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

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 cum ar fi placa video.

Concepte teoretice

Dimensiunea spațiului de adresă virtual al unui proces depinde de dimensiunea registrelor procesorului. Astfel, pe un sistem cu 32-bit un proces va putea accesa 2^32 = 4GB spațiu de memorie (pe de altă parte, pe un sistem de 64-bit va accesa teoretic 2^64). 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 (3G userspace vs 1G kernelspace).

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 4K pe un sistem de 32-bit sau 64-bit).

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](http://www.opengroup.org/onlinepubs/009695399/functions/malloc.html)(4150);

DIE(p == NULL, "malloc failed");

Considerând că o pagină virtuală are 4K = 4096 octeți, atunci apelul malloc va aloca 4096 octeți + 54 octeți = 4K + 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 sistembrk, 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 sistembrk. Acesta prealocă un spațiu mai mare, iar viitoarele apeluri malloc vor folosi acest spațiu. În acest fel, apelurile malloc vor fi eficiente: nu vor face apel de sistem, nu vor face alocare efectivă de spațiu fizic sau virtual, 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* (adică 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 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, majoritatea dispozitivelor nu permit decât accesul secvențial (ex. char device) și sunt incompatibile din această cauză 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 parametrulstart 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 altă adresă apropiată multiplu de dimensiunea unei pagini. Adresa la care se mapează fișierul este întoarsă de funcție.

Parametrul prot specifică tipul de acces care se dorește; poate fi PROT\_READ (citire), PROT\_WRITE (scriere),PROT\_EXEC (execuție) sau PROT\_NONE; dacă zona e folosită altfel decât s-a declarat se va genera un semnalSIGSEGV.

Parametrul flags permite stabilirea tipului de mapare ce se dorește; poate lua următoarele valori (combinate prin SAU pe biți; trebuie să existe cel puțin una: fie MAP\_PRIVATE, fie 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 (în acest fel toate procesele care au realizat mapări vor vedea modificările); pentru ca modificările să fie vizibile și 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 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 (argumentele fd și offset sunt ignorate)

Este de remarcat 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 după aceea funcția se va termina.
* MS\_ASYNC - este inițiată secvența de salvare, dar nu se așteaptă terminarea ei.
* MS\_INVALIDATE - se invalidează mapările zonei din alte procese, pentru a forța recitirea paginii în toate celelalte procese la următorul acces.

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 de flag-uri 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, 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 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.

Detalii despre madvise

[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);

Detalii despre mlock şi mlockall

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 sauSIGBUS procesului. După cum am văzut atunci când am discutat despre semnale, semnalul poate fi tratat cu două tipuri de funcții pe care aici o să le denumim signal și [sigaction](http://linux.die.net/man/2/sigaction). Funcția de tip [sigaction](http://linux.die.net/man/2/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 sauBUS\_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 freecu 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 1-4K, 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 peste limita inferioară.

Pentru a putea verifica și astfel de situații, utilizatorul trebuie să definescă variabila de mediuEF\_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 un moment de timp ulterior, când va fi mai greu să determine cauza erorii 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-editarea bibliotecii libefence. Pentru a vedea utilitatea acestui pachet, să analizăm programul de mai jos:

[ef\_example.c](http://ocw.cs.pub.ro/courses/_export/code/so/laboratoare/laborator-06?codeblock=10)

*#include <stdio.h>*

*#include <malloc.h>*

int main(void)

{

int i;

int \*data\_1, \*data\_2;

data\_1 = [malloc](http://www.opengroup.org/onlinepubs/009695399/functions/malloc.html)(11 \* sizeof(int));

for (i = 0; i <= 11; i++)

data\_1[i] = i;

data\_2 = [malloc](http://www.opengroup.org/onlinepubs/009695399/functions/malloc.html)(11 \* sizeof(int));

for (i = 0; i <= 11; i++)

data\_2[i] = 11 - i;

for (i = 0; i <= 11; i++)

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("%d %d\n", data\_1[i], data\_2[i]);

[free](http://www.opengroup.org/onlinepubs/009695399/functions/free.html)(data\_1);

[free](http://www.opengroup.org/onlinepubs/009695399/functions/free.html)(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 efence 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. Funcțiile [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) sunt prezentate mai în detaliu [aici](http://ocw.cs.pub.ro/courses/so/laboratoare-2013/laborator-05#memorie_partajata_filemapping).

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 funcțieiMapViewOfFile. 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.

