este dată de `RLIMIT_MEMLOCK` (max locked memory). Aceasta are de obicei valoarea 64KB și poate fi configurată folosind `ulimit`.

Bonus Linux

Schimbarea tipului de acces pentru pagini din segmentul de cod

Intrați în directorul `8-hack`. Programul apelează funcția *foo()*. Având determinată pagina în care se află funcția în spațiul de adresă al procesului, i se schimbă drepturile de acces în `PROT_READ|PROT_WRITE|PROT_EXEC` și se modifică valoarea de retur a funcției (se scrie în segmentul de cod).

Analizați cu atenție programul. Analizați comportamentul cu `gdb`. Având *pid-ul* procesului afișat la `stdout`, folosiți pmap [http://linux.die.net/man/1/pmap] pentru a observa pagina cu drepturile schimbate. Observați tipul de acces pentru celelalte pagini din spațiul de adresă al procesului.

Modificați drepturile de acces în `PROT_READ|PROT_EXEC`, compilați și rulați din nou. Observați că fără drepturi de scriere execuția programului este încheiată de un semnal SIGSEGV.

Windows

Exercițiul 1 - Maparea memoriei

Deschideți proiectul `1-intro`. Inspectați și compilați sursa `intro.c`. Rulați proiectul, iar în paralel urmăriți comportamentul programului `intro` în Task Manager - în special coloanele `Memory - Working Set`, `Memory - Private Working Set` și `Page Faults`. Pentru a vedea o listă completă cu coloanele care pot fi activate accesați Task Manager(tabul Processes)→View→Select Columns.

Exercițiul 2 - Crearea unor rutine în mod dinamic

Deschideți proiectul `2-dyn` și urmăriți sursa `dyn.c`. Programul alocă memorie în spațiul de adresă al procesului pentru a stoca o rutină, de forma `dyncode`. Rutina va incrementa parametrul primit și va întoarce această valoare. Urmăriți conținutul lui `code`. Deși în acest caz conținutul rutinei este definit direct în program prin `code`, el ar putea fi primit în orice alt mod (fișier, rețea).

Exercițiul 3 - Mapare fișiere în memorie

Să se scrie un program care copiază un fișier folosind proiectul `3-copy`. Programul primește ca argumente numele fișierului sursă și numele fișierului destinație, mapează în memorie cele două fișiere și copiază conținutul primului fișier folosind `memcpy(3)`. Pentru aflarea lungimii fișierului sursă s-a folosit GetFileAttributesEx [http://msdn.microsoft.com/en-us/library/aa364946(VS.85).aspx]. Fișierul destinație trebuie trunchiat la dimensiunea fișierului sursă folosind SetFilePointer [http://msdn.microsoft.com/en-us/library/aa365541(VS.85).aspx] și SetEndOfFile [http://msdn.microsoft.com/en-us/library/aa365531(VS.85).aspx].

Exercițiul 4 - Tipuri de acces pentru pagini

Încărcați proiectul `4-prot` și inspectați sursa `libvm.c`.

Să se creeze o zonă de memorie în spațiul de adresă, formată din trei pagini virtuale (folosiți un singur apel `VirtualAlloc`). Prima pagina nu va avea vreun drept, a două va avea drepturi de citire, iar a treia va avea drepturi de scriere (folosiți `VirtualProtect` pentru a configura drepturile fiecărei pagini). Să se testeze comportamentul programului când se fac accese de citire și scriere în aceste zone. Urmăriți comentariile cu `TODO 1`.

Adăugați un handler de tratare a excepțiilor care să remapeze incremental zonele cu protecție de citire și scriere la generarea excepțiilor. Astfel, dacă pagina nu are vreun drept, la page fault se va remapa cu drepturi de citire. Dacă pagina are drepturi de citire, la page fault se va remapa cu drepturi de citire + drepturi de scriere. Urmăriți comentariile cu `TODO 2`.

Exercițiul 5 - Detectare 'buffer overrun' - implementare utilitar asemănător cu Electric Fence

Încărcați proiectul `5-ef` și inspectați sursa, ignorând pentru moment funcția `MyMalloc`. Compilați și rulați proiectul.

Completați funcția `MyMalloc` astfel încât orice depășire a bufferului alocat să producă eroare (urmăriți comentariile cu `TODO`). Alocați cu VirtualAlloc [http://msdn.microsoft.com/en-us/library/aa366887%28VS.85%29.aspx] memorie de dimensiunea primită ca parametru + încă o pagină la final (o vom numi `guard page`). Schimbați dreptul de acces pentru pagina de final în PAGE_NOACCESS

utilizând `VirtualProtect` [http://msdn.microsoft.com/en-us/library/aa366898%28v=VS.85%29.aspx]. Întoarceți un pointer la o zonă de memorie cu dimensiunea egală cu dimensiunea cerută, dar care se termină fix înainte de `guard page`).

Testați din nou folosind de data aceasta `MyMalloc`, atât în cazul în care inițializarea vectorului depășește dimensiunea alocată, cât și în cazul în care nu depășește.

Exercițiul 6 - Blocarea paginării

Vă aflați într-o situație în care trebuie să procesați în timp real datele dintr-un buffer și vreți să evitați swaparea paginilor. Intrați în directorul `6-lock` și completați `TODO-urile` astfel încât paginarea să fie blocată pentru variabila *data* pe parcursul lucrului cu aceasta, iar la final să fie deblocată. Adresa trebuie aliniată la limita unei pagini.

Extra

Comparați timpii de execuție ai algoritmilor de numărare a liniilor dintr-un fișier, aflați în această arhivă [http://elf.cs.pub.ro/so/res/laboratoare/lab06-extra.zip]

- Cât de performantă este metoda cu mapare a fișierului în memorie în raport cu celelalte metode?
- Care sunt cele mai importante diferențe între metoda mmap [https://docs.python.org/2/library/mmap.html] din modulul de Python cu același nume și funcția nativă [http://man7.org/linux/man-pages/man2/mmap.2.html] din Linux?

Soluții

Soluții exerciții laborator 6 [http://elf.cs.pub.ro/so/res/laboratoare/lab06-sol.zip]

Resurse Utile

- Wikipedia: Memory Management [http://en.wikipedia.org/wiki/Memory_management]
- Memory Management in Linux [http://tldp.org/LDP/tlk/mm/memory.html]
- Opengroup - mmap [http://www.opengroup.org/onlinepubs/009695399/functions/mmap.html]
- MSDN: Managing Virtual Memory in Win32 [http://msdn.microsoft.com/en-us/library/ms810627.aspx]
- MSDN: Managing Memory-Mapped Files in Win32 [http://msdn2.microsoft.com/en-us/library/ms810613.aspx]
- MSDN: Structured Exception Handling [http://msdn2.microsoft.com/en-us/library/ms680657.aspx]
- Utilizarea vectorilor de excepție (Windows) [http://msdn.microsoft.com/en-us/library/windows/desktop/ms681411(v=vs.85).aspx]