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:

Descrierea parametrilor funcţiei VirtualAllocEx

* 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 foloseșteMEM\_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 apelulVirtualAlloc sau VirtualAllocEx folosind MEM\_RESERVE. Paginile nu pot fi localizate în regiuni adiacente rezervate prin apeluri separate ale VirtualAlloc sau VirtualAllocEx folosindMEM\_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 nici o 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 șiProtect 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 sau nu (MEM\_PRIVATE).

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]) 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 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 excepții cauzate de un acces invalid la memorie, ExceptionCode va fi setat laEXCEPTION\_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 0dacă s-a efectuat o operație de citire sau 1 dacă s-a efectuat o operație de scriere. A doua intrare dinExceptionInformation 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 esteExceptionInformation 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 întoarcăEXCEPTION\_CONTINUE\_EXECUTION, dacă excepția a fost tratată și se dorește continuarea execuției, sauEXCEPTION\_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

Exercițiul 0 - Joc interactiv (2p)

* Detalii desfășurare [joc](http://ocw.cs.pub.ro/courses/so/meta/notare#joc_interactiv).

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 (9p)

Exercițiul 1 - Investigarea mapărilor folosind pmap (0.5p)

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. De ce unele zone de biblioteci sunt mapate cu drept de scriere?

Analizați mapările făcute de procesul init folosind comanda:

**sudo** pmap 1

Exercițiul 2 - Scrierea în fișier - write vs. mmap (1p)

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

Care timp este mai mare și de ce? (Hint: folosiți [strace](http://linux.die.net/man/1/strace) pentru a vedea câte apeluri de sistem se realizează pentru rularea fiecărui program)

Ce se întâmplă dacă în programul mmap.c schimbați flagul de creare al memoriei partajate din MAP\_SHARED înMAP\_PRIVATE? (Hint: [man mmap](http://linux.die.net/man/2/mmap), în special secțiunea despre MAP\_PRIVATE).

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.

Ce conține fișierul 'test\_mmap' în cele două cazuri? Rețineți faptul că apelul mmap folosește una dintre opțiunileMAP\_SHARED sau MAP\_PRIVATE (una singură).

Observăm că în cazul mapării fișierului cu flagul MAP\_SHARED, fișierul test\_mmap conține textul*Testing testing 123…* de 10000 de ori, în schimb în cel de-al doilea caz (cu flagul MAP\_PRIVATE) fișierul este gol deoarece modificările nu se propagă pe disc.

Exercițiul 3 - Detectare 'buffer underrun' folosind ElectricFence (1p)

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

**make**

./bug

Folosiți [ElectricFence](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-06#electricfence) pentru a prinde situația de 'buffer underrun' urmărind pașii:

* Instalati pachetul electric-fence in cazul in care biblioteca libefence.so nu se gaseste pe sistem.
* Setați în bash următoarea variabila 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

Explicați de ce bug-ul nu s-a manifestat anterior urmărind exemplul din secțiunea [ElectricFence](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-06#electricfence).

Exercițiul 4 - Copierea fișierelor folosind mmap (2p)

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.
* Folosiți MAP\_SHARED pentru mapare pentru a fi transmise schimbările în fișier: rețineți faptul că apelul mmap folosește una dintre opțiunile MAP\_SHARED și 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](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-06#maparea-fisierelor).

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). De ce credeți că se folosește această variantă?

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. (Vezi [link](http://stackoverflow.com/a/27987994)).

Exercițiul 5 - Tipuri de acces pentru pagini (3p)

Intrați în directorul 5-prot și inspectați sursa prot.c. Creați trei zone de memorie în spațiul de adresă, cu drepturi de citire, scriere, respectiv nici un drept. Zonele vor avea dimensiunea de o pagină. Testați comportamentul programului când se fac accese de citire și scriere în aceste zone. Completați comentariile cuTODO 1.

Adăugați un handler de tratare a excepțiilor care să remapeze zonele cu protecție de citire și scriere la generarea excepțiilor. Completați comentariile cu TODO 2.

**Atenție:**Este precizat și în comentariile sursei, trebuie să ștergeți prima linieold\_action.sa\_sigaction(signum, info, context); pentru a putea rezolva a doua parte a exercițiului.

Exercițiul 6 - Page fault-uri (0.5p)

Intrați în directorul 6-faults și urmăriți conținutul fișierului fork-faults.c. **Câte page-fault-uri** credeți că se realizează la rulare?

Vom folosi utilitarul pidstat din pachetul sysstat care permite monitorizarea page fault-urilor unui proces (un mic tutorial despre cum poate fi folosit pidstat [găsiți aici](http://www.cyberciti.biz/open-source/command-line-hacks/linux-monitor-process-using-pidstat)). Rulați programul fork-faults. Folosiți ENTERpentru a continua programul și executați comanda

pidstat -r -T ALL -p $(**pidof** fork-faults) 5 100

pentru a urmări page fault-urile.

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

Comanda de mai sus vă afișează câte un mesaj la fiecare 5 secunde. Sincronizați apăsarea tastei ENTER cu afișajul 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.

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 etc. Detalii puteți afla [de aici](http://www.cyberciti.biz/tips/identifying-linux-bottlenecks-sar-graphs-with-ksar.html).

Exercițiul 7 - Blocarea paginării (1p)

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.

* Revedeți secțiunea referitoare la [blocarea paginării](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-06#blocarea_paginarii) precum și [pagina de manual](http://linux.die.net/man/2/mlock).

Cât de mare poate fi *DATA\_SIZE*? Încercați cu diverse valori și explicați comportamentul.

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.

Memoria blocată este prin definiție memorie rezidentă - nu poate fi trimisă pe swap. Puteți urmări cum se modifică dimensiunea memoriei rezidente (cea care nu poate fi trimisa pe swap) folosind după fiecare pas al programului următoarea comandă:

**ps** -p $(**pidof** paging) -o pid,rss,vsz,**comm**

Bonus Linux

1 so karma - 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 înPROT\_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 și recompilați sursa. Ce se întâmplă și de ce?

Windows (9p)

Exercițiul 1 - Maparea memoriei (0.5p)

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 Mem Usage și Page Faults (pentru a vedea o listă completă cu coloanele care pot fi activate, Task Manager→View→Select Columns).

Exercițiul 2 - Crearea unor rutine în mod dinamic (1p)

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 conțintul 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, etc).

Exercițiul 3 - Mapare fișiere în memorie (1.5p)

Să se scrie un program care copiază un fișier folosind proiectul 3-copy. Programul primește ca argumente numele fișierului sursă, numele fișierului desținație, mapează în memorie cele două fișiere și copiază conținutul primului fișier folosind memcpy(3). Pentru aflarea lungimii unui fișier folosiți [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 (3p)

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

Să se creeze trei zone de memorie în spațiul de adresă, cu drepturi de citire, scriere, respectiv niciun drept. Zonele vor avea dimensiunea de o pagină. Să se testeze comportamentul programului când se fac accese de citire și scriere în aceste zone. Urmăriți comentariile cu TODO 1. (în Windows se poate apela [VirtualProtect](http://msdn.microsoft.com/en-us/library/aa366898%28v=VS.85%29.aspx)doar pentru o zonă de memorie alocată cu [VirtualAlloc](http://msdn.microsoft.com/en-us/library/aa366887%28VS.85%29.aspx)).

Adăugați un handler de tratare a excepțiilor care să remapeze zonele cu protecție de citire și scriere la generarea excepțiilor (revedeți secțiunea despre [Schimbarea protecției unei zone mapate](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-06#schimbarea_protectiei_unei_zone_mapate1)). Pentru implementarea handlerului, revedeți secțiunea referitoare la [Excepții](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-06#exceptii1) și urmăriți comentariile cu TODO 2.

Exercițiul 5 - Detectare 'buffer overrun' - implementare utilitar asemănător cu Electric Fence (2p)

Încărcați proiectul 5-ef și inspectați sursa, ignorând pentru moment funcția MyMalloc. Compilați și rulați proiectul. Ar trebui să apară erori?

Completați funcția MyMalloc astfel încât orice depășire a bufferului alocat să producă eroare (urmăriți commentariile 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 (1p)

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. Nu uitați că adresa trebuie aliniată la limita unei pagini.

* Revedeți secțiunea referitoare la [blocarea paginării în Windows](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-06#blocarea-paginarii1).

Cât de mare poate fi *DATA\_SIZE*? Încercați cu diverse valori și explicați comportamentul. Urmăriți comportamentul programului în Task Manager.

EXTRA

Comparați timpii de execuție ai algoritmilor de numărare a liniilor din fișier, aflați în această [arhivă](http://ocw.cs.pub.ro/courses/_media/so/laboratoare-2013/lines_counter.zip)

* Cât de performantă este metoda cu mapare a fișierului în memorie?
* Care sunt cele mai importante diferențe între metoda mmap din modulul de Python cu același nume și funcția nativă din Linux?